Numéro d'ordre : 1322

Année : **1994**

Laboratoire d'Informatique Fondamentale de Lille

THÈSE

présentée à

L'UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE

pour obtenir le titre de

DOCTEUR EN INFORMATIQUE

par

Lenneke Dekker

FROME : Représentation multiple et classification d'objets avec points de vue

Thèse soutenue le 13 Juin 1994, devant la commission d'examen

R. Beuscart	Professeur	Université de Lille II, CERIM
B. Carré	Maître de Conférences	Université de Lille I, LIFL
G. Comyn	Professeur	Université de Lille I, LIFL
R. Ducournau	Ingénieur en Chef	Sema Group Montrouge (*rapporteur*)
J.-M. Geib	Professeur	Université de Lille I, LIFL
A. Napoli	Chargé de Recherche au CNRS	CRIN CNRS - INRIA Lorraine (*rapporteur*)
F. Rechenmann	Directeur de Recherche INRIA	INRIA Rhône-Alpes IMAG/LIFIA

UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE
U.F.R. d'I.E.E.A. Bât M3. 59655 Villeneuve d'Ascq CEDEX
Tél. 20.43.47.24 Fax. 20.43.65.66

UNIVERSITE DES SCIENCES
ET TECHNOLOGIES DE LILLE

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M. H. LEFEBVRE, M. PARREAU

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PARREAU, J. LOMBARD, M. MIGEON, J. CORTOIS, A.DUBRULLE

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNOLOGIES DE LILLE

M. P. LOUIS

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CHAMLEY Hervé	Géotechnique
M. CONSTANT Eugène	Electronique
M. ESCAIG Bertrand	Physique du solide
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. LABLACHE COMBIER Alain	Chimie
M. LOMBARD Jacques	Sociologie
M. MACKE Bruno	Physique moléculaire et rayonnements atmosphériques

M. MIGEON Michel	EUDIL
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilités
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BRASSELET Jean Paul	Géométrie et topologie
M. BREZINSKI Claude	Analyse numérique
M. BRIDOUX Michel	Chimie Physique
M. BRUYELLE Pierre	Géographie
M. CARREZ Christian	Informatique
M. CELET Paul	Géologie générale
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. CROSNIER Yves	Electronique
Mme DACHARRY Monique	Géographie
M. DAUCHET Max	Informatique
M. DEBOURSE Jean Pierre	Gestion des entreprises
M. DEBRABANT Pierre	Géologie appliquée
M. DECLERCQ Roger	Sciences de gestion
M. DEGAUQUE Pierre	Electronique
M. DESCHEPPER Joseph	Sciences de gestion
Mme DESSAUX Odile	Spectroscopie de la réactivité chimique
M. DHAINAUT André	Biologie animale
Mme DHAINAUT Nicole	Biologie animale
M. DJAFARI Rouhani	Physique
M. DORMARD Serge	Sciences Economiques
M. DOUKHAN Jean Claude	Physique du solide
M. DUBRULLE Alain	Spectroscopie hertzienne
M. DUPOUY Jean Paul	Biologie
M. DYMENT Arthur	Mécanique
M. FOCT Jacques Jacques	Métallurgie
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie structurale
M. FRONTIER Serge	Ecologie numérique
M. GLORIEUX Pierre	Physique moléculaire et rayonnements atmosphériques
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie-Physique
M. GRANELLE Jean Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILBAULT Pierre	Physiologie animale
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. HENRY Jean Pierre	Génie mécanique
M. HERMAN Maurice	Physique spatiale
M. LACOSTE Louis	Biologie Végétale
M. LANGRAND Claude	Probabilités et statistiques

M. LATTEUX Michel	Informatique
M. LAVEINE Jean Pierre	Paléontologie
Mme LECLERCQ Ginette	Catalyse
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique atomique et moléculaire
M. LEROY Jean Marie	Spectrochimie
M. LHENAFF René	Géographie
M. LHOMME Jean	Chimie organique biologique
M. LOUAGE Françis	Electronique
M. LOUCHEUX Claude	Chimie-Physique
M. LUCQUIN Michel	Chimie physique
M. MAILLET Pierre	Sciences Economiques
M. MAROUF Nadir	Sociologie
M. MICHEAU Pierre	Mécanique des fluides
M. PAQUET Jacques	Géologie générale
M. PASZKOWSKI Stéfan	Mathématiques
M. PETIT Francis	Chimie organique
M. PORCHET Maurice	Biologie animale
M. POUZET Pierre	Modélisation - calcul scientifique
M. POVY Lucien	Automatique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. RAMAN Jean Pierre	Sciences de gestion
M. SALMER Georges	Electronique
M. SCHAMPS Joël	Spectroscopie moléculaire
Mme SCHWARZBACH Yvette	Géométrie
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
M. SLIWA Henri	Chimie organique
M. SOMME Jean	Géographie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. THIEBAULT François	Sciences de la Terre
M. THOMAS Jean Claude	Géométrie - Topologie
M. THUMERELLE Pierre	Démographie - Géographie humaine
M. TILLIEU Jacques	Physique théorique
M. TOULOTTE Jean Marc	Automatique
M. TREANTON Jean René	Sociologie du travail
M. TURRELL Georges	Spectrochimie infrarouge et raman
M. VANEECLOO Nicolas	Sciences Economiques
M. VAST Pierre	Chimie inorganique
M. VERBERT André	Biochimie
M. VERNET Philippe	Génétique
M. VIDAL Pierre	Automatique
M. WALLART Françis	Spectrochimie infrarouge et raman
M. WEINSTEIN Olivier	Analyse économique de la recherche et développement
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ABRAHAM Francis	
M. ALLAMANDO Etienne	Composants électroniques
M. ANDRIES Jean Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BALL Steven	Génétique
M. BART André	Biologie animale
M. BASSERY Louis	Génie des procédés et réactions chimiques
Mme BATTIAU Yvonne	Géographie
M. BAUSIERE Robert	Systèmes électroniques
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique atomique et moléculaire
M. BERNAGE Pascal	Physique atomique, moléculaire et du rayonnement
M. BERTHOUD Arnaud	Sciences Economiques
M. BERTRAND Hugues	Sciences Economiques
M. BERZIN Robert	Analyse
M. BISKUPSKI Gérard	Physique de l'état condensé et cristallographie
M. BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie végétale
M. BOHIN Jean Pierre	Biochimie métabolique et cellulaire
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie civil
M. BOIVIN Jean Claude	Spectrochimie
M. BOUCHER Daniel	Physique
M. BOUQUELET Stéphane	Biologie appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BROCARD Jacques	Chimie
Mme BROUSMICHE Claudine	Paléontologie
M. BUISINE Daniel	Mécanique
M. CAPURON Alfred	Biologie animale
M. CARRE François	Géographie humaine
M. CATTEAU Jean Pierre	Chimie organique
M. CAYATTE Jean Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie structurale
M. CHIVE Maurice	Composants électroniques optiques
M. COMYN Gérard	Informatique théorique
Mme CONSTANT Monique	Composants électroniques et optiques
M. COQUERY Jean Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique nucléaire et corpusculaire
M. COUTURIER Daniel	Chimie organique
M. CRAMPON Norbert	Tectolique géodynamique
M. CURGY Jean Jacques	Biologie
M. DANGOISSE Didier	Physique théorique
M. DE PARIS Jean Claude	Analyse
M. DECOSTER Didier	Composants électroniques et optiques
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean Paul	Informatique
M. DELORME Pierre	Physiologie animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
Mme DEMUYNCK Claire	Physique atomique, moléculaire et du rayonnement
M. DENEL Jacques	Informatique
M. DEPREZ Gilbert	Physique du solide - christallographie

M. DERIEUX Jean Claude	Microbiologie
M. DERYCKE Alain	Informatique
M. DESCAMPS Marc	Physique de l'état condensé et cristallographie
M. DEVRAINNE Pierre	Chimie minérale
M. DEWAILLY Jean Michel	Géographie humaine
M. DHAMELINCOURT Paul	Chimie physique
M. DI PERSIO Jean	Physique de l'état condensé et cristallographie
M. DUBAR Claude	Sociologie démographique
M. DUBOIS Henri	Spectroscopie hertzienne
M. DUBOIS Jean Jacques	Géographie
M. DUBUS Jean Paul	Spectrométrie des solides
M. DUPONT Christophe	Vie de la firme
M. DUTHOIT Bruno	Génie civil
Mme DUVAL Anne	Algèbre
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FARVACQUE Jean Louis	Physique de l'état condensé et cristallographie
M. FAUQUEMBERGUE Renaud	Composants électroniques
M. FELIX Yves	Mathématiques
M. FERRIERE Jacky	Tectonique - Géodynamique
M. FISCHER Jean Claude	Chimie organique, minérale et analytique
M. FONTAINE Hubert	Dynamique des cristaux
M. FORSE Michel	Sociologie
M. GADREY Jean	Sciences économiques
M. GAMBLIN André	Géographie urbaine, industrielle et démographie
M. GOBLOT Rémi	Algèbre
M. GOURIEROUX Christian	Probabilités et statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie organique
M. GUELTON Michel	Chimie physique
M. GUICHAOUA André	Sociologie
M. HAIMAN Georges	Modélisation,calcul scientifique, statistiques
M. HOUDART René	Physique atomique
M. HUEBSCHMANN Johannes	Mathématiques
M. HUTTNER Marc	Algèbre
M. ISAERT Noël	Physique de l'état condensé et cristallographie
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et statistiques
M. JEAN Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KOENIG Gérard	Sciences de gestion
M. KOSTRUBIEC Benjamin	Géographie
M. KREMBEL Jean	Biochimie
Mme KRIFA Hadjila	Sciences Economiques
M. LANGEVIN Michel	Algèbre
M. LASSALLE Bernard	Embryologie et biologie de la différenciation
M. LE MEHAUTE Alain	Modélisation,calcul scientifique,statistiques
M. LEBFEVRE Yannic	Physique atomique,moléculaire et du rayonnement
M. LECLERCQ Lucien	Chimie physique
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Marc	Composants électroniques et optiques
M. LEFEVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
M. LEGRAND Michel	Astronomie - Météorologie
M. LEGRAND Pierre	Chimie
Mme LEGRAND Solange	Algèbre
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne

M. LE MAROIS Henri	Vie de la firme
M. LEMOINE Yves	Biologie et physiologie végétales
M. LESCURE François	Algèbre
M. LESENNE Jacques	Systèmes électroniques
M. LOCQUENEUX Robert	Physique théorique
Mme LOPES Maria	Mathématiques
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU François	Sciences économiques
M. MAHIEU Jean Marie	Optique - Physique atomique
M. MAIZIERES Christian	Automatique
M. MANSY Jean Louis	Géologie
M. MAURISSON Patrick	Sciences Economiques
M. MERIAUX Michel	EUDIL
M. MERLIN Jean Claude	Chimie
M. MESMACQUE Gérard	Génie mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MOCHE Raymond	Modélisation,calcul scientifique,statistiques
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie organique
M. MORE Marcel	Physique de l'état condensé et cristallographie
M. MORTREUX André	Chimie organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
M. NIAY Pierre	Physique atomique,moléculaire et du rayonnement
M. NICOLE Jacques	Spectrochimie
M. NOTELET Francis	Systèmes électroniques
M. PALAVIT Gérard	Génie chimique
M. PARSY Fernand	Mécanique
M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. PERTUZON Emile	Physiologie animale
M. PETIT Daniel	Biologie des populations et écosystèmes
M. PLIHON Dominique	Sciences Economiques
M. PONSOLLE Louis	Chimie physique
M. POSTAIRE Jack	Informatique industrielle
M. RAMBOUR Serge	Biologie
M. RENARD Jean Pierre	Géographie humaine
M. RENARD Philippe	Sciences de gestion
M. RICHARD Alain	Biologie animale
M. RIETSCH François	Physique des polymères
M. ROBINET Jean Claude	EUDIL
M. ROGALSKI Marc	Analyse
M. ROLLAND Paul	Composants électroniques et optiques
M. ROLLET Philippe	Sciences Economiques
Mme ROUSSEL Isabelle	Géographie physique
M. ROUSSIGNOL Michel	Modélisation,calcul scientifique,statistiques
M. ROY Jean Claude	Psychophysiologie
M. SALERNO Françis	Sciences de gestion
M. SANCHOLLE Michel	Biologie et physiologie végétales
Mme SANDIG Anna Margarette	
M. SAWERYSYN Jean Pierre	Chimie physique
M. STAROSWIECKI Marcel	Informatique
M. STEEN Jean Pierre	Informatique
Mme STELLMACHER Irène	Astronomie - Météorologie
M. STERBOUL François	Informatique
M. TAILLIEZ Roger	Génie alimentaire
M. TANRE Daniel	Géométrie - Topologie
M. THERY Pierre	Systèmes électroniques
Mme TJOTTA Jacqueline	Mathématiques
M. TOURSEL Bernard	Informatique
M. TREANTON Jean René	Sociologie du travail

M. TURREL Georges	Spectrochimie infrarouge et raman
M. VANDIJK Hendrik	
Mme VAN ISEGHEM Jeanine	Modélisation,calcul scientifique,statistiques
M. VANDORPE Bernard	Chimie minérale
M. VASSEUR Christian	Automatique
M. VASSEUR Jacques	Biologie
Mme VIANO Marie Claude	
M. WACRENIER Jean Marie	Electronique
M. WARTEL Michel	Chimie inorganique
M. WATERLOT Michel	géologie générale
M. WEICHERT Dieter	Génie mécanique
M. WERNER Georges	Informatique théorique
M. WIGNACOURT Jean Pierre	
M. WOZNIAK Michel	Spectrochimie
Mme ZINN JUSTIN Nicole	Algèbre

Lenneke est disparue accidentellement le 1er Juillet 1994.

voor pappa,
mamma
en Rob

Je tiens à remercier Bernard Carré d'avoir été autant un ami qu'un directeur de thèse. Je rends hommage à sa modestie et son honnêteté scientifique. Il m'a beaucoup appris.

Je remercie Jean-Marc Geib de me faire le plaisir d'être président du jury.

Je remercie Roland Ducournau d'avoir pris tout le temps nécessaire pour rapporter cette thèse avec grande attention. Je le remercie également des nombreuses discussions que nous avons eues.

Je remercie Amedeo Napoli d'avoir accepté de rapporter ce travail avec grand soin et des nombreuses remarques qu'il m'a faites. Qu'il soit également remercié pour sa précieuse et très prompte assistance bibliographique.

Je remercie Régis Beuscart d'être examinateur de cette thèse. Qu'il trouve ici l'expression de ma profonde reconnaissance pour son accueil au CERIM et pour m'avoir permis de terminer ma thèse dans de bonnes conditions.

Je suis très honorée de la présence de Gérard Comyn. Je le remercie de m'avoir aidée à trouver un financement pour cette thèse. Je le remercie également de s'être toujours intéressé à nos travaux de sa lointaine Bavière et du temps qu'il a consacré à la lecture d'une première version de ce document.

Je remercie François Rechenmann pour l'intérêt porté à nos travaux et sa participation au jury.

Je remercie les membres du groupe ROME : Hassène, Gilles et Laurent pour les discussions que nous avons eues et également tous ceux qui contribuent à l'ambiance au LIFL.

Je tiens à associer à ces remerciements Francis qui m'a motivée pour l'informatique et qui m'a aidée à trouver le chemin de Lille III à Lille I.

Je remercie les collègues du CERIM pour leur accueil chaleureux et leur gentillesse.

Je remercie les membres du groupe "Classification" du PRC-IA et du groupe "Objets" du GDR de Programmation pour les discussions fructueuses qui ont animé les réunions.

Enfin, je remercie tous mes proches de m'avoir soutenue durant ces années. En particulier, merci à Philippe qui m'a encouragée sans relâche pendant ces dernières années.

Cette thèse a été financée en partie par la Région Nord-Pas de Calais.

Table des Matières

Introduction Générale

Dans tous les domaines d'application de l'informatique on voit apparaître un besoin croissant de manipuler des quantités importantes de connaissances. L'organisation adéquate de ces connaissances contribue à une meilleure exploitation de celles-ci par des Systèmes à Base de Connaissances (ou SBC). La Représentation des Connaissances est de ce fait un aspect important de l'étude des Systèmes à Base de Connaissances.

Plusieurs types de représentation ont été élaborés. Citons la représentation de type relationnel/logique et la représentation structurée, sous la forme d'entités structurées en termes de composantes et de caractéristiques. L'approche relationnelle met en avant les relations entre les concepts, alors que l'approche structurée privilégie les concepts.

Le cadre de ce travail est celui de la Représentation Structurée des Connaissances, qui s'est inspirée d'études sur les frames et les langages orientés objets. La richesse de cette double inspiration, parfois complétée par l'inspiration relationnelle/logique, s'est montrée dans des modèles hybrides combinant divers aspects des différentes approches.

Les problèmes étudiés dans ce cadre sont nombreux. Citons les aspects suivants :

- Aspects structurels (statiques), relationnels et fonctionnels des entités représentées.
- Aspects liés à l'abstraction et à la généralisation.
  L'étude de ces aspects a par exemple mené à des modèles distinguant un niveau générique (classe) et un niveau spécifique (instance) ; elle a également fait surgir la notion de spécialisation et la définition de hiérarchies taxonomiques.
- Aspect de l'héritage, étroitement lié au point précédent.
- Aspect de la multiplicité des représentations et des liens entre elles.
- Les formes de raisonnement qui exploitent les connaissances représentées pour en inférer de nouvelles.
- La puissance réflexive du système de représentation. Il s'agit de la capacité du système à se représenter lui-même dans ses propres termes et à influer sur ses propres mécanismes, liée directement à l'extensibilité du système : définition de nouvelles structures et mécanismes d'inférence.

Ce mémoire veut contribuer à la fois à la définition d'un modèle de représentation riche et à l'exploitation de ce modèle par un mécanisme de raisonnement.

Le modèle que nous proposons, appelé FROME, est un système de représentation des connaissances hybride frames-objets, important dans ce cadre les capacités de représentation multiple et évolutive issues du langage à objets ROME [Car89]. Nous proposons une forme d'exploitation des connaissances représentées par un système de classification d'instances adapté à ce modèle.

Le mémoire est organisé en deux parties, chacune consacrée à un des objectifs cités. La première partie étudie la représentation par objets. Dans un premier temps, nous présentons certains aspects de systèmes existants, en rapport avec notre problématique. Ensuite, nous proposons un modèle hybride de représentation, appelé FROME.

Le modèle FROME retient une structuration riche des objets, inspirée des systèmes à base de frames. La double inspiration frames/langages orientés objet s'exprime dans l'aspect comportemental, qui est composé à la fois d'éléments provenant des frames (réflexes) et de la programmation objet (méthodes). Nous retenons l'approche à base de classes et instances, plus courante en programmation objet.

Nous retenons la représentation multiple et évolutive et la notion de point de vue pour la sélection des représentations. L'héritage multiple s'inscrit naturellement dans ce cadre.

Nous montrons enfin comment les capacités réflexives d'un langage orienté objet (en l'occurrence ROME) sont exploitées directement pour la définition de notre modèle et garantissent par là même son extensibilité. Comme exemple de cette extensibilité, nous citons la définition d'un module de règles et la définition du filtrage.

La partie II étudie plus en détail les conditions et les mécanismes qui gèrent le raisonnement classificatoire dans un environnement orienté objet à base de classes et instances. Plus particulièrement, nous nous intéressons à la classification d'instances dans un graphe hiérarchisé de classes, par comparaison des caractéristiques.

Au travers de l'étude de quelques systèmes existants, nous identifions quelques problèmes concernant :

- l'appariement entre classes et objets,
- la description des objets en vue de leur classification,
- le contrôle du processus de classification.

Guidés par ces problèmes, nous proposons une forme de classification d'instances pour le modèle FROME. L'étude de l'appariement montre que celui-ci dépend fortement de l'interprétation de la notion de classe. Dans notre solution, nous intégrons deux interprétations différentes de la notion de classe. Nous proposons un moyen de décrire librement les objets à classifier. La classification elle-même est répartie sur les classes, de manière orientée objet.

Nous traitons un problème inhérent à la représentation par objet, qui est celui de la présence de caractéristiques de même nom (attributs homonymes) dans des classes différentes. La présence d'attributs homonymes peut poser des problèmes d'ambiguïté et de conflit. Le modèle FROME traite ces problèmes efficacement dans le cadre de la représentation, grâce à la notion de point de vue. Dans le cadre de la classification, le problème de l'homonymie se pose d'une nouvelle manière. Nous proposons une solution combinant l'identification complète d'attributs par sélection de point de vue et la prise en compte de l'ambiguïté par le processus même de la classification.

L'implantation de la classification se fait de nouveau par extension, ce qui montre la puissance de la métaprogrammation orientée objet.

Partie I:

La Représentation par Objets

1
Introduction

Dans cette première partie, nous nous intéressons à la notion d'objet telle qu'elle est exploitée pour représenter des connaissances, dans le domaine de l'Intelligence Artificielle. Il est bien connu que la notion d'objet a des origines et des acceptions multiples [Mas&89]. Nous essayons d'identifier (chapitre 2) les différents traits apportés par les différentes approches — aussi bien l'approche de la programmation par objets, que celle de la représentation par frames et celle des langages dits terminologiques — et de retenir ceux qui peuvent contribuer à une notion d'objet pour la représentation des connaissances. Le but n'est pas de traiter tous les aspects de chaque approche à fond, mais de dégager les notions intéressantes pour la représentation des connaissances et d'examiner certaines solutions proposées dans l'une ou l'autre des approches pour résoudre un problème commun. Pour ne citer que le plus saillant, le problème de l'héritage multiple est présent aussi bien en programmation par objets (PPO) qu'en représentation (RPO). C'est pourquoi nous serons amenés à discuter des solutions proposées par des langages de PPO comme EIFFEL et des solutions proposées en RPO (KRL, OBJLOG,...), même si les acceptions de la notion d'objet dans ces différentes approches sont assez divergentes. Certains aspects rencontrés dans des systèmes parfois classiques, que nous considérons comme originaux mais qui sont souvent passés sous silence, sont traités plus en détail.

Le chapitre 3 décrit le langage FROME, langage hybride dans lequel plusieurs traits des différentes approches examinées sont réunis pour la RPO. Ce langage s'inscrit dans le cadre conceptuel de ROME [Car89], lequel est consacré à la représentation multiple par points de vue et évolution d'objets. Parmi les traits retenus, citons la description d'attributs à l'aide de facettes, à la manière des langages de frames, la distinction classe-instance, l'héritage multiple, la description multi-points de vue et la représentation évolutive d'objets.

Dans le chapitre 4, nous montrons comment l'implantation de FROME de manière objet tire profit de la puissance de la métaprogrammation objet, rendue possible par la métacircularité du langage orienté objet sous-jacent utilisé ROME [Car&91b]. Les qualités orientées objet de FROME sont ainsi obtenues par héritage. En particulier, FROME hérite de l'extensibilité, dont nous montrons quelques exemples, comme la définition de classes d'attributs particuliers et la définition d'un module de filtrage [Dek93].

2
La représentation par objets

2.1 La notion d'objet en représentation des connaissances

La notion d'objet en programmation est issue d'un souci de modularité [Weg90][1]. Ce souci est déjà présent dans la programmation structurée, mais est poussé plus loin dans la PPO, où le module que forme l'objet regroupe données et traitements sur ces données au sein de la même entité, sous la forme d'attributs et de méthodes. L'objet en RPO, avant tout entité structurée, est issue des préoccupations de psychologues, linguistes, puis informaticiens à la recherche d'un modèle de représentation des connaissances humaines, afin de pouvoir modéliser la façon dont les humains se servent de ces connaissances (psychologie cognitive) et/ou de faire exploiter les connaissances représentées par un ordinateur (intelligence artificielle). La notion de frame [Min75] ou de schéma est à la base des développements d'une multitude de langages de représentation des connaissances [Mas&89]. Parmi les premiers conçus, citons KRL [Bob&77] et FRL [Rob&77a/b]. Dans la même mouvance de modélisation de concepts, on trouve les notions de graphes conceptuels [Sow84] et de réseaux sémantiques [Woo75] [Bra77], très proches des réseaux de frames imaginés par Minsky. La famille de langages qui descendent de KL-ONE [Bra&85] [Bra&91] est fortement inspirée des frames et des réseaux sémantiques. Ces langages, appelés langages terminologiques[2], ont une forte tendance relationnelle/logique. Ils sont essentiellement dédiés à la classification (cf. la partie II de ce mémoire).

Ces différents courants se sont inspirés mutuellement et ont donné naissance à des langages hybrides, tel que LOOPS [Bob&83], combinant des aspects de PPO (comportement des objets) avec des aspects représentationnels (attributs élaborés et enrichis de facettes de typage ou de réflexes). Une autre source d'inspiration ayant donné lieu à des hybridations [Fik&85] est celle de l'approche logique des systèmes de production. On citera le système expert CENTAUR [Aik83]. La combinaison d'objets (frames) et règles permet d'une part de représenter des bases de faits structurés plus riches et d'autre part d'exploiter les bases de connaissances que représentent les réseaux de frames interconnectés en les dotant d'un mécanisme de raisonnement. Les langages hybrides se trouvent donc à l'intersection de plusieurs paradigmes et empruntent tantôt aux uns tantôt aux autres. Selon les emprunts, nous trouvons des langages avec une notion d'objet plus ou moins structuré, plus ou moins actif par un comportement propre, plus ou moins manipulable par des mécanismes de raisonnement extérieurs.

Suivant les thèmes que nous abordons dans ce chapitre, nous plaçons les différents

1. "*[...] The software crisis of the late 1960s led to a change of goals in language design, away from expressive power and towards program structure. [...] At the macro level there was greater emphasis on modularity, first in terms of functions and procedures and later in terms of objects and data abstraction.*"
2. Ces langages sont aussi connus sous le nom de langages à subsomption de termes ou encore langages de description de concepts. On y classe entre autres : KRYPTON, NIKL, KL-TWO, CLASSIC, KANDOR, BACK, LOOM, MESON, SB-ONE, OMEGA.

langages passés en revue sur un schéma triangulaire. En fonction des distances d'un langage par rapport à chacun des sommets, celui-ci possède plus ou moins de caractéristiques "type" de l'approche représentée par ce sommet. Cette schématisation reste volontairement approximative. Elle a pour but de donner une idée de la proximité entre les différents langages, plutôt que de les classer de façon stricte. Aucun de ces schémas ne saurait prétendre à l'exhaustivité.

La Fig. 1 donne un schéma général indiquant la situation des différents langages par rapport aux domaines de la PPO, de la RPO et de la PPL (programmation par la logique). Près du centre se trouvent les langages dits hybrides et les systèmes intégrant programmation et représentation par objets et règles de production.

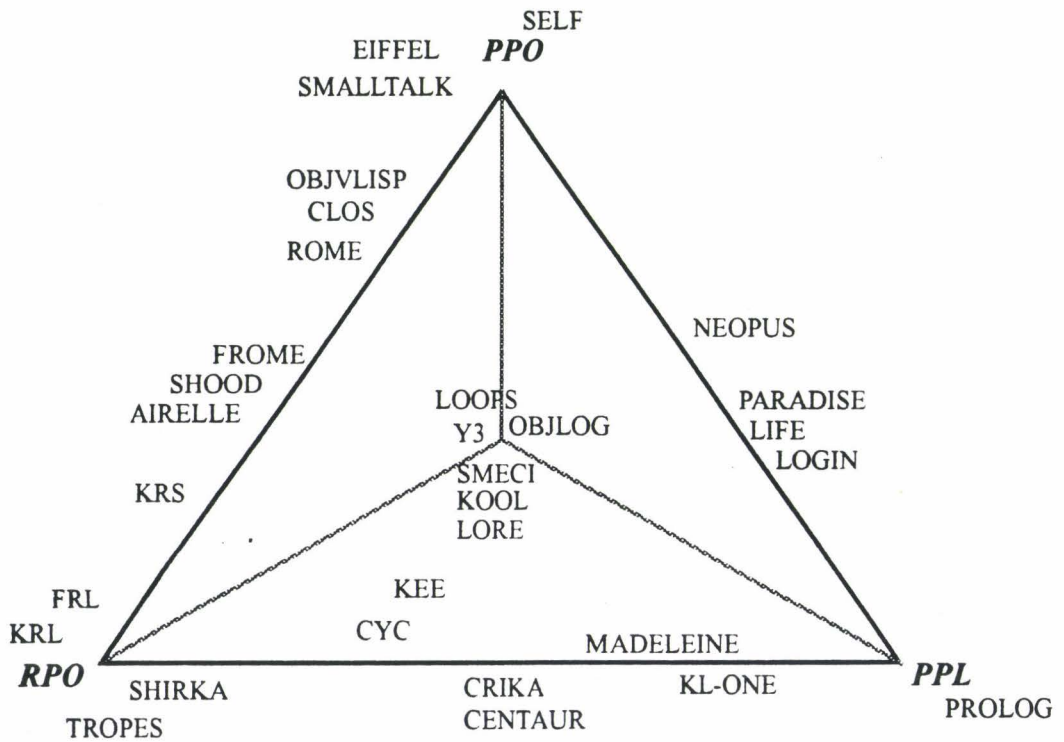


Fig. 1 Langages des domaines de la PPO, RPO et PPL

Nous signalons les travaux de Aït-Kaci sur LogIn [Aït&86], LIFE [Aït91b], et Paradise [Aït91a], fortement orientés par la logique, qui présentent une combinaison des paradigmes objet (types structurés appelés ψ-termes), fonctionnel et logique (relations et contraintes).

Le projet CYC [Len&86] [Guh&90], dont l'objectif est la construction d'une immense Base de Connaissances, capable de s'auto-enrichir par une panoplie de mécanismes d'inférence, se déplace d'une représentation structurelle vers une représentation de plus en plus logique (1er ordre).

2.1.1 Structure

2.1.1.1 Attributs, variables d'instance, slots

L'entité structurée que nous appelons ici objet, s'appelle *object* (Smalltalk, LOOPS), *frame*

(FRL), *schéma* (Shirka), *unit* (KRL), *prototype*, *modèle* (Yafool [Duc91]), *concept* (KRS [Mar88], KL-ONE), *classe*, ... tous niveaux de généricité confondus. Les propriétés de ces entités sont appelées *attributs*, *variables d'instances* (ou *de classe*), *slots*, *roles*, *subjects*, *champs*...

Dans l'approche PPO, il s'agit souvent d'objets *instances* d'une *classe*. Cette dernière définit la structure de ses instances sous la forme d'une liste de variables d'instances ou d'attributs. Ces variables d'instances peuvent être typées (*features* d'EIFFEL [Mey88]) ou non typées comme en Smalltalk [Gol&83] et ObjVlisp [Coi87] et en général dans tous les langages à base de lisp.

Dans l'approche RPO, on trouve souvent des attributs plus structurés. La valeur de l'attribut pour une instance donnée devient un aspect particulier de cet attribut, parmi d'autres informations telles que sa ou ses valeurs par défaut, des moyens de calcul, des annotations de documentation, des restrictions de son type ou des contraintes plus complexes sur les valeurs admissibles. Ces informations sont associées à l'attribut dans des *facettes* (Shirka), *aspects* (Objlog [Dug88]), ou *descripteurs* (KRL), au niveau de sa classe, ou directement dans l'objet dans les systèmes à base de prototypes. Ce sont autant de méta-informations [Mae87a] portant sur l'attribut lui-même, destinées à le décrire et à le contrôler.

2.1.1.2 Le typage

A l'image du typage des variables dans les langages procéduraux, les variables d'instances et les arguments des méthodes sont typés dans nombre de langages de PPO, tels Simula, EIFFEL, BETA [Kri&87], et de façon générale tous les langages compilés de la lignée désignée par le nom "Ecole Scandinave" dans [Mas&89]. Ces langages pratiquent un typage fort, c'est à dire que le compilateur peut garantir que le programme s'exécutera sans erreur de typage [Car&85]. A la différence des langages à typage statique, pour lesquels une analyse statique du programme suffit pour déterminer le type de chaque expression, les langages à typage fort ont recours à la liaison dynamique pour déterminer les types effectifs lors de l'exécution. A l'opposé, le typage est absent dans les langages interprétés comme Smalltalk. Des propositions pour étendre Smalltalk au typage ont été faites, entre autres dans [Joh&86], principalement dans un but d'optimisation de code (efficacité). De même, un langage comme CLOS [Kee89] offre une *slot option* (facette) *:type*, qui peut servir à la compilation, mais le langage n'effectue aucun contrôle de type.

Les langages de frames, tels que FRL, KRL, SHIRKA [Rec&90], disposent d'un moyen puissant pour associer des informations aux attributs : les facettes. Parmi les facettes dédiées à toutes sortes d'informations sur les attributs, certaines servent à la spécification des valeurs susceptibles de "remplir" un attribut, décrivant le type, le domaine, ou les contraintes devant être respectées (cf. Tableau 1) .

Ensuite, des langages comme PIE [Gol&80], KEE [Fik&85] et LOOPS, s'inspirant des deux approches RPO et PPO, reprennent les facettes (*annotations*) pour enrichir la description des attributs (variables de classe et d'instance en LOOPS et *memberslots* et *ownslots* en KEE). PIE est une extension de Smalltalk, proposant des perspectives multiples, l'héritage multiple (cf. 2.3) et une "métadescription" des attributs, sous la forme de valeurs par défaut, contraintes (typage) et attachements procéduraux. Les facettes de LOOPS sont stockées sur une liste de propriétés associées à un attribut. Elles peuvent être utilisées pour garder un historique des

valeurs de l'attribut, exprimer des contraintes, des degrés de certitude, pour documenter ou pour typer les attributs [Ste&86] [Ste&85]. Les facettes ne sont pas limitées à une liste prédéfinie et n'ont pas de sémantique a priori. La facette *datatype* par exemple, doit être exploitée par les méthodes d'affectation associées à des valeurs actives (cf. 2.1.2). En FRL, les slots ne sont pas typés, mais on peut y associer des contraintes sous la forme d'une expression (lisp) en valeur de la facette *$REQUIRE* [Rob&77a/b]. Le système de planification de réunions NUDGE [Gol&77] (implanté en FRL) ajoute de plus la facette *$PREFER*, qui permet de spécifier des contraintes faibles, "négociables".

Tableau 1 : Facettes liées au typage, contraintes et valeurs par défaut

	FRL [Rob& 77a/b]	KRL [Bob&77]	LOOPS [Bob83]	KEE [[Int85]	SHIRKA [Rec&90]	YAFOOL [Duc91]	KRS [Mar88]	OBJLOG [Dug88]
type		a/an ... with	datatype[a]	valueclass[b]	$un $liste-de	un des	a/an	domaine/ domaine-ref[c]
contraintes	$value $require	the ... from which or xor not <set specif[d]>		cardina- lity.min cardina- lity.max	$valeur $domaine $intervalle $a-verifier $card-min $card-max	si-possible regle- possible[e]		valeur/ valeur-ref contraintc- cardinalite
defaut	$default	default[f]		value	$defaut	value		defaut/ defaut-ref

a. *property annotation*, associée avec une valeur active pour le contrôle de type[Ste&86]
b. cette facette peut prendre comme valeur une classe, intervalle de valeurs, ou une expression construite à l'aide d'une panoplie de constructeurs : ONE.OF, NOT.ONE.OF, NOT.IN, UNION, INTERSECTION, SUBCLASS.OF, MEMBERP (une fonction retournant t ou ()).
c. *domaine* indique un type simple prédéfini, *domaine-ref* un type objet, instance d'une classe.
d. Spécification d'ensemble avec les descripteurs : SetOf, Items, NotItems, AllItems, ListOf, Sequence.
e. YAFLOG [Duc91]
f. annotation associée à n'importe quelle description.

Le domaine des valeurs que peut prendre un attribut est exprimé primitivement par une expression lisp (comme en FRL) attachée à une seule facette, par une expression composée à l'aide de constructeurs, toujours attachée à une seule facette, comme en KEE (cf. note b. page 20), par un ensemble de facettes prédéfinies (SHIRKA, OBJLOG), ou encore par un ensemble extensible de facettes, nécessairement accompagnées des méthodes d'accès aux attributs qui les prennent en compte. Ainsi, la réification des attributs (implantation sous forme d'objets instances d'une classe) en PTITLOO [Fer89] favorise la définition de nouvelles classes d'attributs, pouvant introduire des facettes particulières, en plus de l'unique facette de typage prédéfinie *range*. OBJLOG+ [Fau91], fondé sur la réification comme principe de l'extensibilité, propose une trentaine de facettes (*aspects*) prédéfinies, dont une dizaine contraignantes. Les procédures d'exploitation et d'interaction de ces facettes sont extrêmement élaborées. Leur maîtrise et la définition de nouvelles facettes ne s'en trouvent pourtant pas facilitées pour l'utilisateur. Plutôt que de multiplier le nombre de facettes, [Nic92] élabore une notation de domaines, destinée à exprimer le domaine d'un attribut dans

une description unique attachée à une seule facette, à la manière de KEE. La description unique d'un domaine permet l'introduction de règles de simplification et de comparaison de domaines. Les domaines sont construits récursivement à partir de types primitifs et structurés, et de types construits à l'aide de constructeurs, de combinateurs et d'introducteurs de contraintes. Cette approche est implantée en AIRELLE [Ada90][Ada&88], où les types, les constructeurs et les combinateurs[1] sont réifiés. Chaque opérateur a une méthode de vérification propre. Les combinateurs combinent les résultats des différentes méthodes selon leur sémantique propre pour décider de l'appartenance d'une valeur à un domaine.

Nous pouvons constater que le typage en PPO est issu d'un souci de fiabilité et d'efficacité, en ce qui concerne les langages à typage fort, les langages à typage faible — s'ils s'intéressent déjà au typage — visant surtout l'efficacité (CLOS, Smalltalk typé). L'approche RPO, à travers l'éventail de facettes et de constructeurs de type, offre une plus grande richesse et souplesse d'expression, liée avant tout à un souci de description. Pratiquement, les qualités recherchées pour le génie logiciel ne sont pas les mêmes que celles recherchées pour la représentation des connaissances.

2.1.1.3 Les relations structurées

Les attributs ou *roles* des langages de la famille KL-ONE sont exprimés comme des relations entre concepts. Dans la plupart de ces langages, les relations sont définies séparément des concepts, à l'aide d'une primitive *define-role*, comme en CLASSIC [Bor&89] ou *defrelation* (NIKL, LOOM)[Kac&86][McG88]. L'importance des relations, au même niveau que les concepts, vient des réseaux sémantiques, qui sont à l'origine de KL-ONE. Ces réseaux sont d'abord des graphes, constitués d'un ensemble de nœuds, représentant des entités, et d'un ensemble de liens, représentant les propriétés de ces entités et les relations entre elles. [Bra77] soulève le problème de l'insuffisance des simples liens (relations binaires) pour décrire des attributs ou des parties de concepts. Il introduit un nœud particulier de définition d'attributs et de parties, appelé *role description node*. Le lien primitif *DATTRS* relie un concept à ses

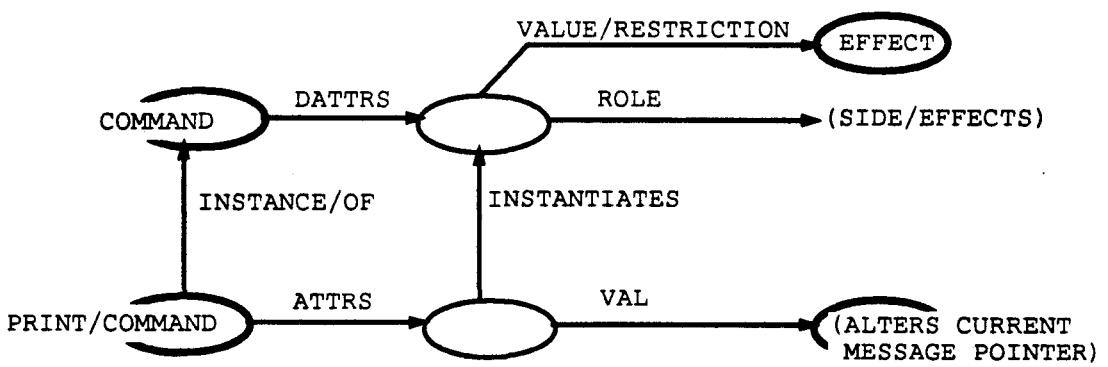


Fig. 2 Réseau sémantique avec nœuds de description de rôles [Bra77].

1. Les combinateurs définis en Airelle sont : `ouvert`, `ferme` (intervalle), `enumere`, `et`, `ou`, `ltype` (liste d'éléments typés), `vtype` (vecteur d'éléments typés), `etoile` (structures dont tous les éléments ont le même type), `cardinal`, `atype` (arbres dont toutes les feuilles ont le même type). [Ada&88]

nœuds de définition d'attributs. Divers liens partant de ces nœuds particuliers permettent de caractériser les attributs. Certains liens expriment des relations de spécialisation ou de redéfinition entre nœuds de description de rôles, d'autres caractérisent (des contraintes sur) la valeur de l'attribut représenté, à la manière des langages de frames (VALUE/RESTRICTION, NUMBER). Le lien VAL indique la valeur d'un attribut d'une instance de concept.La distinction entre le niveau de description (associé au lien DATTRS) et le niveau assertionnel (le lien ATTRS : affirme l'existence de valeurs particulières pour une instance particulière) préfigure sa généralisation en *TBox* et *ABox*, introduite plus tard avec KRYPTON [Bra&83].

La structuration des rôles en facettes est retenue en KL-ONE et reprise par tous ses successeurs. Leur statut en tant que concept indépendant est renforcé en NIKL, qui ordonne les relations en une hiérarchie explicite, exploitée pour la classification[1] [Kac&86]. Le formalisme de description a glissé de la représentation purement graphique des réseaux sémantiques vers une description logique, à l'aide d'opérateurs de construction de termes (formant la composante TBox des langages). Un terme définit un concept ou un rôle. Les opérateurs de construction permettent d'introduire de nouveaux termes ou de les former par combinaison ou restriction de termes existants. L'ensemble des opérateurs détermine l'expressivité du langage de définition. De nombreux travaux portent sur les problèmes de complétude et de consistance de l'algorithme de classification entraînés par une "trop" grande expressivité de la TBox [Bra&84][Neb88][Pat89][Hol&90].[2] Certains préfèrent limiter l'expressivité du langage afin de garder les bonnes propriétés pour la classification [Pat84][Vil85]. D'autres réclament au contraire une grande expressivité et rejettent les critères de consistance, complétude et complexité "dans le pire des cas" comme mesures de qualité d'un système de représentation [Doy&91].

Pour introduire plus de sémantique dans les réseaux sémantiques, [Bra77] propose la notion de "STRUCTURAL/CONDITION", destinée à décrire des relations entre rôles d'un concept. Ces relations contribuent à exprimer le sens des différents rôles dans le contexte du concept. Dans les réseaux sémantiques classiques, les liens n'ont de sémantique que leur nom. Brachman veut faire en sorte que la structure devienne porteuse de sens. Pour décrire un concept, il ne suffit pas d'énumérer ses attributs, mais il faut préciser la condition sous laquelle le tout tient ensemble et forme le concept décrit (la condition *structurelle*). Cette idée est reprise dans KL-ONE sous le nom de "role set relation" (RSR) [Lip82] ou "structural description" (SD) [Bra&85]. Les conditions structurelles ne sont rien d'autre que des contraintes faisant entervenir plusieurs attributs. La Fig. 3 montre la condition structurelle (symbolisée par un losange) attachée au concept d'arche. Elle est spécifiée par une instance "paramétrique" `Support`, qui référence à la fois les rôles `supportee` et `supporter` du concept de même nom dont elle est instance et les rôles correspondants `lintel` et `upright` du concept `Arch`. Cette *coréférence* des rôles `supportee` et `lintel` d'une part et `supporter` et `upright` d'autre part indique que les valeurs de ces rôles doivent être égales pour toute instance du concept Arch. Les instances paramétriques ressemblent aux prédicats SHIRKA (sous la forme de schémas) que l'on peut attacher à l'attribut `lui-meme` d'un schéma de classe sous la facette `$a-verifier` (cf. Tableau 1), et qui permettent de vérifier une contrainte globale, dont les paramètres peuvent être les attributs du schéma.

1. placement automatique d'un nouveau concept "au bon endroit" dans la hiérarchie (cf. partie II de ce mémoire).
2. [Neb90] dresse un tableau des résultats de complexité connus pour différents ensembles d'opérateurs.

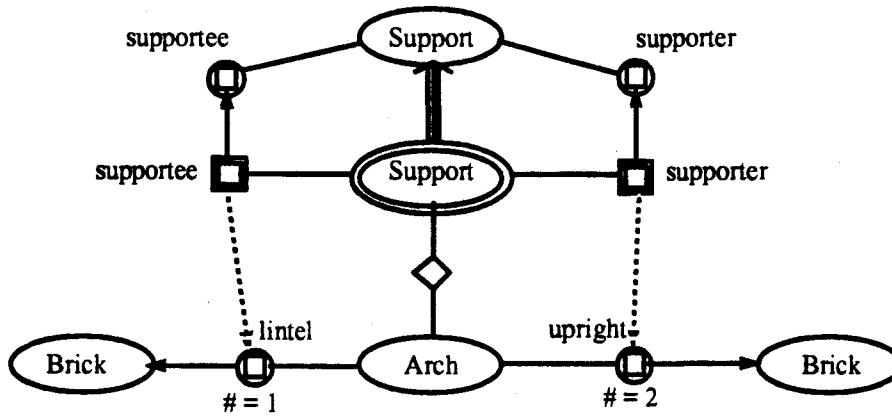


Fig. 3 Une arche : un linteau posé sur deux supports [Lip82]

A cause des problèmes qu'elles posent pour la classification, la majorité des successeurs de KL-ONE ne prennent pas en compte les conditions structurelles, ou seulement sous une forme restreinte : les "role value maps" ou RVMs, imposant l'égalité ou l'inclusion des ensembles de valeurs de deux rôles (pouvant être des compositions de rôles ou *role chains*), ou encore une forme restreinte de RVM, définie sur des rôles monovalués (CLASSIC [Bra92]). Une utilisation de RVM avec composition de rôles est illustrée dans la Fig. 4. La condition structurelle n'est pas représentée par une instance paramétrique d'un concept générique, mais relie directement le rôle `sender` et la composition de rôles `recipient ∘ immediate-supervisor` par la contrainte d'égalité.

Une tentative d'intégration des SDs sous forme de prédicats n-aires dans le système TAXON, tout en conservant la complétude pour la classification, est décrite dans [Han92]. Le regain d'intérêt actuel pour les contraintes relance les recherches sur les relations entre rôles et les contraintes globales à une classe faisant intervenir plusieurs de ses attributs [Free90] [Pan&93].

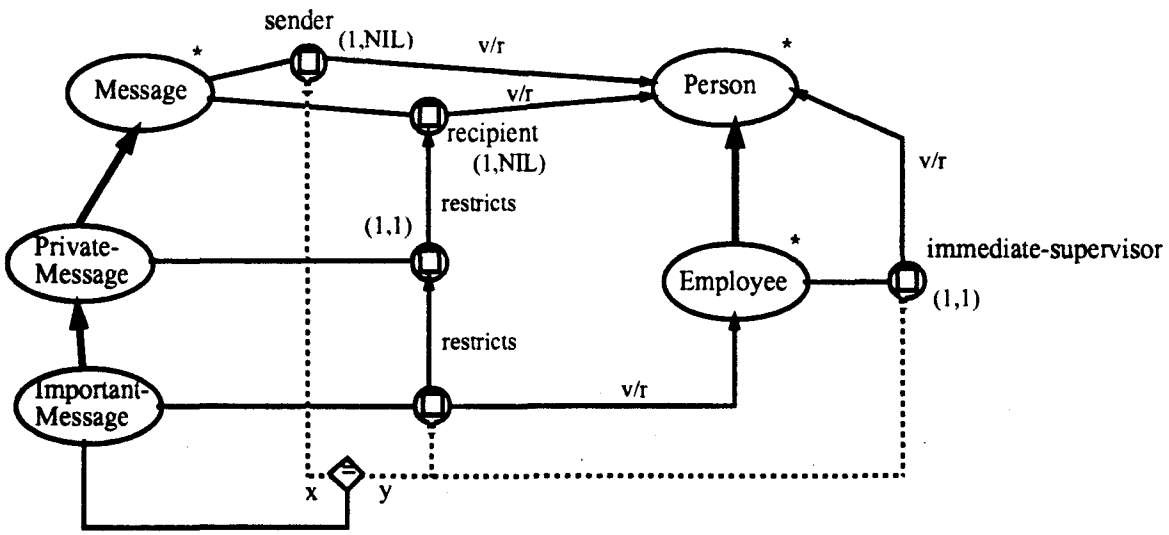


Fig. 4 "An Important-Message is a Private-Message whose recipient is an Employee, and whose sender is the same as the immediate-supervisor of its recipient"

La caractérisation des rôles étant passée de liens graphiques étiquetés à des opérateurs logiques d'introduction ou de restriction de rôle, en passant par des facettes, il est difficile d'intégrer tous les langages de la famille KL-ONE dans le tableau des facettes de typage et de contraintes de la page 20. Nous pouvons néanmoins tenter d'établir un tableau résumant les formalismes de restriction de rôles de plusieurs de ces langages (cf. Tableau 2). Une comparaison des ensembles d'opérateurs de différents langages est présentée sous un formalisme unique dans [Neb90]. Les quatre facettes de KL-ONE (au moins dans sa première version [Goo79]) sont V/R pour "value restriction", introduisant une restriction de type, N/R pour "number restriction", introduisant la cardinalité, Name qui indique simplement le nom du rôle et Modality, qui permet d'indiquer s'il s'agit d'un rôle obligatoire (i.e. nécessairement valué) ou optionnel. Cette dernière est reprise par peu de systèmes. La facette $default des langages de frames n'a pas d'équivalent dans les langages à la KL-ONE.[1] On retrouve dans presque tous les langages les facettes V/R et N/R sous une forme ou une autre. Les types peuvent souvent être spécifiés à l'aide de connecteurs logiques (AND, OR, NOT), la cardinalité minimum est parfois couplée à un type, ou précisée à l'aide de quantificateurs. On trouve pour ainsi dire autant de langages que de compositions de TBox possibles. Rares sont, par contre, les langages qui permettent d'étendre l'éventail de base choisi. Le langage ProObj [Str&90][Str&89] permet non seulement de définir de nouvelles facettes, mais aussi d'y associer les méthodes nécessaires à leur prise en compte lors de la classification. Il devient ainsi possible de simuler d'autres systèmes (comme un KL-ONE simplifié ou le système BACK) par la "simple" définition des facettes correspondantes, ce qui semble particulièrement intéressant pour expérimenter différents niveaux d'expressivité des langages.

Tableau 2 Opérateurs d'introduction et de restriction de rôles

KL-ONE [Bra&85]	KRYPTON [Bra&83]	KL-TWO [Vil85]	KANDOR [Pat84]	CLASSIC [Bra92]	LOOM [McG88]	CANDIDE [Bec&89]	OMEGA [Att&86][a]
V/R	VR Generic	VRDiff CRestrict	generic all	ALL ONE-OF MIN MAX[b]	:all :range :domain[c]	ALL DOMAIN[d] VALUE	a/an with-every
N/R	NR Generic	Cmin	exists	AT-LEAST AT-MOST	:at-least :at-most :exactly	ATLEAST ATMOST EXACTLY	with with-unique
<SD>				TEST-C TEST-H			
<RVM> <RoleChain>	Role Chain			SAME-AS			
VAL			value	FILLS		VALUE	with

a. OMEGA est un langage de description logique assez proche des langages terminologiques actuels.
b. MIN et MAX spécifient des intervalles de valeurs (et non pas la cardinalité)
c. :range et :domain sont des facettes attachées à la définition d'une relation (en dehors d'un concept) ; elles décrivent le type des objets en relation. La restriction de cette relation dans le concept (qui forme le :domain) se fait à l'aide des autres facettes.
d. DOMAIN prend un type simple, une classe, ou un type construit avec les constructeurs RANGE, SET, SETDIF, COMPOSITE.

1. Elle est jugée incompatible avec l'objectif de la classification automatique : un concept pourrait être classé à des endroits différents suivant les valeurs par défaut prises en compte.

Parmi les liens de spécialisation entre rôles (comme "restricts" sur la Fig. 4), KL-ONE compte un lien intéressant de différenciation de rôles (*DIFFS*), qui permet d'éclater un attribut en plusieurs attributs spécialisés suivant le (sous)type. Ainsi, le rôle `membres` de type `Personne` et de cardinalité [1,nil] du concept `Groupe-Social` peut être différencié dans le sous-concept `Famille` en trois sous-rôles : `père`, avec une restriction de type `Homme` et de cardinalité [0,1], `mère`, avec restriction de type `Femme`, cardinalité [0,1] et `enfants`, de cardinalité [0,25]. Le rôle `membres` continue de désigner tous les membres d'une famille, qui sont aussi accessibles selon leur rôle spécifique de `père`, `mère` ou `enfants`.

L'éclatement d'attributs a été repris par TROPES [Mar91].

2.1.1.4 Les relations entre objets

Comme constaté ci-dessus, les attributs ou rôles de KL-ONE et ses successeurs sont assimilés à des relations entre concepts. Ils sont souvent déclarés séparément, indiquant le domaine et le co-domaine, comme en LOOM, et/ou réifiés comme concepts à part entière dans une hiérarchie de relations, comme en NIKL. La spécificité sémantique des relations n'est pourtant pas souvent exploitée. Les inférences possibles dues aux propriétés particulières d'une relation, comme la transitivité, la symétrie, l'anti-réflexivité, etc. sont rarement effectuées. Dans la plupart des systèmes, ces propriétés ne peuvent pas être décrites. Si quelques systèmes offrent une facette "inverse", généralement celle-ci n'est pas pleinement exploitée par crainte de problèmes de complexité pour la classification [Bra92]. Du côté de la PPO, les relations entre objets ont longtemps été (mal) prises en compte par des attributs. Lorsqu'il s'agit de "vraies" relations, pour lesquelles le relatif a toujours son corrélatif[1], le paradigme objet paraît mal adapté. En effet, faut-il décrire la relation comme une caractéristique de l'objet domaine, ou comme caractéristique de l'objet du co-domaine, ou dans les deux ? Les deux premières solutions paraissent arbitraires, alors que la troisième introduit des redondances et des problèmes de maintien de cohérence. L'alternative est de décrire les relations hors des classes. Elles réclament alors un statut.

Un argument pour une approche "purement objet", serait de dire que la participation dans une relation est une caractéristique de l'objet. La relation fait donc bien partie de sa description et peut être regroupée avec les autres éléments caractérisant l'objet (i.e. dans sa classe). Ceci est également valable pour la relation inverse (le corrélatif). D'un point de vue de la représentation des connaissances, il semble donc que la description d'une relation a sa place dans les caractérisations de tous les objets reliés par elle. Des problèmes de redondance ou de maintien de cohérence ne sont pas pertinents à ce niveau.

Les relations privilégiées qui existent entre le tout et ses parties, dans le cadre des objets composés, ont donné lieu a de nombreux travaux [Bla&87][Car89][Esc&90][Mar91][Nap92]. Un problème délicat est celui du partage supporté par ces relations.

[Loo&87] et [Rum87] soulèvent le problème des relations en général, qu'ils classifient en

1. Une des catégories de l'Etre, la relation (ou "le relatif"), est étudiée par Aristote dans son traité des Catégories. Parmi les propriétés des relatifs, il identifie celle-ci : "Tous les relatifs ont leur corrélatifs : par exemple, l'esclave est dit esclave du maître, et le maître, maître de l'esclave ; ". Il ne s'agit pas là d'une propriété évidente à première vue : "— Cependant il y a des cas où la corrélation semblera ne pas se produire : c'est quand on n'a pas rendu de façon appropriée le terme auquel le relatif est rapporté et qu'on s'est trompé en l'exprimant". (In *Organon I. Catégories - II. De l'Interprétation*, Trad. J. Tricot, Librairie Philosophique J. Vrin, Paris, 1984)

trois types, dont seul le premier est traité couramment en PPO : généralisation, association et agrégation. Ils réclament un statut indépendant pour les relations, et proposent une combinaison du modèle Orienté Objet avec le modèle Entité-Relation, intégrant classes et relations. Ils se rapprochent ainsi du modèle issu des réseaux sémantiques, discuté ci-dessus. Une objection importante (valable aussi pour les logiques de description) est que cette approche globalise les relations : l'espace de définition se trouvant hors des objets, chaque relation (nommée) est identifiée une fois pour toutes, fixant sa sémantique de façon unique. Dans l'approche objet "classique", les caractéristiques (attributs, méthodes) décrites dans une classe sont encapsulées dans celle-ci et indépendantes des caractéristiques décrites dans une classe indépendante. C'est une des qualités de la modularité. La relation qui relie un écolier à son maître n'est pas la même que celle qui relie un chien à son maître. Ces deux relations ne peuvent donc pas être définies comme une seule (de façon globale). Pourtant, elles peuvent très bien avoir le même nom. La relation maître[1] (comme nombre d'autres relations) ne semble pas pouvoir être décrite de façon univoque en dehors de tout contexte. Elle n'a de sens que rapportée aux objets reliés par elle. Les classes de ces objets semblent donc l'endroit approprié pour décrire aussi bien les relations que les attributs et les méthodes et garantir le polymorphisme lié à l'encapsulation[2]. Ce sera la classe qui déterminera le contexte sémantique de la relation. Ceci n'exclue pas la réification des relations (comme propriétés particulières), de façon à pouvoir les décrire (de la même manière que les facettes permettent de caractériser les attributs). Cette solution est adoptée par plusieurs systèmes à tendance RPO, comme TROPES, SHOOD, SMECI, et YAFOOL.

Comme il est parfois difficile de décider si une propriété est un attribut ou une relation, SRL [Wri&83][3] a choisi l'approche "tout attribut est une relation" (implantée comme un slot), ce qui implique que l'information peut être transférée dans les deux sens. Deux objets (schémas) sont en relation dès lors que l'un figure comme valeur dans un slot de l'autre. Une relation inverse est mise en place dans l'objet en valeur d'un slot, lors de l'affectation du slot. De plus, pour tout slot un schéma de même nom est défini, décrivant les informations associées à la relation et précisant sa sémantique par défaut, de façon très élaborée et modifiable par l'utilisateur (domain, range, relation inverse, cardinalité, propriétés mathématiques, héritabilité...). L'association d'un schéma pour chaque relation la rend du même coup globale et supporte de ce fait une définition unique, ce qui nous ramène à la critique énoncée ci-dessus.

Les attributs sont assimilés aux relations également en SHOOD [Esc&90]. Leur définition entraîne également la création de classes, mais les noms de celles-ci sont préfixés par la classe de définition, ce qui permet de partitionner l'espace de définition pour ne pas rendre la définition d'un attribut globale au système. Ainsi l'attribut `age` de la classe `Personne` est décrit par la classe `Personne.age`. SHOOD permet d'exprimer des propriétés mathématiques également : réflexivité, anti-réflexivité, symétrie, anti-symétrie et quelques opérations comme l'union, l'intersection, l'inverse et la composition (sans précision dans [Esc&90]). La notion de dépendance entre les objets reliés est intéressante pour qualifier les relations entre composants et composé dans les objets complexes. SHOOD distingue les dépendances

1. à ne pas confondre avec le *concept* Maître, qui pourrait être représenté par une classe. (Distinguer de la même façon la relation mère et le concept Mère.)
2. Il s'agit d'un polymorphisme "ad hoc" [Car&85], qui existe lorsque une fonction (relation, méthode) est définie sur des objets de types (et de structures) différents.
3. "*In a description of a "person", is having a "head" an attribute or a relation?*"

exclusives (le composant n'est composant que d'un seul composé), partagées (relations plus générales), nil (pas de dépendance), existentielles (le composant ne peut exister indépendamment du composé), spécifiques. Le dernier type de dépendance permet de traiter des exceptions (composant particulier lié à un composé particulier), les autres dépendances étant valables de façon générique et spécifiées au niveau de la classe.

TROPES [Mar91] distingue trois types d'attributs, chacun avec ses facettes pertinentes : les *propriétés*, les *relations* et les *composants*.

AIRELLE comporte des attributs qui sont de simples symboles (comme en SMALLTALK), mais qui peuvent être réifiés à volonté. Réifier un attribut, c'est le définir explicitement comme unité (objet), instance d'une classe d'attributs. Parmi les classes d'attributs, il y a Attributs, avec ses sous-classes Monovalués et Multivalués. La classe Relations, sous-classe de Multivalués, définit les champs *racine* et *inverse*, et plusieurs méthodes permettant entre autres de calculer la fermeture transitive d'une relation et de l'afficher sous forme d'arbre. La facette *inverse* n'est pas exploitée pour des mises à jour automatiques. La réification d'un attribut, comme celle des relations en SRL, rend sa définition unique et globale pour toutes les classes, et interdit tout affinement de cette définition dans les sous-classes[1].

Le langage objet LORE, pensé à la base selon une interprétation purement ensembliste de la notion de classe, introduit les relations pour supporter l'aspect conceptuel qui consiste à lier des propriétés aux objets [Cas85]. Toutes les propriétés (attributs, méthodes) sont représentées par des relations mono- ou multi-valués. La factorisation des propriétés se fait grâce aux ensembles en tant que domaines de relations. Au cours de l'évolution du langage[2], les différents types de propriétés sont organisés dans un graphe de classes [Ben&89]. A part les relations et les fonctions, qui sont des propriétés définies globalement, il y a les attributs et méthodes, qui sont des relations et fonctions pouvant être affinées, grâce à la classe Restriction, classe des propriétés redéfinissables.

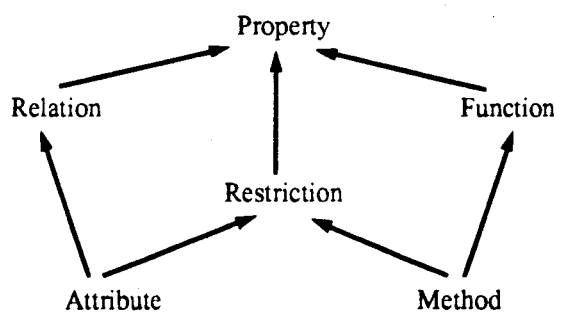


Fig. 5 Propriétés globales et spécialisables en LORE

L'utilisation de facettes de typage (*un*, *des*) ou de la facette *lien-inverse* lors de la définition d'un attribut dans un frame (*modèle*) en YAFOOL, fait de cet attribut une relation [Duc91] [Mas&89]. Lors de la création d'un modèle, on procède à sa *dualisation*, qui consiste à définir un objet pour chaque propriété du modèle (attributs, méthodes). Selon les facettes utilisées, il

1. Le manuel [Ada&88] évoque la possibilité de définir des attributs dans le package (type lisp) d'une classe, pour avoir des définitions locales, mais cette possibilité ne permet pas non plus l'affinement.
2. commercialisé pendant un moment sous le nom de SPOKE, et ressurgi récemment sous le nom de LAURE aux Etats-Unis [Cas91b].

est possible d'identifier les méthodes, les attributs mono- ou multivalués (resp. *attribut1* et *attribut2*, cf. Fig. 6), et les relations, qui sont des attributs à valeur dans l'ensemble des objets. Ces objets forment le dual.

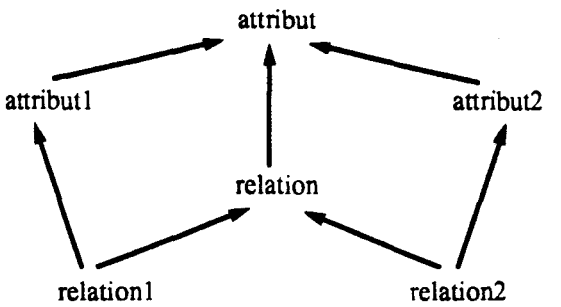


Fig. 6 Partie de la hiérarchie du "dual" de YAFOOL

La définition des propriétés reportée au niveau global par la dualisation n'implique pas ici l'impossibilité de leur affinement. En effet, chaque propriété dualisée contient la liste des frames qui la définissent, et le look-up se charge de retrouver la définition ou la valeur pertinente, qui se trouve dans les frames eux-mêmes. La cohérence des relations est assurée par des réflexes SI-AJOUT et SI-ENLEVE installés automatiquement, à condition d'avoir indiqué correctement les liens inverses lors de leur définition. YAFOOL définit des fonctions de fermeture transitive pour les relations.

OBJLOG+ définit une facette (*aspect*) *inverse*, multivaluée, utilisable pour tout champ dont le domaine est de type objet. La mise à jour est effectuée automatiquement lors d'une affectation.

```
PERSONNE
 <état-civil,nil>
          domaine: {CHAINE}
          cardinalité: <1,1>
 <père,nil>
          domaine: {PERSONNE}
          cardinalité: <1,1>
          inverse: {<PERSONNE,enfants,nil>}
 <mère,nil>
          domaine: {PERSONNE}
          cardinalité: <1,1>
          inverse: {<PERSONNE,enfants,nil>}
 <enfants,nil>
          domaine: {PERSONNE}
          inverse: {<PERSONNE,père,nil>, <PERSONNE,mère,nil>}
```

Fig. 7 Facette inverse en OBJLOG+ [Fau91]

Sur l'exemple de la Fig. 7, la valuation du champ `père` d'une `Personne` Toto avec la valeur `Jean` (instance de `Personne`) aura pour effet l'affectation de `Toto` au champ `enfants` de `Jean`. On peut constater ici l'utilisation problématique d'un lien inverse *multiple* : quelle mise à jour effectuer suite à l'affectation du champ `enfants` d'une personne ? Le choix du corrélatif de

façon symétrique se montre essentiel pour la cohérence (enfants-parent paraît un meilleur choix ici).

En SMECI [ILO91], un attribut (*champ*) de type objet (facette *type* = *object* ou *lobject*) possède des facettes complémentaires de type relation, comme *domain* (le co-domaine), *inverse*, *import* et *link-semantics*. Comme en YAFOOL, si la facette inverse est renseignée, les propagations sont assurées automatiquement, dans un sens, ou dans les deux, si les inverses sont indiqués des deux côtés. La facette *import* indique les champs dont on veut importer ("hériter") la valeur suivant le lien en question. En spécifiant une (ou plusieurs) valeur(s) pour la facette link-semantics, on définit autant d'étiquettes pour un champ, représentant un type de relation. La même étiquette attachée à plusieurs champs permet de regrouper logiquement des objets de types différents, mais qui entretiennent la même relation avec l'objet définissant le champ, par exemple ses composants, tous les membres de la famille, ... Plutôt que d'un éclatement d'un champ en plusieurs sous-champs, comme en KL-ONE (modificateur *DIFFS*) et en TROPES, il s'agit d'un regroupement (logique) de champs constituant une même relation. Un champ peut, de plus, participer à plusieurs relations, par l'attachement de plusieurs étiquettes.

L'expression des relations peut servir un simple objectif de représentation, comme dans la spécification de la "sémantique" du lien (SMECI) ou des informations véhiculées par elles (héritage sélectif). Elle peut aussi remplir le rôle d'expression de contraintes d'intégrité et permettre au système d'effectuer certaines inférences par propagation de ces contraintes le long des liens. La notion de relation comme lien de dépendance entre objets (de type maître-esclave) a été développée dans le système Othelo [For&89]. Pour chaque lien défini entre un objet influant (maître) et un objet dépendant (esclave), on peut préciser les opérations sur l'objet influant qui nécessitent une mise à jour de l'objet dépendant, ainsi que l'action à effectuer pour rétablir la cohérence.

Etre Composé-de ou être Partie-de

Un type de relation déjà mentionné ci-dessus est celle qui relie un objet composé à ses parties. Elle entraîne des problèmes de dépendance (cf. différents types de dépendance en SHOOD), des problèmes d'encapsulation et d'accès aux parties [Bla&87], des problèmes de partage d'information (le fameux problème de la couleur de la voiture et de celle de sa carrosserie, partagée et de ce fait propagée ("héritée") dans un sens ou dans l'autre), et des problèmes de cohérence à l'intérieur de l'objet composé (relations entre les composants).

Un objet composé peut être vu comme l'agrégation de ses parties, qui sont des objets, éventuellement composés (décomposables) à leur tour, et récursivement. Cette décomposition induit une hiérarchie des parties, qui a été remarquée comme relation (transitive) particulière depuis les réseaux sémantiques. Si les langages de PPO n'offrent aucun moyen d'expression adapté aux objets composés (comme pour les relations en général), les langages à tendance RPO prévoient parfois des classes dédiées à leur représentation, comme LOOPS' CompositeObject [Ste&85] et objet-composite de YAFOOL. Ces classes permettent une description des parties, et la création (resp. la destruction) d'une instance composée entraîne la création (resp. la destruction) d'objets pour toutes ses parties.

En YAFOOL, les composants sont définis dans la propriété *compose-de* qui contient leurs types et cardinalités, la propriété *lien-de-composition* contenant leurs noms (dans le même

ordre). La Fig. 8 décrit les cycles comme des véhicules composés de deux objets de type roue, d'un cadre et d'un guidon, accessibles comme des champs de noms (resp.) mes-roues, mon-chassis et ma-gouv.

```
(defmodele cycle
           (est-un vehicule)
           (compose-de (2 roue) cadre guidon)
           (lien-de-composition mes-roues mon-chassis ma-gouv))
```

Fig. 8 Définition d'objet composite en YAFOOL [Duc91]

Les frames roue, cadre et guidon sont supposés être définis par ailleurs et peuvent eux-mêmes être des objets composés.

OBJLOG réserve la facette *statut* (valeur : *structurel*) à l'expression des relations structurelles, c.-à-d. des relations qui représentent des agrégations d'objets pour former des objets complexes. Il s'agit de relations telles que *partie-de*, mais n'importe quelle relation (champ avec domaine de type objet) peut être déclarée structurelle. [Dug88] développe pour les relations structurelles un mécanisme puissant de partage de champs, appelé héritage sélectif. La facette *heritage* sert à spécifier un filtre d'héritage qui indique tous les champs à partager entre les objets en relation structurelle. L'héritage sélectif opère une "unification sémantique" de ces champs, fusionnant toutes les facettes les décrivant. Il s'agit donc d'un réel partage de champ, et non d'une simple importation (*import* de SMECI). L'héritage sélectif étant symétrique et transitif, l'unification sémantique peut faire intervenir plusieurs objets, partageant tous le même champ, qui peut être affecté par chacun d'entre eux. Il n'y a donc pas de notion de dépendance de type maître-esclave (Othelo).

THINGLAB [Bor79] est un des premiers systèmes à juger que la hiérarchie de décomposition est aussi indispensable pour la modélisation des systèmes complexes que la hiérarchie d'héritage. Plutôt que d'offrir une classe particulière d'objets composites, *chaque* classe contient alors un ensemble de descriptions de parties. L'expression des contraintes (cf. SDs de KL-ONE) entre ces parties est déclarative, et transformée par le système en méthodes, qui sont ensuite invoquées pour maintenir la cohérence. La notion de contrainte occupe une place primordiale dans le système et a inspiré de nombreux travaux sur la programmation par contraintes.

Nous mentionnons le traitement d'objets composites dans le cadre de la classification multi-points de vue en TROPES [Mar91], où les objets sont décomposés différemment selon des points de vue différents et où la classification d'instances se traduit récursivement comme la classification des parties.

Un travail original sur l'exploitation de hiérarchies de parties en combinaison avec une représentation multi-points de vue par agrégation est celui décrit dans [Wol90].

Citons également les travaux sur les objets composites dans le domaine des Bases de Données Orientés Objet, par exemple [Kim&87][Kim&89].

Enfin, d'un point de vue cognitif et linguistique, citons l'étude approfondie des différents types de relations réunies sous la désignation "partie-de" et utilisés couramment dans le langage ordinaire (anglais) par [Win&87] (cf. aussi [Nap92] et [Mot93]). Les auteurs identifient 6 sémantiques différentes et étudient leurs propriétés.

2.1.2 Comportement

L'approche PPO se fait fort d'intégrer au sein d'un même module les deux aspects intervenant dans tout programme informatique : données et procédures. Si nombre de langages objets regroupent attributs et méthodes, définissant structure et comportement de l'objet, certains se posent le problème (analogue à celui des relations symétriques) des méthodes faisant intervenir *plusieurs* objets, sans que l'on puisse clairement les identifier comme le comportement de l'un d'entre eux. Une solution à ce problème est proposée sous la forme de fonctions génériques et multi-méthodes.

L'approche RPO privilégie beaucoup moins cet aspect de comportement faisant partie intégrante de l'objet structuré. Les premiers langages de frames fournissaient un large éventail de fonctions de manipulation des structures riches mais passives qu'étaient les frames. Certaines procédures à effectuer dans des cas spécifiques pouvaient néanmoins être stockées dans les structures elles-mêmes — selon l'idée même de Minsky [Min75] —, et ont introduit une forme de comportement spécifique aux langages de frames : celui des attachements procéduraux.

Plusieurs langages hybrides adoptent à la fois le comportement offert par les méthodes, pour une programmation de type PPO et celui des procédures attachées aux attributs et déclenchées lors des accès à ceux-ci — le "comportement des attributs" — pour une programmation dirigée par les accès [Ste&86].

A l'opposé, les systèmes de la famille KL-ONE ignorent totalement les deux types de comportement. Même si la "Structural/Condition" de [Bra77] était à l'origine une procédure à appliquer aux arguments (DATTRS) (cf. 2.1.1.3.) pour déterminer l'applicabilité du concept auquel elle était attachée et si KL-ONE prévoit une forme d'attachement procédural (appelé "interpretative hooks" ou *ihooks*) [Bra&85, p. 199], l'implantation en est avouée être très simple et aucun des systèmes ultérieurs ne l'a reprise[1]. Ce sont tous des systèmes purement descriptifs, dont le seul aspect procédural est incarné dans le classificateur.

2.1.3 Accès à la structure

Par accès à la structure, nous entendons l'accès aux informations contenues dans la structure, c.-à-d. la lecture et l'écriture des valeurs d'attributs, et non pas l'accès à la structure elle-même, qui relève du niveau méta, s'il existe (cf. 2.5).

Dans la majorité des langages de PPO [Gol&83], l'encapsulation des attributs est une notion primordiale qui garantit la modularité et la stricte séparation entre spécification (disponible par l'interface) et implantation des objets. L'accès aux attributs n'est possible qu'au travers de méthodes définies expressément. L'exigence d'encapsulation est quasiment inexistante en RPO, où il est important d'avoir accès aux connaissances représentées, synonyme de souplesse dans la consultation comme dans la modification des informations.

La majorité des langages de frames et langages hybrides emploient des primitives d'accès aux attributs [Ste&86] [Rob&77b][Bob&83][Wri&83][Rec&90]. Il en existe souvent des versions qui déclenchent les procédures attachées et des versions sans exécution de ces

1. Notons que, pour augmenter le confort d'utilisation du système, les versions récentes du système CLASSIC proposent des règles de calcul de valeurs, au travers desquelles on peut spécifier une fonction lisp [Res&93].

procédures. Les langages de RPO pratiquant l'envoi de message avec méthodes d'accès aux attributs proposent des options lors de la définition de slots pour la génération automatique de méthodes de lecture et/ou écriture, avec exécution des réflexes intégrée [Ada&88] [ILO91][Kee89].

2.2 Instanciation ou dérivation

2.2.1 Classe/Instance/Héritage

On trouve la distinction classe/instance surtout en PPO. La classe est une abstraction de ses instances et décrit leur structure et comportement. Elle sert de "moule" à la création des instances, qui sont construites exactement selon la structure de la classe. Une classe est sous-classe d'une ou plusieurs autres classes (ses super-classes). L'ensemble de ces classes est organisé en hiérarchie (arbre ou graphe) induite par la relation sous-classe. Cette hiérarchie est le support du mécanisme d'héritage entre classes. L'héritage entre classes permet le partage et la factorisation d'attributs et de méthodes. Une instance possède tous les attributs et méthodes définis et hérités par sa classe.

L'héritage est loin d'être une notion simple. Elle est interprétée de nombreuses manières. Dans l'approche PPO, l'héritage est vu avant tout comme un mécanisme de partage de code, que de nombreux auteurs associent au sous-typage. Pour des raisons d'implantation, la hiérarchie de sous-typage ne correspond pas toujours à la hiérarchie conceptuelle [Hal&87]. D'autres considèrent au contraire qu'il faut distinguer héritage et sous-typage [Sny87] [Ame87] [Sny91]. Selon cette vue, l'héritage opère au niveau "code", c.-à-d. au niveau de l'implantation. Le sous-typage concerne le niveau spécification : un type spécifie d'abord un comportement [Ame87][Ame90]. L'implantation des spécifications peut faire appel à l'héritage, mais les deux relations sont vues comme orthogonales. Sakkinen [Sak89] préfère parler d'héritage essentiel pour l'héritage de spécification et d'héritage accidentel pour l'héritage d'implantation.

Même si les premiers langages de RPO sont plutôt fondés sur la théorie des prototypes (2.2.2), il en existe aussi à base de classes/instances. Un exemple type est le langage de schémas Shirka [Rec&90]. L'approche RPO a plutôt tendance à superposer héritage et spécialisation conceptuelle. Le graphe des classes *représente* un ensemble de concepts en relation de spécialisation. Une classe qui hérite d'une autre classe est plus spécifique que celle-ci, et inversement si une classe en spécialise une autre, alors elle hérite toutes les caractéristiques décrites par elle. La hiérarchie d'héritage est une hiérarchie conceptuelle et les caractéristiques décrites dans les classes ont un caractère conceptuel. Cependant, même en représentation des connaissances, l'héritage qui se conforme à la hiérarchie conceptuelle ne fait pas l'unanimité. D'une part, l'héritage est souvent considéré comme trop systématique. Les partisans d'une approche plus souple proposent différentes façons de prendre en compte des "exceptions à l'héritage". Une classe peut alors être sous-classe d'une autre sans en hériter toutes les caractéristiques [Duc&89]. D'autre part, quelques auteurs séparent explicitement le niveau conceptuel du niveau d'implantation. [Nic91] manipule une hiérarchie de classes ("types abstraits") avec héritage d'implantation (structure et méthodes) parallèlement à une hiérarchie de schémas qui héritent des propriétés (conceptuelles) communes. [Fer89] distingue pour MERING IV une hiérarchie de classes conceptuelles et des entités définissant la structure et le comportement des instances (niveau implantation). La plupart des langages hybrides à base de classes et instances (orientés RPO mais avec des fonctionnalités de PPO) combinent la hiérarchie conceptuelle avec un mécanisme d'héritage [Bob&83][Alb88][Ben&89][ILO91][1].

1. SMECI associe à chaque classe un "prototype", contenant les valeurs communes des instances, valeurs par défaut etc.

Remarquons que la distinction des niveaux n'est pas la même pour tous les auteurs. Les points de vue PPO et RPO ne mettent pas les accents aux mêmes endroits.

Retenons que quelle que soit l'approche, l'héritage est avant tout un mécanisme de partage, que ce soit le partage de code (PPO) ou de connaissance (RPO). Ce mécanisme a comme support une relation d'ordre partiel, qui peut être la relation de sous-typage, la relation ensembliste associée à la spécialisation de concepts, ou une relation ad-hoc inspirée par des considérations de réutilisation d'implantation.

Dans l'approche classe/instance, l'héritage permet d'inférer toute la structure des instances, en général au moment de l'instanciation, et leur comportement, de manière dynamique lors de tout envoi de message. Cette inférence n'est qu'une inférence faible pour la RPO [Pat91] [Nap92]. D'autres formes d'inférence doivent la compléter (filtrage, subsomption, classification).

2.2.2 Prototype/Dérivation/Héritage

L'approche par prototype, comme celle fondée sur la dualité classe/instance, est présente aussi bien en PPO qu'en RPO. En PPO, elle est à la base des langages d'acteurs [Lie86a/b] et des langages de prototypes [LaL&86] [Bor86] [Ung&87] [Don&92]. En RPO, elle a donné naissance à toute la lignée des langages de frames [Rob&77a/b] [Bob&77] [Mar88] et reste à la base d'un certain nombre de langages hybrides [Duc91][Fik&85].

La différence fondamentale entre les deux approches est la distinction instanciation/ dérivation. La création de nouveaux objets par instanciation en fait des structures exactement conformes à leur classe, qui ne diffèrent entre elles que par les valeurs de leurs attributs. Les objets ne possèdent pas d'attributs (ni de méthodes) en propre. Cette approche convient assez bien en PPO, mais est jugée trop rigide en RPO. Les langages de frames proposent une création d'objets par *dérivation* à partir d'objets déjà existants, considérés comme parents. Le lien parent autorise un mécanisme d'héritage, qui est à la fois un héritage de structure et de valeurs. Si un objet ne possède pas un attribut en propre, celui-ci est recherché de proche en proche sur les parents selon un algorithme d'héritage (héritage dynamique comme pour les méthodes dans les systèmes à base de classes/instances). Les valeurs d'attributs (ou les moyens de les calculer) sont recherchées selon le même algorithme. La dérivation exprime une relation de spécialisation, qui est réalisée par masquage et extension de caractéristiques. Cette spécialisation est une spécialisation à sémantique faible, car elle n'obéit pas forcément à des règles d'affinement (sous-typage). Dans les langages de frames, il n'y a pas de distinction entre objets génériques et objets spécifiques comme dans le modèle classe/instance. Tout objet est la spécialisation d'un (ou de plusieurs) autres objets et peut être spécialisé à son tour. Certains langages hybrides introduisent une distinction (superficielle) entre objets génériques "classes" et objets individuels "instances" [Duc91]. Cependant, cette distinction est très différente de celle qui existe entre classes et instances du modèle classe/instance, qui traduit une relation d'instanciation par moulage. Dans les langages de frames, la seule relation entre objets est celle de spécialisation (*isa*)[1]. Certains langages hybrides distinguent le lien *sorte-de* et *est-un* et se rapprochent ainsi du modèle classe/instance [Fik&85].

Les langages terminologiques s'inscrivent également dans l'approche de spécialisation par

1. Ce seul lien *isa* est alors (sur)chargé de sens, cf. [Bra83].

dérivation. Chaque concept est décrit en fonction de concepts existants, dont il hérite de ce fait. Ces langages respectent cependant une sémantique rigoureuse d'affinement (grâce à la subsomption). La distinction entre concepts génériques et concepts individuels n'est que superficielle.

Les langages d'acteurs et les langages de programmation à base de prototypes[1] procèdent plutôt par *copie* que par dérivation pour créer de nouveaux objets. Là où les langages d'acteurs mettent en œuvre un mécanisme de *délégation* des objets vers leurs parents (*proxies*) [Lie86a][Ste87][Ste&88], les langages de prototypes comme SELF combinent copie et héritage [Ung&87]. Une fois copié, le nouvel objet n'a en général plus de lien vers son prototype générateur. Par contre, il possède un ou plusieurs slots parents, exploités par le mécanisme d'héritage (dynamique). Dans les deux cas, les parents ne sont pas fixés une fois pour toutes et peuvent être modifiés dynamiquement ; cela entraîne une modification de l'héritage. La copie peut être modifiée et étendue et servir à son tour de prototype pour d'autres copies.

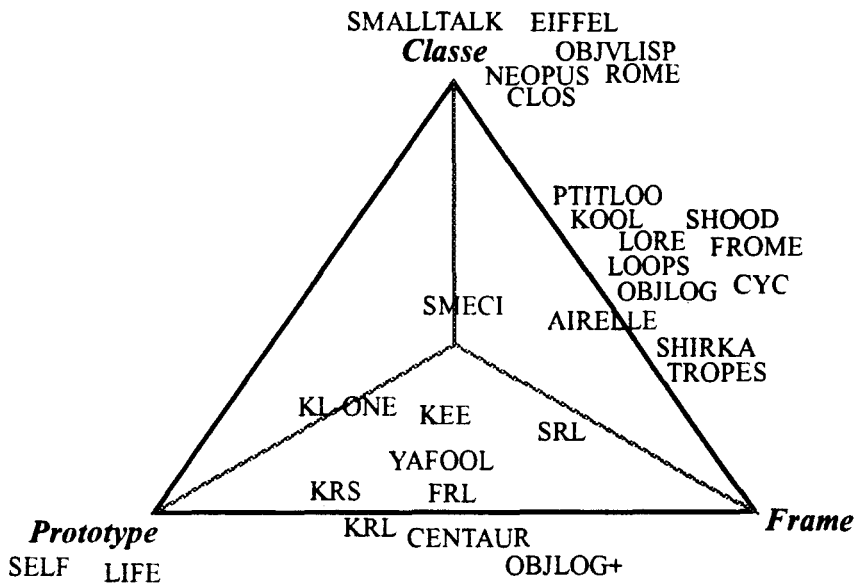


Fig. 9 Langages à base de classes, prototypes, et/ou frames

1. On parle maintenant de langages de programmation "sans classes" [Ung&91].

2.3 Représentations multiples

2.3.1 Position du problème

> *"The crucial problem is that the world is a continuum and concepts are discrete. For any specific purpose, a discrete model can form a workable approximation to a continuum, but it is always an approximation that must leave out features that may be essential for other purposes."*
>
> J.F. Sowa[1]

2.3.1.1 Points de vue multiples

La classification est une façon d'organiser les connaissances couramment utilisée aussi bien dans la vie de tous les jours que dans les disciplines scientifiques (biologie, mathématiques, programmation...) [Weg86]. La description d'un domaine sous la forme d'une hiérarchie de classes est rarement unique. En effet, la façon de hiérarchiser le domaine et les critères de subdivision d'une classe en sous-classes plus spécifiques dépendent fortement de la personne qui opère cette description, de son domaine de compétence et surtout de l'objectif de la classification, autrement dit *du point de vue* que l'on porte sur les objets du domaine. Partout où il est question de classification (taxonomie) dans la littérature scientifique, la multiplicité est évoquée. Par exemple, dans l'Introduction de la Classification Internationale des Maladies, [OMS68] nous lisons :

> "La classification est fondamentale pour l'étude quantitative de tout phénomène. Elle est reconnue comme la base de toute généralisation scientifique ; c'est pourquoi elle représente un élément essentiel au point de vue de la méthodologie statistique. Des définitions uniformes et des systèmes uniformes de classement sont une condition préalable au progrès de la connaissance scientifique.[...]"
>
> "Il existe de nombreuses façons d'envisager le classement des maladies. L'anatomiste, par exemple, peut désirer un classement d'après la partie du corps qui est atteinte. En revanche, le pathologiste est surtout intéressé par l'évolution de la maladie. Le clinicien, qui doit considérer la maladie sous ces deux angles, a besoin d'une connaissance plus approfondie de l'étiologie. En d'autres termes, les systèmes de classement sont nombreux, et celui qui sera choisi dans chaque cas particulier sera déterminé par les préoccupations du chercheur."

Les différents experts (anatomiste, pathologiste) se réfèrent chacun à la classification qui correspond à leur propre domaine de compétence. Il apparaît néanmoins que dans un domaine de compétence donné, on peut être amené à considérer plusieurs points de vue en même temps (exemple du clinicien). L'expertise consiste alors à savoir prendre en compte les différents points de vue et à les mettre en relation.

La définition d'une classification unique comme base commune nécessaire à la communauté scientifique a demandé des efforts considérables dans les divers domaines et implique de nombreux compromis. Elle revient souvent à privilégier la classification établie

1. [Sow84] p. 345.

selon un des points de vue en y apportant des aménagements. La classification internationale des maladies publiée par l'OMS repose à la base sur le point de vue anatomique.

Si les classifications à vocation scientifique sont souvent extrêmement complexes et élaborées, le profane est amené à classifier ses connaissances de manière plus naïve. La classification ne sera pas unique pour autant.

Minsky, dans sa note intitulée "Niveaux et classifications" ([Min88] 8.10) montre deux classifications possibles des objets physiques (cf. Fig. 10). Il n'y a pas une seule hiérarchie correcte ; tout dépend de ce que l'on veut en faire. Ainsi, "avion" et "oiseau" peuvent se trouver éloignés dans une classification et proches dans une autre.

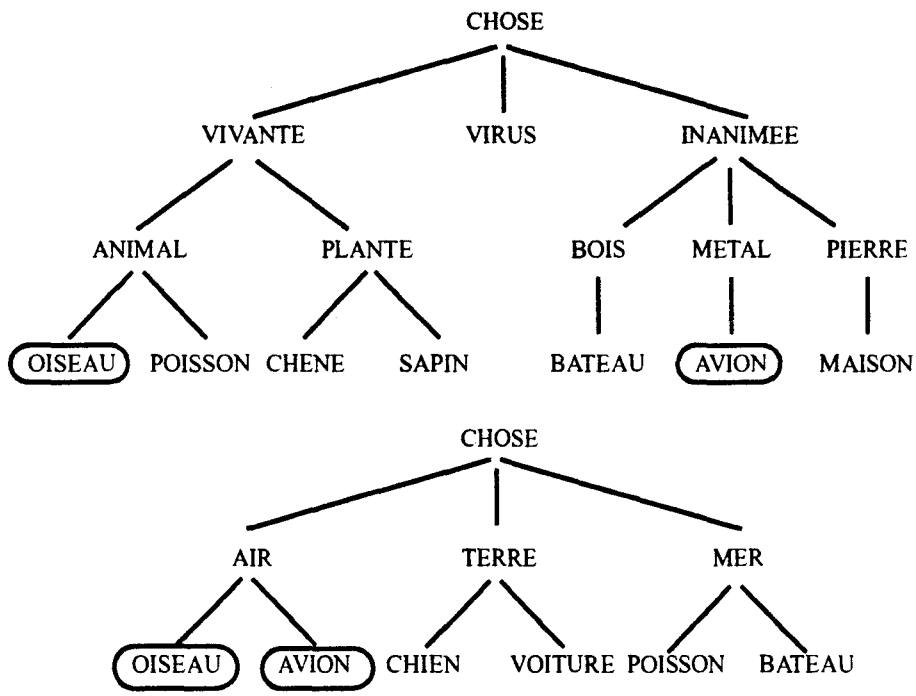


Fig. 10 Deux classifications d'objets physiques

Pour montrer que nous utilisons (souvent) plusieurs classifications à la fois, Minsky prend l'exemple du canard en porcelaine, qui peut à la fois servir d'oiseau (dans le jeu d'un enfant) et être traité comme un objet fragile.

Sowa [Sow84] rapporte un exemple de conceptualisation multiple décrit par Kierkegaard[1]. Il s'agit de la description d'une forêt selon les approches de différentes personnes. L'approche écologique verra la forêt comme un système complexe de flore et de faune interdépendantes. L'approche esthétique mettra en avant la diversité des odeurs, des sons et des vues. L'approche industrielle y voit une source de matière première pour la fabrication de papier, alors qu'une vue mystique de la forêt la considère comme une manifestation de l'harmonie de l'univers. Aucune de ces vues ne détient la vérité absolue, chacune exprimant un aspect de la réalité. Pour Kierkegaard, chaque aspect invalide (exclut) les autres. En poussant la relativité des conceptualisations jusqu'au bout, il en arrive à la constatation plutôt pessimiste que chaque vue peut avoir une cohérence interne, mais que les vues (et les systèmes conceptuels) de deux

1. Kierkegaard, Sören (1843) *Either/Or*, two vols., Anchor Books, Garden City, NY.

individus peuvent être incompatibles. Il n'y a alors plus de communication possible.

Entre les deux positions extrêmes de la vue unique d'un côté et de l'incompatibilité totale des vues de l'autre, il semble y avoir suffisamment de place pour l'élaboration constructive de conceptualisations multiples. Plutôt que d'avoir des vues qui s'excluent, on cherchera à les mettre en rapport, afin de disposer de plusieurs façons de comprendre un domaine, ou de résoudre certains problèmes qui s'y posent.

La multiplicité des descriptions semble donc résulter naturellement de la multiplicité des regards portés sur les objets du domaine.

2.3.1.2 Décomposition d'un domaine complexe

D'autre part, la multiplicité des descriptions peut provenir d'un souci de décomposition afin de simplifier un domaine complexe, comportant de nombreux aspects. L'analyse du domaine mène à un découpage en descriptions partielles selon des aspects différents, dans le but de mieux le décrire et de mieux le comprendre.

Dans le domaine de l'apprentissage de la programmation, par exemple, le système EXPLAINER [Rat&93], décrit des programmes informatiques selon différents aspects (texte source, représentation graphique, exemples d'exécution, documentation, ...) afin d'apprendre à l'utilisateur la programmation par l'exemple. Chaque aspect représenté doit contribuer à la compréhension du programme étudié.

Dans le contexte d'un environnement de développement de logiciels, Shilling et Sweeney [Shi&89] suggèrent des vues multiples comme des abstractions destinées à simplifier la structure complexe d'un système ; chaque vue fournit une interface adaptée à un utilisateur (et/ou développeur) particulier du système.

Les points de vue décrits dans [Mar&93] sont également destinés à extraire des sous-ensembles d'informations pertinentes pour des utilisateurs différents d'un modèle complexe commun[1].

2.3.1.3 La prise en compte de la multi-description

Dans les systèmes à objets, différentes solutions ont été imaginées pour pouvoir prendre en compte les descriptions multiples des objets représentés. Dans les paragraphes suivants, nous passons en revue l'héritage multiple, l'agrégation de perspectives, l'approche par coréférence et enfin la représentation multiple d'objets, qui ont tous plus ou moins bien traité le problème. Ensuite nous montrons les différentes façons possibles d'accéder à une description selon un point de vue.

2.3.2 Prise en compte de descriptions multiples

2.3.2.1 L'héritage multiple

L'héritage multiple offre une technique d'intégration de descriptions multiples par

1. Les modèles visés appartiennent au domaine spatial, qui sont si complexes que les auteurs ont renoncé à les exposer dans l'article. L'application de la notion de point de vue dans le domaine spatial est étudiée dans [Car92].

combinaison. Les hiérarchies induites par des classifications sous des points de vue différents sont combinées pour en constituer une seule. Le produit croisé des classes issues des différentes hiérarchies se traduit en de nouvelles classes, décrivant les objets considérés selon plusieurs points de vue. L'exemple de Minsky de la Fig. 10 peut être résumé en une hiérarchie

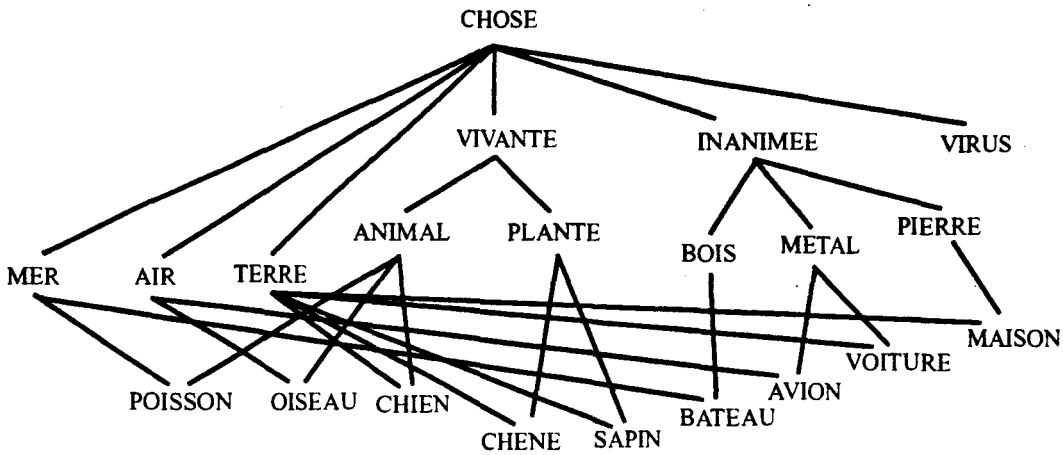


Fig. 11 Combinaison de descriptions par héritage multiple

combinée, comme le montre la Fig. 11. Les points de vue ne sont pas identifiés en tant que tels dans ce graphe et la combinaison n'est effectuée qu'au niveau des feuilles. Une véritable intégration de deux classifications consisterait en une combinaison hiérarchisée de toutes les classes (cf. Fig. 12). On imagine facilement que cette approche mène à une explosion combinatoire du nombre de classes "produits" dès que le nombre de points de vue représentés dans le même graphe augmente.

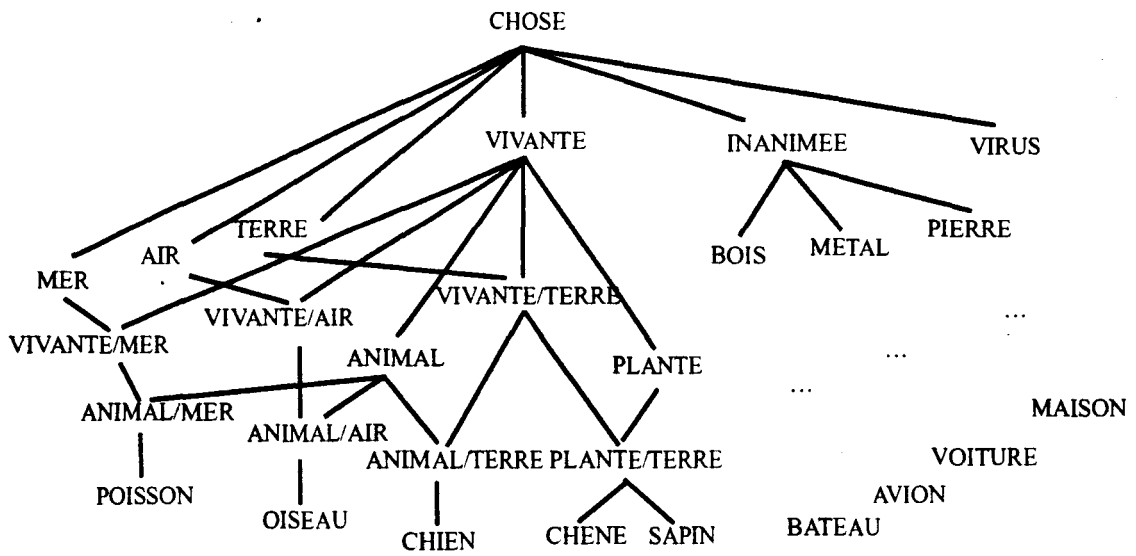


Fig. 12 Combinaison hiérarchisée de classifications d'objets

L'exemple de la Fig. 13 [Car&90a] présente une approche plus modulaire, qui sépare les différents points de vue et regroupe les classes issues d'un même point de vue dans des sous-hiérarchies. C'est une approche qui procède par spécialisation par point de vue : chaque sous-classe de la classe SoftwareCompanyProduct spécialise celle-ci selon un point de vue. Ces sous-classes représentent alors les produits logiciels décrits du point de vue du domaine

de l'application, les produits logiciels décrits du point de vue de la maintenance etc., chaque point de vue induisant la définition de caractéristiques propres décrites dans les différentes classes (non montrées ici). La combinaison des sous-hiérarchies des différents points de vue par héritage multiple, afin de pouvoir représenter des objets particuliers selon plusieurs points de vue, complique ensuite ce graphe modulaire par l'ajout des classes produits.

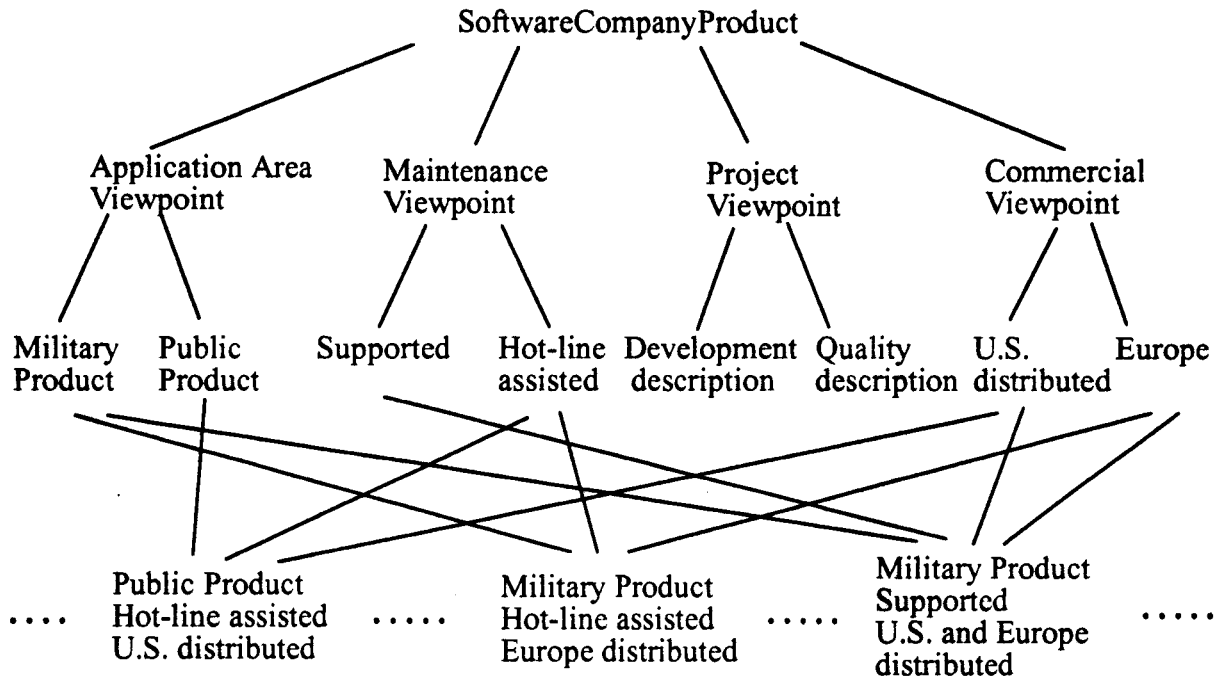


Fig. 13 Description multi-point de vue des produits logiciels

2.3.2.2 L'agrégation de perspectives

Le premier système à intégrer des perspectives multiples pour prendre en compte des descriptions partielles multi-point de vue est KRL [Bob&77] :

> "A description must be able to represent partial knowledge about an entity and accommodate multiple descriptors which can describe the associated entity from different viewpoints."

En KRL, on décrit des classes d'objets par des unités (*UNIT*). On distingue plusieurs catégories d'unités, dont *Basic*, *Abstract* et *Specialization*. La catégorie Basic sert à partitionner les concepts du monde. Un objet particulier ne peut appartenir à deux unités Basic. Abstract est la catégorie des unités abstraites, qui factorisent des propriétés à hériter par leurs spécialisations. Les unités de la catégorie Specialization sont des spécialisations d'unités Basic ou Abstract. La catégorie *Individual* est destinée à la description des objets spécifiques, qui sont construits comme des agrégats de perspectives, où chaque perspective exprime l'appartenance à une unité (Basic ou Specialization).

Si les unités Voyageur et Client sont définies comme des spécialisations de l'unité Personne, l'individu Laurent peut être décrit selon les trois perspectives correspondantes : en tant que personne, voyageur et client (cf. Fig. 14).

```
[Laurent UNIT Individual
 <SELF { (a Personne with
           prénom = "Laurent"
           nom = {(a Nom)
                   (a String with firstCharacter = "F")}
           age = (which IsgreaterThan 21))
         (a Voyageur with
           age = Adulte
         (a Client with
           credit = (a CarteDeCrédit with
                      banque = CréditLyonnais
                      numéro = "4567 Z"))}>]
```

Fig. 14 Description d'une entité individuelle comme un ensemble de perspectives en KRL

Les différentes perspectives sont parfaitement indépendantes et ne partagent aucune information. Chaque perspective regroupe les attributs propres à son espace de description.

L'approche des perspectives multiples de KRL a été reprise dans PIE [Gol&80] et LOOPS [Bob&83]. Dans PIE, environnement de développement conçu pour Smalltalk-76, l'utilisation de perspectives multiples est motivée par la possibilité qu'elles offrent d'hériter du comportement défini par des classes différentes. C'est une façon de pallier l'absence d'héritage multiple. Les objets sont instance de la classe *Node* qui définit une variable d'instance *perspectives*, contenant la liste des perspectives. Une perspective est instance d'une sous-classe de *Perspective* et a un lien vers son node. Chaque perspective possède son état local. La cohérence entre perspectives est gérée par l'instance de Node, qui peut aussi contenir un état local, accessible aux perspectives.

Dans le contexte de LOOPS, qui possède à la fois l'héritage multiple et les perspectives multiples, [Ste&85] définit la notion de perspective comme :

"...a form of composite object interpreted as different views on the same conceptual entity".

On y retrouve les classes Node et Perspective, le lien inverse *perspectiveNode* de la perspective vers son node, et des méthodes de manipulation de perspectives. L'exemple de la Fig. 14 peut être traduit en LOOPS comme ci-dessous (Fig. 15).

Si l'approche par agrégation a été retenu dans différents systèmes du début des années 80, elle est rarement décrite en détail[1]. Il n'est jamais question de la hiérarchisation (possible, souhaitable ?) des classes décrivant les perspectives, et d'éventuels partages de propriétés qui pourraient en résulter. Dans l'exemple (Fig. 15), nous aurions aussi bien pu décrire la classe Personne comme une sous-classe de Perspective, soit superclasse des classes Voyageur et Client, soit classe sœur de celles-ci. Aucun article ne donne des indications précises concernant l'utilisation de ces notions.

1. Le manuel de LOOPS y consacre à peine une page...

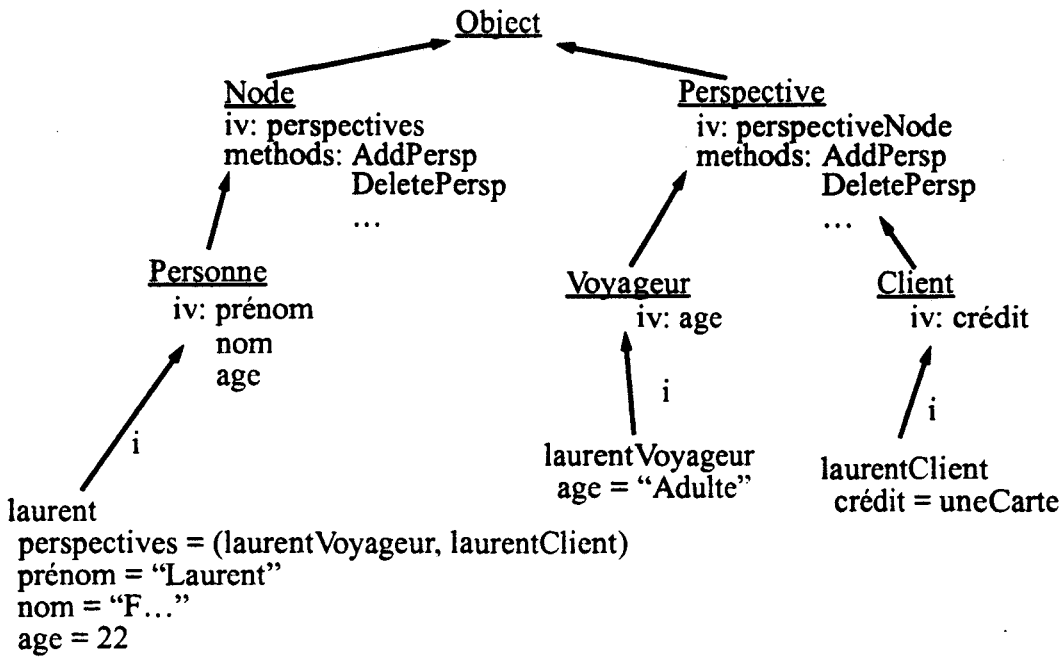


Fig. 15 Une instance de Node composée d'instances de perspectives

Récemment, la même approche a été appliquée en robotique dans le système de conception et simulation de robots SYSTALK [Wol&91]. Un (composant de) robot est décomposé en un ensemble de perspectives appelées *facettes*, qui représentent les "activités" de l'objet. Une hiérarchie des facettes pertinentes dans le domaine de la robotique est définie comme un arbre de classes. Un composant "possède" un ensemble de facettes, telles que Structure, Fonction, qui peuvent encore se décomposer en d'autres facettes, comme Effort mécanique et Forme tridimensionnelle. Selon Wolinski, un objet est composé de facettes ; il n'"est" pas chacune de ses facettes. Les perspectives sont donc explicitement considérées comme des parties de l'objet, ce qui peut prêter à confusion dans un système où la relation de décomposition (du robot en composants et sous-composants) est traitée également. Rien n'empêche de voir la facette Structure comme une classe décrivant tel type de composant "du point de vue de la structure". On se ramène alors à une relation d'appartenance (entre un composant particulier et la classe qui le décrit selon le point de vue de la structure), implantée par agrégation, comme en PIE et LOOPS.

Notons que THINGLAB [Bor79] a également recours à l'agrégation pour implanter l'héritage multiple[1] : les super-classes sont intégrées comme autant de composantes (perspectives) dans la description d'une sous-classe.

2.3.2.3 La coréférence

L'idée qui sous-tend la représentation multiple par coréférence est celle de plusieurs descriptions référençant le même objet. Woods [Woo75] reprend la distinction entre intension

1. Voir [Car89] pour une comparaison des notions d'agrégation et d'héritage. Un inconvénient majeur de l'approche par agrégation est l'absence de l'inférence de propriétés, qui doit être programmée explicitement. Cette inférence est obtenue "gratuitement" par héritage.

et extension, étudiée en philosophie (Frege, Quine), pour montrer que plusieurs descriptions intensionnelles (entités mentales représentées dans un système) peuvent correspondre à un seul objet dans le monde extérieur (extension unique de descriptions multiples).

Le premier "objet" à supporter la notion de coréférence en représentation des connaissances de façon embryonnaire semble être le *Nexus* de KL-ONE, introduit dans [Bra&85] dans le chapitre sur le langage assertionnel, qui préfigure la ABox des langages terminologiques (cf. page 22). Le nexus y est présenté comme une entité sans structure qui réunit des affirmations de coréférence ou de "co-spécifications de description". Si un objet satisfait une description (c. à d. un Concept en KL-ONE), l'assertion de cette information passe par la création d'un lien de description ("Description Wire") entre un Nexus et le Concept en question. Un même Nexus peut ainsi être relié à plusieurs Concepts. Les Nexus ("Nexi" ?) sont regroupés en Contextes, qui peuvent représenter des mondes ou états hypothétiques différents (croyances) dans le système. La multi-description induite par les liens de coréférences se situe donc elle-même à l'intérieur d'un contexte. La multiplicité des contextes pris comme des états hypothétiques (et éventuellement temporels) différents rejoint l'acception des points de vue multiples dans des systèmes comme Omega [Att&85] ("Viewpoints"), ART [Inf85], et CYC [Guh91] [Bla&92]("Contexts" et "Microtheories").

Ferber et Volle ([Fer&88] [Fer88] [Fer89] [Vol89]) étudient plus profondément l'aspect intensionnel des descriptions évoqué par Woods. [Fer89] élabore une théorie intensionnelle de la représentation, où les objets sont compris comme des points de vue dénotant un même référent. Les objets sont des implantations de concepts et dénotent des entités du monde. Les objets qui dénotent la même entité sont coréférentiels.

Dans la syntaxe de MADELEINE [Vol89], la multi-description de l'individu Laurent peut se formuler comme suit (cf. Fig. 15) :

```
(defgen personne ~ qqch                    ; définition de concept général
 (age ~ slot (range = (un nombre)))
 …)

(defgen voyageur ~ personne
 (age ~ slot (range = (un classe-d-age))))

…

(un personne => Laurent                    ; concept individuel
 (age = 22)                                ; de référent Laurent
 (prénom = "Laurent")
 (nom = "F…"))

(un voyageur => Laurent
 (age = adulte))

(un client => Laurent
 (crédit = (uneCarte => ?-
                (banque = 'CréditLyonnais)
                (numéro = "4567 Z"))))
```

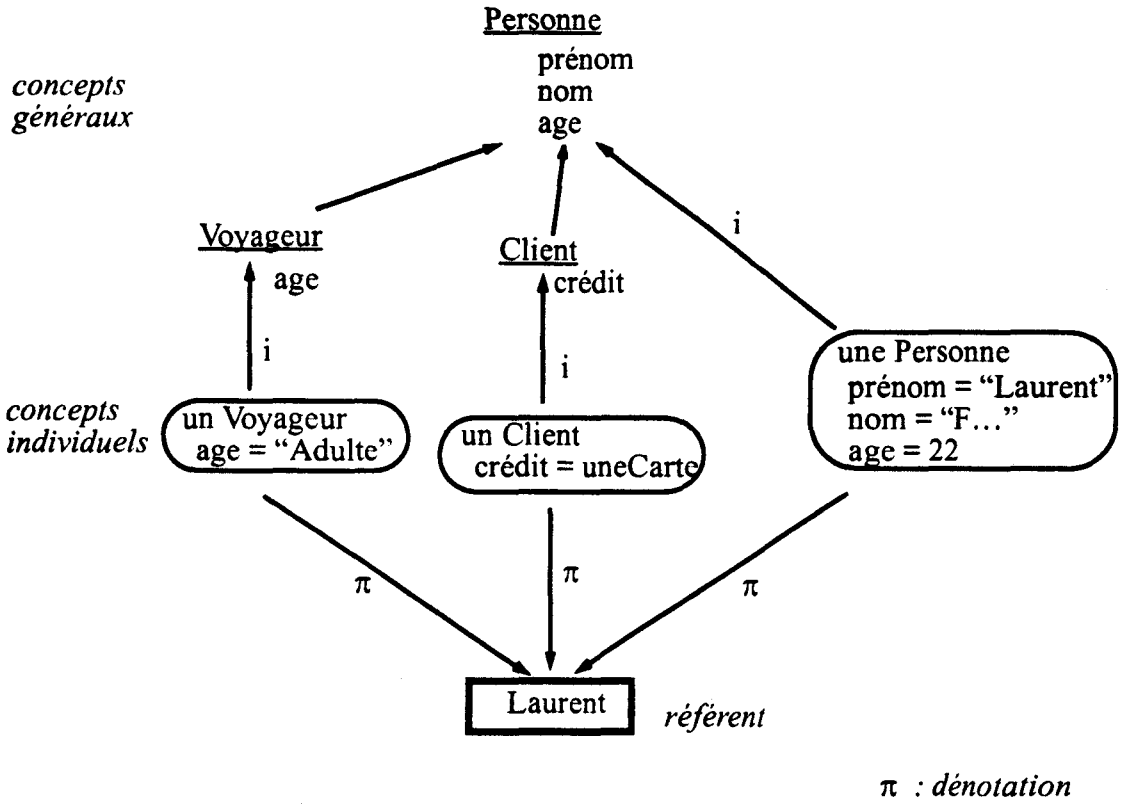


Fig. 16 Concepts individuels coréférents

Les concepts individuels qui dénotent le même référent constituent une classe de coréférence.

2.3.2.4 La représentation multiple

En plus de l'héritage multiple entre classes, ROME [Car89] [Car&90a] offre la représentation multiple d'objets. Les objets sont instances d'une classe d'instanciation unique, mais peuvent appartenir à plusieurs sous-classes de celle-ci, par des liens de *représentation*. Les différentes sous-classes peuvent appartenir à des sous-graphes distincts et indépendants, qui héritent tous de la classe d'instanciation, qui reste la classe d'appartenance essentielle et minimale de l'objet. Chaque appartenance établie par un lien de représentation vers une classe dans un des sous-graphes traduit la description de l'objet selon le point de vue représenté par le sous-graphe. Le problème de l'explosion combinatoire des classes produits, nécessitées pour l'instanciation des objets dans l'approche par héritage multiple, illustré Fig. 13 (page 41) est résolu, car les objets sont reliés directement aux différentes classes qui les décrivent (cf. Fig. 17). De façon générale, on ne définit que les classes qui ont une raison d'être, c.-à-d. qui spécialisent leurs superclasses par ajout ou redéfinition de caractéristiques.

Plusieurs contraintes contrôlent la représentation multiple en ROME :

- le **principe de représentation minimale**, qui contraint un objet à être toujours caractérisé à l'intérieur du domaine d'affinement du concept général correspondant à sa classe d'instanciation [Car89].
- les contraintes de représentation induites par l'**exclusion entre classes** : un objet ne peut

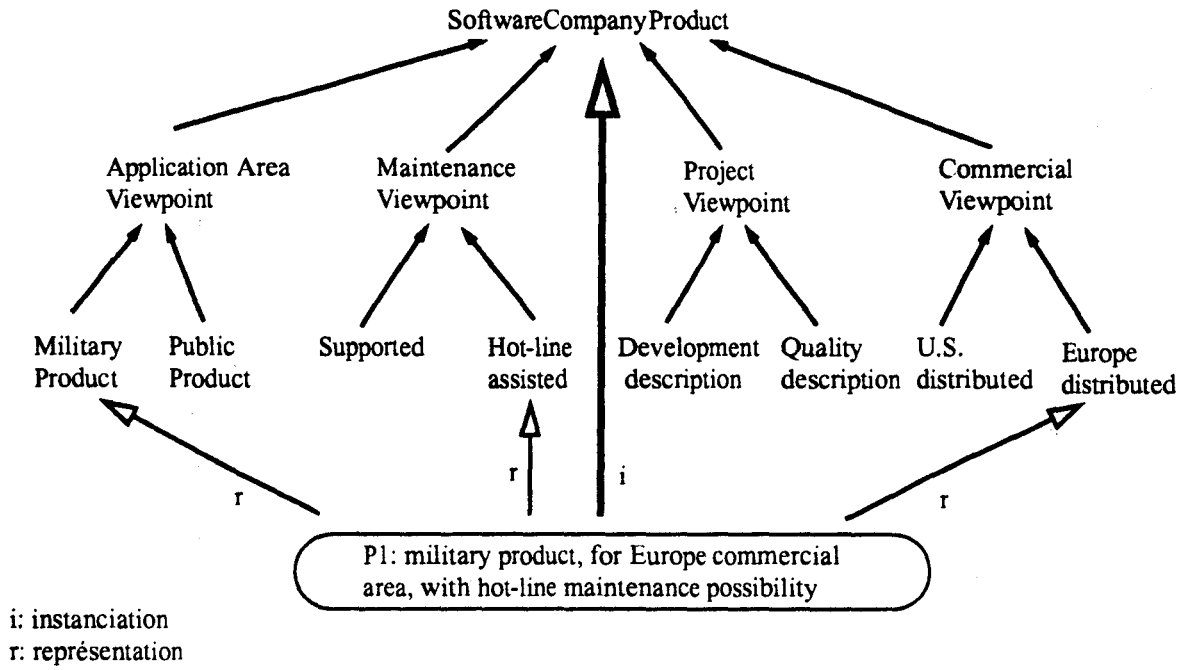


Fig. 17 Représentation multiple en ROME

appartenir à des classes qui sont en relation d'exclusion mutuelle [Dek89]. Exemple trivial : les sous-classes Homme et Femme de la classe Personne peuvent être déclarées comme étant en exclusion mutuelle, ce qui interdit aux instances d'être représentant des deux classes à la fois. Ceci entraîne également l'impossibilité de créer des sous-classes communes.

La représentation multiple de ROME est directement liée à sa représentation évolutive. Si la représentation multiple offre une échappatoire à la contrainte d'instanciation unique des langages orientés objet classiques, la représentation évolutive remet en question la contrainte d'instanciation figée. Tout en respectant le principe de représentation minimale et les contraintes d'exclusion, l'objet peut changer de représentation au cours de son évolution. Un étudiant peut terminer ses études et devenir fonctionnaire, ou chômeur ; un logement à vendre peut devenir un logement squatté, ou vendu, etc. Ces évolutions se traduisent par des ajouts ou retraits de liens de représentation entre un objet et ses classes d'appartenance. La représentation évolutive est un atout particulièrement puissant pour l'évolution d'une base de connaissances. Les instances existantes peuvent ainsi évoluer et leur description s'affiner quand la base est étendue par de nouvelles classes qui viennent spécialiser les classes d'appartenance de ces objets.

Un point fort de l'approche par représentation multiple et évolutive est la notion d'identité de l'objet, qui est centrale. C'est le même objet qui est décrit de façons multiples de par son appartenance à plusieurs classes ; c'est le même objet qui évolue, en restant toujours au moins instance de sa classe d'instanciation qui le décrit de façon minimale. Cette notion d'identité existe également dans l'approche par héritage multiple. Par contre, elle est absente notamment dans l'approche par agrégation, où l'objet est éclaté en sous-objets. La combinaison de la représentation évolutive avec la représentation multiple a l'avantage de ne pas réclamer des combinaisons de classes particulières (ce qui serait inévitable avec l'héritage multiple) au cours de l'évolution d'un objet. L'évolution d'objet dans l'approche par agrégation entraîne de

nombreux problèmes de gestion de la cohérence des sous-objets, exposés en détail dans [Car89].

L'idée de représentation évolutive a été reprise dans [Aug&91] qui développe un mécanisme de gestion de versions, comme des vues temporelles d'objets. La représentation évolutive d'objets sans la représentation multiple est également disponible en KOOL [Alb88] et en SMECI [ILO91]. Hendler propose le traitement de l'évolution d'objets par mixins, appelés *enhancements* [Hen86]. Ces mixins ajoutent des fonctionnalités aux instances (et non aux classes, comme les mixins utilisés pour l'héritage multiple), sans constituer de spécialisation de leurs classes d'appartenance. Ainsi, l'aptitude dont dispose une personne par le fait d'être médecin est considérée non pas comme une spécialisation de la classe Personne (qui se traduirait par une sous-classe) mais comme une fonctionnalité supplémentaire indépendante, ajoutée à n'importe quel objet par "enhancement".

La représentation multiple est appliquée de façon systématique en TROPES [Mar89] [Mar&90], qui est un modèle de représentation dédié à la classification d'instances multi-point de vue. Chaque objet appartient à un concept (et un seul). Un concept est décrit selon un nombre fixe de points de vue, chacun représentant une arborescence de classes. L'objet appartient à exactement une classe sous chaque point de vue. Etant donnée une instance d'un concept, l'algorithme de classification essaie de trouver la classe d'appartenance la plus spécifique dans chaque point de vue. Chaque point de vue comporte un sous-ensemble d'attributs du concept (les attributs visibles du point de vue), qui peuvent être affinés sous ce point de vue. Un point de vue particulier peut induire une décomposition particulière des objets du concept [Mar91].

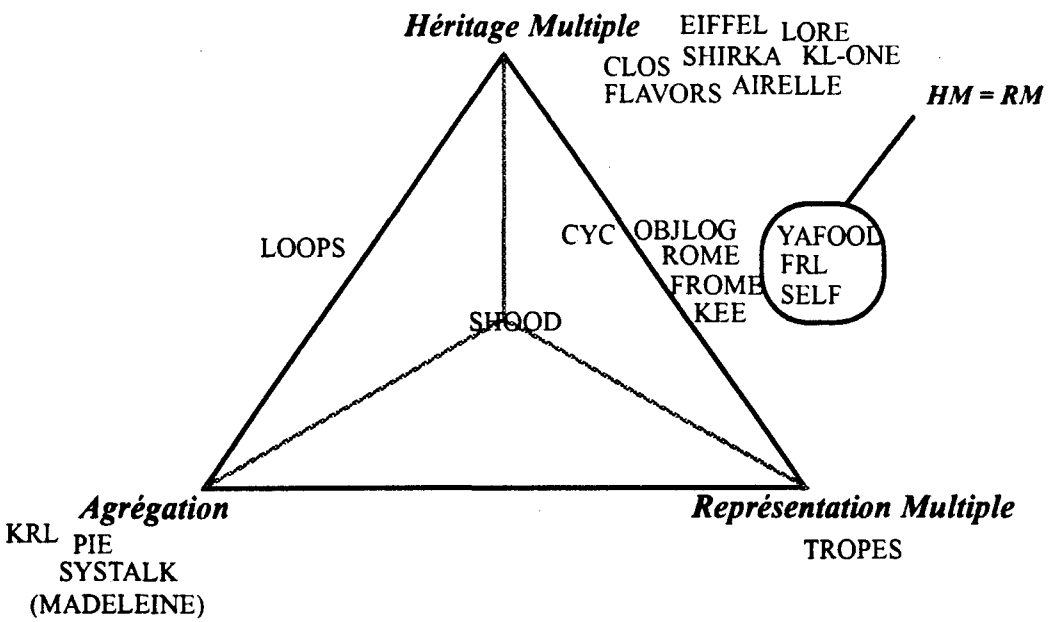


Fig. 18 Différentes méthodes de représentation multiple

Le langage OBJLOG [Dug88] [Dug91] permet des liens *sorte-de* et *est-un* multiples. La notion de point de vue est définie pour fixer l'interprétation d'un attribut par rapport à une classe.

Dans SHOOD [Rie&91a/b], la représentation multiple d'objets est appelée multi-instanciation, car (comme en OBJLOG et contrairement à ROME) il n'y a pas de

différenciation entre le lien qui lie une instance à sa classe de création et le lien qui la relie à d'autres classes d'appartenance. Il n'y a pas de notion de représentation minimale sous la classe privilégiée qui a servi à la création de l'objet. L'héritage multiple semble exister mais son utilisation n'est pas encouragée.[1]

Notons que pour les langages qui ne font pas de distinction entre classes et instances, ou plus exactement, qui utilisent une seule relation de spécialisation entre classes et instances, la distinction entre représentation multiple et héritage multiple n'a pas de sens. Il s'agit simplement d'une relation de spécialisation multiple. Dans la Fig. 18, nous avons entouré ces langages (noté HM=RM).

MADELEINE figure entre parenthèses du côté de l'agrégation, car l'approche par coréférence s'apparente à l'agrégation. Ce système a en commun avec ceux qui l'entourent la prise en compte de la multi-description par la multiplicité des objets. Contrairement aux autres, il ne regroupe pas ces objets multiples au sein d'un objet agrégat.

2.3.2.5 L'approche orientée vue

Une approche assez originale (et isolée) est celle de H. Davis, implantée dans son système VIEWS [Dav87]. Nous l'appelons l'approche orientée vue, car la brique de base du système est la "vue", et non pas l'objet représenté. On représente des situations, par des réseaux de nœuds (parties) reliés par des relations et des contraintes. Une telle structure se rapprocherait le plus de la notion de *frame-system* telle qu›elle est décrite par Minsky [Min75]. Davis soutient que ces vues en forme de réseaux structurels sont adaptées pour la représentation de perspectives spatiales (décrites également dans [Min75]), aussi bien que pour représenter des taxonomies multiples (chaque taxonomie est une vue) et des vues non-monotones comme les *viewpoints* du système Omega (cf. page 44).

2.3.3 Relations entre points de vue

Les descriptions issues de différents points de vue sont disjointes ou non, mais même dans le cas de points de vue disjoints, ces descriptions sont en relation par le fait même qu'elles partagent leur objet de description. Dans les paragraphes suivants, nous décrivons les relations implicites induites par l'héritage et la représentation multiples, les relations explicites qui permettent de passer d'une représentation à une autre (cas des règles de coréférence), et les relations de transformation (cas d'une interprétation spatiale des points de vue).

2.3.3.1 Relations implicites

Les relations implicites entre points de vue favorisent le partage et la combinaison de caractéristiques. Dans l'approche par héritage multiple, les liens entre points de vue sont induits par l'héritage même et sont exprimés implicitement par la sous-classe réunissant plusieurs points de vue par spécialisation. La sous-classe hérite de toutes les caractéristiques des différents points de vue et peut encore les affiner. Certains langages fournissent des moyens pour spécifier comment doit opérer l'héritage. Flavors, qui met en avant l'héritage multiple comme moyen de combiner des modules, propose plusieurs types de combinaisons de méthodes pour une sous-classe qui hérite de spécialisations multiples d'une méthode. L'un

1. Pour des raisons peu claires, l'utilisation de l'agrégation est prônée en cas de spécialisation multiple.

de ces types de combinaisons (l'utilisateur peut en définir d'autres) consiste en l'appel de tous les démons *before* et *after* hérités, autour de la méthode *primary* la plus spécifique. Les différents modules (*flavors*) combinés par héritage multiple "communiquent" par combinaison de méthodes et partage de variables d'instance [Moo86][1]. KEE définit une panoplie de types d'héritage au choix, permettant de combiner des valeurs ou des méthodes héritées. Comme en FLAVORS, l'héritage des méthodes fait intervenir des combinaisons de démons : *beforecode*, *aftercode* et *wrappercode*, autour du *maincode* (le plus spécifique) [Int85]. Citons également YAFOOL, qui propose des accès aux attributs en I, en N et en Z [Duc91].

Le partage d'un attribut peut entraîner la combinaison des descriptions l'affinant sous différents points de vue. Ainsi, en OBJLOG, la fusion des descriptions d'attributs de même nom et de même type est implicite dans une sous-classe de plusieurs classes, pourvu qu'il s'agisse d'un attribut hérité d'une superclasse commune (attribut partagé par différents points de vue). Cette fusion est appelé unification des champs sémantiquement équivalents (considérés comme ayant la même interprétation) [Dug88].

La solution par représentation multiple porte également des liens implicites entre points de vue, non pas à travers une classe intersection de points de vue, mais dans les objets eux-mêmes. L'objet représenté selon plusieurs points de vue réunit les caractéristiques décrites dans les classes de chacun de ses points de vue. Un attribut de l'objet décrit de manière multiple correspond à toutes ses descriptions partielles. Il satisfait en particulier toutes les contraintes imposées dans les affinements selon différents points de vue. Ceci est vérifié pour les attributs sémantiquement équivalents dans OBJLOG, et aussi dans TROPES [Mar89], où un attribut peut être affiné avec des restrictions de domaine dans différents points de vue à l'intérieur du concept.[2]

2.3.3.2 Relations explicites

A part les relations induites par l'héritage, plusieurs moyens ont été proposés pour la mise en relation explicite des descriptions issues de différents points de vue. En OBJLOG, l'unification sémantique de deux champs (non partagés) peut être demandée explicitement si ces champs sont compatibles (même nom, même type).

Dans les approches ne bénéficiant pas du partage et de la combinaison induits par l'héritage multiple, comme l'agrégation et l'approche par coréférence, toutes les relations entre points de vue doivent être spécifiées explicitement.

[Vol89] décrit les règles de coréférence qui sont censées gérer la cohérence entre descriptions coréférentielles. Ces règles établissent des correspondances entre descriptions, qui servent à traduire une description dans une autre. L'application de ces règles peut donner lieu à l'émergence de nouveaux objets, puisque l'existence d'une description se traduit en celle d'une autre. Ces règles expriment également des contraintes qui relient les différentes descriptions coréférentielles.

Un rôle similaire à celui des règles de coréférence est joué par les passerelles de TROPES. Ce sont des relations entre classes appartenant à différents points de vue, qui indiquent une

1. [Moo86] fait malheureusement l'amalgame entre le lien de composition et celui de spécialisation. Ainsi, la fameuse tarte aux pommes se trouve hériter de pâte, de cannelle et de pomme.
2. Ce principe, que nous appelons le principe de multiplicité des contraintes, est également respecté par FROME pour l'affinement multiple des slots, comme nous allons le montrer dans 3.3.1.

implication (cas des passerelles uni-directionnelles) ou une équivalence (passerelles bi-directionnelles). Les passerelles sont utilisées intensivement lors du processus de classification, car elles représentent des contraintes d'appartenance pour les instances : une instance appartenant à une classe en "prémisse" d'une passerelle appartient, par définition, à la classe en "conclusion". L'utilisation des passerelles est donc un moyen original de faire avancer la classification des instances, en navigant d'un point de vue à l'autre. Nous reviendrons sur la classification en TROPES dans la partie II de ce mémoire.

Des contraintes explicites de ce type sont utilisées en VIEWS, pour imposer des liens d'équivalence entre objets faisant partie de plusieurs structures ("vues"). Ainsi, on exprime par une contrainte que l'objet Clyde, qui est un éléphant de cirque dans la taxonomie des intervenants dans un cirque, doit être un éléphant dans la taxonomie des animaux.

2.3.3.3 Règles de transformation

Un autre type de relation, intervenant dans la représentation des perspectives spatiales [Min75] [Dav87], est celle exprimée par des règles de transformation. Elles expriment comment passer d'une vue (situation) à une autre, à travers des liens comme "avancer", "reculer", "tourner à droite", etc. Un très grand nombre de vues ainsi interconnectées pourraient, par exemple, représenter l'intérieur d'une pièce, ou plusieurs pièces communiquant entre elles, et permettre à un robot de s'y promener.

2.3.4 Sélection de description

Si la multi-description se fait selon le principe de la modularité, alors les descriptions sont indépendantes. Chaque description peut être élaborée indépendamment des autres, ce qui implique le libre choix des caractéristiques à définir. La multi-description peut alors mener à la définition de caractéristiques homonymes. Un objet décrit de manière multiple se trouve alors avoir plusieurs caractéristiques de même nom, ce qui peut poser des problèmes d'accès à ces caractéristiques. Ces problèmes sont communément appelés "conflits d'héritage multiple". De nombreuses études ont tenté d'éclaircir le problème des conflits, dont [Sny87] [Knu88] [Duc&89] [Car&90b] [Duc&92a/b][Duc93].

La multiplicité des descriptions pose surtout des problèmes dans le cadre du génie logiciel, où la fiabilité est une caractéristique primordiale. L'ambiguïté pouvant mener à des erreurs lors de l'exécution, de nombreux systèmes, s'ils ne renoncent pas purement et simplement à l'héritage, interdisent la multiplicité effective.

D'autres systèmes, au contraire, veulent maintenir la multiplicité et proposent des moyens de sélection d'une description particulière.

2.3.4.1 Inhibition de la multiplicité

L'inhibition de la multiplicité se traduit de différentes façons. La première est le renommage obligatoire, pratiqué par EIFFEL [Mey88]. Dès qu'une classe hérite de plusieurs caractéristiques de même nom, l'ambiguïté doit être levée en les renommant. Une classe présentant une ambiguïté est refusée par le compilateur.

Une caractéristique héritée selon plusieurs chemins différents est considérée comme partagée : elle ne donne pas lieu à des conflits. Le renommage d'une caractéristique partagée

permet néanmoins de la dupliquer. Cette possibilité est appelé "héritage répété". Il n'y a pas de limitation pour le nombre de fois que l'on peut hériter d'une classe et dupliquer ses caractéristiques dans une sous-classe.

Une autre façon d'éliminer la multiplicité est d'identifier une caractéristique à son seul nom et de fusionner toutes les descriptions qui s'y rapportent. Ainsi, en TROPES, l'espace de définition d'un attribut est le concept. Cela veut dire que le nom d'un attribut désigne le même attribut dans toutes les classes des différents points de vue qui décrivent un concept. Comme la description d'un attribut dans une classe TROPES se résume à une spécialisation de son type par ajout de contraintes, l'attribut d'un objet appartenant à plusieurs classes qui spécialisent cet attribut respecte la totalité des contraintes. Il n'y a pas de conflit entre descriptions différentes d'un même attribut.

La façon la plus répandue de réduire la multiplicité est sans doute la linéarisation, pratiquée dans de nombreux langages comme algorithme de *look-up*. La linéarisation consiste à transformer l'ordre partiel induit par le graphe d'héritage multiple en un ordre total. On se ramène ainsi au cas de l'héritage simple. La linéarisation considère toutes les caractéristiques de même nom comme des caractéristiques identiques. Les conflits qui sont résolus par la linéarisation ne sont alors pas des *conflits de noms*, mais des *conflits de valeurs* [Duc&92a]. Le conflit de nom étant celui entre deux attributs *différents* mais de même nom (attributs homonymes), le conflit de valeurs, que l'on pourrait appeler aussi conflit de définitions ou conflit de descriptions surgit quand la *même* propriété est décrite de façon multiple. La première valeur (description, définition) de la caractéristique rencontrée selon l'ordre total calculé est retenue. La linéarisation peut traduire des parcours en profondeur d'abord, en largeur d'abord, ou des parcours plus sophistiqués, tels que les extensions linéaires [Duc&87], qui ont la caractéristique de préserver la modularité du graphe d'héritage (voir [Duc&89] qui contient une étude détaillée des linéarisations pour l'héritage multiple). Toute linéarisation passe néanmoins par un choix (presque toujours arbitraire) d'un ordre sur les super-classes d'une classe (appelé la "multiplicité" dans [Duc&89]). Le but de la linéarisation est d'obtenir une caractéristique unique (toujours la même). Elle supprime donc nécessairement et volontairement la multiplicité des descriptions.

Notons que dans les techniques de linéarisation, il y a souvent des mécanismes qui permettent de faire appel à la caractéristique "suivante" dans l'ordre produit par la linéarisation, par exemple le *call-next-method* de CLOS [Kee89].

2.3.4.2 Sélections explicites

Les stratégies linéaires pour l'héritage multiple évoquées ci-dessus ont pour effet la réduction de l'héritage multiple à l'héritage simple. Elles ne conservent pas la multiplicité induite par la hiérarchie, qui est en forme de graphe.

D'autres approches tentent de conserver cette multiplicité, sans transformer le graphe. Celles-ci sont appelées stratégies graphiques (*graph-oriented strategies*) dans [Sny87]. En principe, toutes les caractéristiques issues de classes indépendantes sont héritées par leurs sous-classes communes (Principe d'Indépendance [Car89]). Plusieurs de ces langages proposent alors des moyens de désigner les caractéristiques homonymes multiples de manière non-ambiguë.

Citons les sélecteurs composés de Smalltalk étendu à l'héritage multiple [Bor&82] :

<classe>.<sélecteur>

qui désignent explicitement la classe à partir de laquelle une méthode ambiguë de nom <sélecteur> doit être recherchée (en remontant). Cette solution ressemble à une sélection de point de vue. Cependant, cette sélection explicite résout l'ambiguïté au coup par coup, et ne maintient pas le point de vue exprimé pendant l'application de la méthode. D'autres conflits peuvent apparaître sur des caractéristiques ambiguës utilisées à l'intérieur des méthodes, liés directement au premier conflit. Pour éviter ces conflits, chaque méthode ambiguë doit être redéfinie dans la sous-classe où apparaît le conflit en remplaçant les caractéristiques ambiguës référencées par des désignations complètes.

Cette technique est non seulement contraignante mais amène de nouveaux problèmes, liés à l'affinement des caractéristiques désambiguïsées : la redéfinition éventuelle de ces caractéristiques dans une sous-classe n'est pas prise en compte, puisque la désignation complète a pour effet que la recherche ne commence qu'à partir de la classe désignée. La généricité interne des méthodes, due à la prise en compte de l'affinement de caractéristiques référencées par elles est donc invalidée par l'utilisation des techniques de désignation explicite. La sélection explicite se révèle être trop rigide pour réellement manipuler la multiplicité des caractéristiques.

Tous ces problèmes sont exposés en détail dans [Car&90b].

2.3.4.3 Sélections de points de vue

La notion de point de vue de ROME propose une réponse aux problèmes attachés à la désignation explicite évoqués ci-dessus.

Rappelons brièvement les principes qui sont à la base de l'héritage multiple en ROME [Car89]. Ces principes sont illustrés sur la Fig. 19.

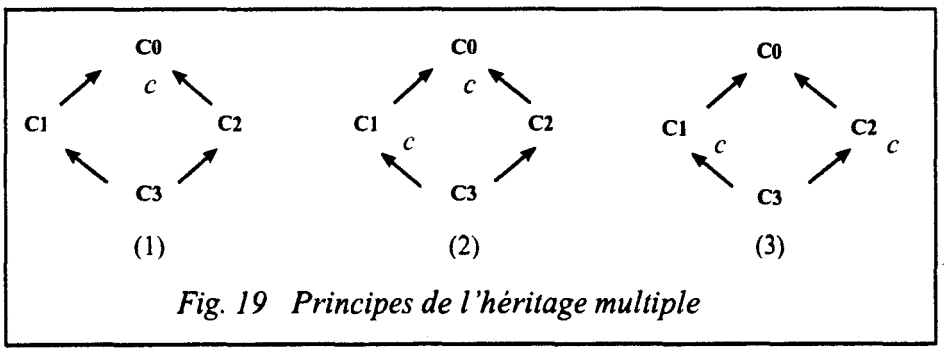


Fig. 19 Principes de l'héritage multiple

(1) Les caractéristiques (attributs, sélecteurs et méthodes) qui sont accessibles suivant plusieurs chemins d'héritage sont héritées **une seule fois** (pas d'héritage répété). Les instances de C3 ont une seule occurence de la caractéristique *c*.

(2) Une classe qui redéfinit une caractéristique masque celle héritée. Les instances de C3 ont une seule caractéristique *c*, qui est celle de C1. Ce principe est connu sous le nom de **principe d'affinement**. En ROME il s'applique aux méthodes et aux sélecteurs.

(3) Le **principe d'indépendance** : il n'y a pas de conflit d'héritage entre caractéristiques homonymes héritées de classes indépendantes. Les instances de C3 ont deux caractéristiques *c*. La notion de **point de vue** intervient pour exprimer la préférence

pour les caractéristiques d'une sous-classification par rapport à une autre.

Un point de vue sur un objet est défini comme un sous-ensemble du graphe et est déterminé par une classe :

> Soit l'**ensemble de représentation** d'un objet O l'ensemble des classes de représentation de O, i.e. toutes les classes auxquelles est relié O par un lien de représentation ou un lien d'instanciation, et leurs super-classes. Le graphe de représentation est noté Rep(O).
>
> Soit la **dépendance** de C l'ensemble de toutes les super-classes et des sous-classes et C elle-même. La dépendance de C est notée Dep(C).
>
> Le **point de vue** d'une classe C sur un objet O est défini :
>
> $$PdV(C,O) = Rep(O) \cap Dep(C)$$

La sélection de point de vue s'exprime par une as-expression :

(<caractéristique> as <classe>)

Les as-expressions de ROME sélectionnent un point de vue sur une caractéristique ambiguë (attribut, sélecteur, méthode). Un point de vue implicite est maintenu pour désambiguïser les caractéristiques en cas d'absence de point de vue explicite. Le point de vue implicite est établi par les règles suivantes :

- pour un envoi de message, le point de vue implicite est celui des classes de représentation directe de l'objet : il détermine l'ensemble de représentation de l'objet Rep(O).
- pour une application de méthode, ou une référence à un attribut, le point de vue implicite est celui de la classe dans laquelle se trouve cette référence : il détermine l'ensemble $Rep(O) \cap Dep(C)$.

En tout état de cause, l'application d'un point de vue (explicite ou implicite) n'intervient que quand il y a ambiguïté. L'application du point de vue a pour effet de choisir entre plusieurs caractéristiques celle qui est définie dans une classe faisant partie du point de vue, tout en respectant le principe d'affinement. Il ne s'agit pas d'une restriction du graphe d'héritage a priori, qui pourrait résulter en un contournement d'affinement "sideways" : l'expression [O '(c as C2)] pour O représentant de C3 dans le cas (2) de la figure résulte en une sélection de la version affinée de c, définie dans C1. L'affinement est donc toujours respecté.

Notons qu'une sélection de point de vue trop général peut maintenir l'ambiguïté : il faut alors préciser l'expression en utilisant une classe plus spécifique.

Les sélections de points de vue en Objlog [Dug88] [Dug91] ressemblent à celles de ROME, mais ne concernent que les attributs. La stratégie d'application des méthodes est en profondeur d'abord avec *back-tracking*, en respectant le masquage : s'il y a plusieurs méthodes de même nom, elles sont toutes appliquées, sauf si elles sont masquées. Les points de vue associés aux attributs sont exprimés dans des *étiquettes*. L'association d'une étiquette lors de la référence à un attribut permet de le distinguer des autres attributs de même nom. Si un même attribut est affiné dans plusieurs classes indépendantes, et qu'une classe en hérite, il s'agit d'une unification sémantique de ces affinements, et l'étiquette associée contient toutes

les classes indépendantes en question. Toute référence à cet attribut doit être accompagnée de la totalité des classes de l'étiquette (et exactement celles-là), ce qui semble trop contraignant.

Dans les approches par agrégation (KRL, PIE, LOOPS), la sélection de caractéristiques passe nécessairement par la sélection d'une perspective. La caractéristique est alors celle de l'objet perspective. Comme le précise [Gol&80] et le rappelle Wolinski pour son système SYSTALK [Per&92], "dans la plupart des cas, l'expéditeur du message connaît le point de vue que le récepteur doit employer pour l'interpréter".

Exemple d'accès à l'attribut `crédit` de Laurent (cf. Fig. 15) :

```
((laurent as: Client) crédit)
```

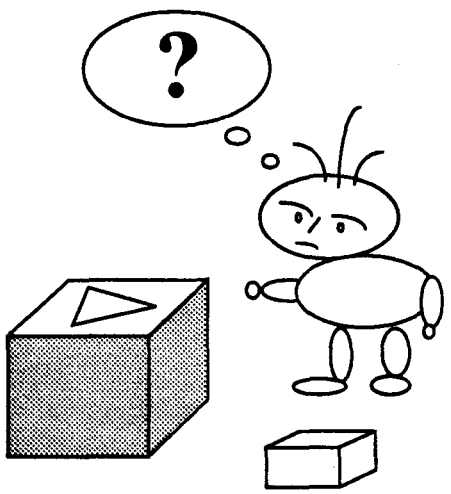
où `(laurent as: Client)` retourne l'objet perspective.

2.4 Formes de raisonnement

Dans l'approche de RPO, diverses formes de raisonnement permettent d'exploiter les connaissances représentées. L'héritage reste le principal mécanisme d'inférence, "cablé" dans la structure même d'une base de connaissances à objets. A ce mécanisme de base, et exploitant celui-ci, s'ajoutent des mécanismes de raisonnement pour des besoins spécifiques de l'intelligence artificielle : appariement d'objets, filtrage, spécialisation, classification, inférence à l'aide de règles de production. Une exploitation possible d'une base d'objets peut être la consultation en mode navigation. Le couplage d'une base de connaissances à objets et d'un outil de navigation sous forme d'hypertexte a été expérimenté par Grivaud et Rechenmann [Gri&92].

Dans les paragraphes suivants, nous étudions l'utilisation de l'appariement, du filtrage, des règles de production et de la classification comme mécanismes de raisonnement dans les divers systèmes de RPO.

2.4.1 L'appariement

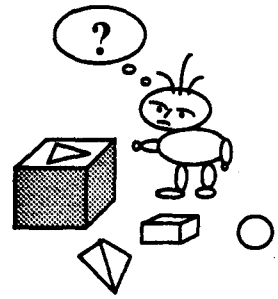


2.4.1.1 Filtrage, classification, appariement, *pattern-matching* ?

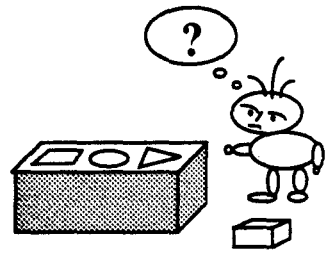
Une certaine confusion semble régner autour des termes de filtrage, classification, appariement et *pattern-matching*. Filtrage et appariement sont couramment confondus et utilisés indifféremment pour traduire *pattern-matching*. Tantôt, le filtrage est assimilé au mécanisme de classification d'instances, tantôt à la sélection d'instances vérifiant un ensemble de conditions appelé filtre ([Mas&89] glossaire).

Nous proposons la distinction suivante : **l'appariement** ou *pattern-matching* correspond à la comparaison d'un objet et d'un modèle. Il s'agit d'établir si l'objet correspond (éventuellement en partie) au modèle.

Le filtrage consiste à retrouver toutes les instances qui correspondent à un modèle appelé filtre. Le filtrage utilise donc l'appariement. Il s'agit de l'appariement d'un seul modèle avec plusieurs instances candidates. Cf. 2.4.2



La classification d'instances consiste à trouver la (les) classe(s) d'appartenance (modèles) d'une instance. La classification fait donc appel à l'appariement également ([Byl&89]). Il s'agit de l'appariement de plusieurs modèles candidats avec une seule instance. Cf. 2.4.4 et la partie II de ce mémoire.



En termes de subsomption : le filtrage consiste à trouver les objets qui sont subsumés par le filtre. La classification consiste à trouver les classes qui subsument l'objet à classer.

2.4.1.2 L'appariement en KRL

KRL [Bob&77] rejette l'idée d'une description complète d'un objet en termes de sa structure qui serait sa "définition". La description d'un objet est plutôt construite selon des points de vue différents sur l'objet (cf. 2.3.2), basés sur des comparaisons avec d'autres objets (prototypes).

La structure donnée dans une description — intensionnelle — peut être utilisée pour reconnaître une entité conceptuelle et pour la comparer avec d'autres entités. Les trois opérations fondamentales sont alors la spécialisation d'une description par l'ajout de nouvelles connaissances, l'appariement de deux descriptions données pour déterminer leur compatibilité par rapport à un but particulier, et la recherche d'entités correspondant à une description particulière.

L'appariement joue donc un rôle important en KRL, aussi bien pour le filtrage que pour la classification. [Bob&77] passe en revue différentes utilisations et stratégies d'appariement possibles, réunies sous le terme de "description matching". Seules quelques stratégies simples ont été réellement implantées en KRL, les autres restant surtout une source d'inspiration pour la conception de langages ultérieurs.

Le processus d'appariement prend deux arguments : un modèle (*pattern*) et un objet spécifique à tester. Il s'agit en fait de deux descriptions, composées chacune d'un certain nombre de "descripteurs". En principe, tous les descripteurs d'un modèle doivent être satisfaits par la donnée (l'objet) pour que l'appariement soit complet. L'annotation *Criterial* permet d'indiquer éventuellement les descripteurs suffisants pour prouver l'appariement. Le

résultat le plus simple d'un appariement est le succès, avec un ensemble de liaisons entre les variables du modèle et les constantes (ou variables) de la donnée, ou l'échec.

Dans le cas de descriptions complexes, l'appariement est organisée en tâches et sous-tâches gérées dans un agenda. Ces sous-tâches peuvent être des procédures spécialisées qui essaient de déduire des informations supplémentaires, en utilisant l'héritage, ou de les chercher ailleurs, y compris par interrogation de l'utilisateur. L'ordre d'exécution des sous-tâches, l'arrêt, les ressources allouées,... peuvent être contrôlés par l'utilisateur. Le résultat peut alors être partiel : à part le succès ou l'échec, il peut y avoir une évidence insuffisante. Après examen des résultats intermédiaires, l'utilisateur peut alors décider de continuer avec plus de ressources, ou d'arrêter.

Une utilisation comparable à la classification est l'appariement de plusieurs modèles avec une seule donnée, afin de trouver le "*best match*", par exemple dans une application de diagnostic médical. Il faudrait alors un moyen d'associer des mesures de la qualité de l'appariement et aussi des mesures indiquant la fiabilité du résultat.

Les mêmes mécanismes d'appariement peuvent être utilisés pour le filtrage : plusieurs unités candidates sont appariées avec un modèle décrivant une requête. Bobrow et Winograd pensaient combiner le filtrage avec d'autres mécanismes, basés sur des liens associatifs entre unités, et la structuration de la base de connaissances par contextes pour y retrouver des unités correspondant à une description donnée.

Une idée originale est celle de l'appariement forcé. Ce mode d'appariement répond à la question : "Qu'est-ce qu'il faut croire pour que l'appariement marche ?".

L'exemple suivant (Fig. 21) est librement inspiré de [Bob&77] et Platon[1] et montre l'intérêt que peut avoir ce type d'appariement pour exprimer des analogies.

L'appariement forcé de "Cours-de-Politique" et "TransactionCommerciale" retournerait l'hypothèse à accepter pour que "ça match": il faut considérer

- un Elève comme un Client
- un Professeur comme un Marchand
- la Connaissance comme un Produit.

Cette hypothèse reflète une analogie entre marchands et professeurs[2].

Le procédé de l'appariement forcé semble avoir un intérêt double :

- assistance pour reformuler des descriptions erronées, au lieu d'un rejet catégorique de type échec d'appariement
- trouver des analogies entre descriptions.

Dans la même optique on peut essayer d'apparier 2 "individuals" (différents par définition). Le résultat serait une spécification de la manière dont ils diffèrent.

1. Platon, "Protagoras ou les sophistes", in *Protagoras et autres dialogues*, Trad. E. Chambry, Garnier Flammarion, Paris, 1967.

2. Dans le dialogue entre Socrate et Protagoras, il est question d'évaluer l'objet et l'intérêt de l'enseignement du sophiste Protagoras pour le jeune Hippocrate. Comme tout professeur, Protagoras "vend" sa science et peut donc être comparé à un marchand. Pour Socrate, les sophistes sont des "marchands de denrées spirituelles", étant donné qu'ils vantent ce qu'ils vendent (comme les marchands ambulants qui vendent des aliments), mais que l'on ne peut pas leur faire confiance quant aux ingrédients de leurs produits, qui rendent peut-être malade celui qui les ingurgite (analogie entre le corps malade et l'âme malade).

```
[Transfert UNIT Specialization
     <SELF          (an Evenement)>
     <destinataire  (a Personne)>
     <origine       (a Personne)>
     <objet         (a Thing)>]

[Cours UNIT Specialization
     <SELF          (a Transfert)>
     <destinataire  (an Eleve)>
     <origine       (a Professeur)>
     <objet         (a Connaissance)>]

[TransactionCommerciale UNIT Specialization
     <SELF          (a Transfert)>
     <destinataire  (a Client)>
     <origine>      (a Marchand)>
     <objet>        (a Produit)>]

[Cours-de-Politique UNIT Individual
     <SELF          (a Cours with
                      destinataire  = Hippocrate
                      origine       = Protagoras
                      objet         = Vertu)>]

[Hippocrate UNIT Individual
     <SELF          (an Eleve)>]

[Protagoras UNIT Individual
     <SELF          (a Professeur)>]
```

Fig. 21 Professeurs et Marchands

2.4.1.3 Prédicats de correspondance en MERING

MERING I [Fer83], écrit en VLisp, définit un ensemble de primitives ("prédicats") de comparaison d'entités, sous la forme de méthodes et d'applicateurs. Un applicateur est une méthode appliquée à un couple <objet, attribut>. Ces "prédicats de correspondance" comparent aussi bien des valeurs que des domaines d'attributs.

On distingue entre prédicats d'égalité, de contenance (appartenance) et de correspondance (appariement). Le prédicat d'égalité (=) teste l'égalité de deux entités sur la base d'une comparaison des valeurs d'attributs. Les prédicats de contenance décident si un attribut contient une ou plusieurs valeurs d'un certain type :

```
:(laurent locomotion cont?-un velo)
= peugeot-sprint
;l'attribut locomotion de laurent contient-il une valeur de type velo?
;oui : peugeot-sprint

:(jean petits-fils cont?-des personnes)
= (pierre jacques)
;l'attribut petits-fils de jean contient-il des valeurs de type
personnes?
```

```
;oui : pierre et jacques
:(laurent locomotion == (voiture velo trottinette))
= peugeot-sprint
;l'attribut locomotion de laurent contient-il une valeur de type voiture,
;velo ou trottinette ?
;oui : peugeot-sprint est un velo
```

Les prédicats d'appariement comparent deux entités quant à la compatibilité des domaines de valeurs (facettes *$dom* et *$valeurs*) de leurs attributs. Je cite le prédicat *=?=* :

```
(jules passions =?= gugus passions)
```

retourne vrai si les facettes valeurs sont identiques ou si le domaine de l'attribut `passions` de `jules` est un sur-type du domaine de l'attribut `passions` de `gugus`. Le prédicat suivant :

```
(jules =?= gugus '(age passions))
```

compare `jules` et `gugus` uniquement par rapport aux attributs passés en argument. L'appariement peut donc être partiel, l'appariement complet (au sens de l'inclusion des domaines) étant exprimé par :

```
(jules =?= gugus).
```

2.4.2 Le filtrage

Le filtrage est le mécanisme par excellence qui fait appel à l'appariement. Un modèle, appelé filtre, est apparié avec plusieurs données. Le résultat du filtrage est l'ensemble des données qui correspondent au filtre.

2.4.2.1 Utilisation du filtrage

Le filtrage, dans un contexte de représentation par objets, peut être utilisé dans des requêtes au système de frames par l'intermédiaire d'un langage de requêtes comparable à ceux des bases de données (cf. SQL). Le but est alors de retrouver tous les objets existants dans le système qui satisfont les conditions de la requête. En KEE par exemple ([Int85]), de telles requêtes peuvent être exprimées à l'aide du langage de requêtes *TellAndAsk*, qui permet de faire des assertions ou de retrouver des informations dans la base de connaissances.

Une autre façon d'utiliser les filtres consiste à les attacher à des attributs au moyen de facettes spécialisées, comme la facette *$sib-filtre* de Shirka ([Rec&90]). La valeur de l'attribut est calculée au besoin en instanciant le filtre.

Enfin le filtrage joue un rôle primordial dans les systèmes intégrant objets et règles. Les variables des règles sont instanciées par des objets à travers un mécanisme de filtrage.

2.4.2.2 Filtrage par requêtes

L'interaction avec une base de connaissances est comparable à l'interrogation d'une base de données, abstraction faite des mécanismes mis en œuvre pour retrouver les informations demandées. Ces mécanismes peuvent nécessiter des inférences (héritage, déclenchement de facettes, règles,...) dans le cadre d'une base de connaissances, alors que dans une base de données "classique" il s'agit de parcourir les enregistrements de la base suivant un algorithme adéquat et de retenir ceux qui correspondent à la requête. Certains langages de frames

permettent d'exprimer des requêtes pour retrouver les frames qui répondent à certains critères.

FRL

FRL [Rob&77a] propose une fonction appelée FQUERY, qui permet de faire des requêtes avec une utilisation limitée de variables. La requête suivante permet de sélectionner tous les fils d'hommes politiques :

```
(FQUERY '(? (AKO       ($VALUE HOMME)
            (FILS-DE   ($VALUE ?PERE)
                       ($REQUIRE ((AKO? :VALUE 'HOMME-POLITIQUE))))))
```

Les variables (ici PERE) sont précédées d'un '?'. Le prédicat AKO? est vérifié si la valeur pour l'attribut FILS-DE est une spécialisation de HOMME-POLITIQUE. Les frames sélectionnés sont ceux dont les valeurs correspondent à la requête et dont les contraintes sont vérifiées. La contrainte associée à la facette $REQUIRE est purement fonctionnelle et ne permet pas d'imbriquer les contraintes associées aux variables. Une requête qui consisterait à introduire plusieurs variables à différents niveaux d'imbrication ne peut s'exprimer qu'avec une seule variable et une contrainte fonctionnelle plus ou moins complexe. S'il est possible d'exprimer une requête demandant les petits-fils de Jean, cette requête s'exprime peut-être comme ceci :

```
(FQUERY '(? (AKO     ($VALUE HOMME)
            (FILS-DE($VALUE ?PARENT)
                     ($REQUIRE
                            ((EQUAL (FGET :VALUE 'ENFANTS-DE '$VALUE)
                                    'JEAN)))))))
```

L'expression des contraintes se fait donc de façon ad hoc. Quand il s'agit d'exprimer des imbrications plus grandes, par exemple de trouver les arrière-petits-fils, la contrainte devient vite trop compliquée à exprimer.

La fonction particulière (FRINGE *frame slot*) retrouve toutes les feuilles d'un arbre construit par la fermeture transitive d'une relation, réalisée par la fonction FTREE. La relation est simplement matérialisée par un nom d'attribut et n'a pas de statut particulier. Si (FTREE 'JEAN 'PERE-DE) retourne l'arbre (JEAN (PAUL (JACQUES PIERRE))), le résultat de (FRINGE 'JEAN 'PERE-DE) sera (JACQUES PIERRE). D'autres fonctions de ce type sont (FCHILDREN *frame slot*) et (FDESCENDANTS *frame slot*), qui retournent respectivement les descendants immédiats du frame suivant la relation du slot et les descendants indirects suivant la clôture de cette relation [Rob&77b].

Une autre forme de requête est celle exprimée à l'aide de la fonction SELECT. La requête suivante cherche, parmi tous les descendants d'hommes politiques (FDESCENDANTS 'HOMME-POLITIQUE 'PERE-DE), les personnes qui s'intéressent à la politique :

```
(SELECT PERSON (FDESCENDANTS HOMME-POLITIQUE 'PERE-DE)
     (AND    (INDIVIDUAL? PERSON)
             (MEMBER 'POLITIQUE (FGET PERSON 'INTERESTS '$VALUE)))
```

Le test INDIVIDUAL? sert à ne retenir que les individus. En effet, en FRL, les liens d'instanciation et de sous-classement sont confondus en un seul lien de spécialisation AKO. Les frames représentant des classes ou des instances sont alors distingués par la valeur de leur slot CLASSIFICATION qui peut être "GENERIC" ou "INDIVIDUAL".

La différence entre les deux formes de requête FQUERY et SELECT semble être la possibilité

de manipuler des variables dans la première. FQUERY reste plus près de la structure du frame, alors que SELECT est une simple fonction Lisp.

KEE

Le langage **KEE** [Int85] comporte un langage de requêtes appelé *TellAndAsk*, destiné à gérer les interactions entre l'utilisateur et la base de connaissances. Ce langage lui permet de modifier la base en faisant des assertions et de retrouver des informations au travers de requêtes. Les expressions TellAndAsk ont deux formats : un format "langage naturel" et un format préfixe. Exemple d'assertion suivant les deux formats :

```
ASSERT '(ALL MOUNTAIN.BIKES ARE BIKES))
ASSERT '(SUBCLASS.OF MOUNTAIN.BIKES BIKES))
```

Ces assertions sont équivalentes et ont pour effet la création d'une sous-classe de la classe BIKES, appelée MOUNTAIN.BIKES. Les assertions peuvent créer de nouveaux objets, ajouter des slots à un objet, donner une valeur à des slots, à des facettes, modifier des liens de sous-classement et des liens classe/instance. Exemples d'affectations :

```
ASSERT '(THE CENTRE OF CERCLE.1 IS POINT.3))
ASSERT '(THE ABSCISSE OF (THE CENTRE OF CERCLE.1) IS 5.0))
```

La première assertion affecte POINT.3 au slot CENTRE de CERCLE.1. La deuxième assertion a pour effet l'affectation de la valeur 5.0 au slot ABSCISSE du frame en valeur du slot CENTRE du frame CERCLE.1. La fonction RETRACT défait une assertion.

Les requêtes sont exprimées avec la fonction QUERY. Une requête est la spécification d'un filtre contenant des variables. La fonction de traitement des requêtes retrouve les instances du filtre. Le langage est le même que pour les assertions. En outre, des variables (préfixées par '?') peuvent remplacer les termes :

```
? (QUERY '(THE ABSCISSE OF POINT.3 IS ?X))
= ((THE ABSCISSE OF POINT.3 IS 5.0))
```

Les expressions d'une requête peuvent être imbriquées, de façon à contenir des variables implicites. La requête suivante contient une variable implicite référençant le slot CENTRE de CERCLE.1 :

```
?(QUERY '(THE ABSCISSE OF (THE CENTRE OF CERCLE.1) IS ?X))
= ((THE ABSCISSE OF POINT.3 IS 5.0))
```

L'imbrication des requêtes semble pourtant se limiter au cas où les slots imbriqués sont monovalués. Une requête du type :

```
(QUERY '(THE FILS OF (THE ENFANTS OF JEAN) IS ?PETITS.FILS))
```

avec les slot FILS et ENFANTS multivalués ne semble pas autorisée.

L'utilisation des connecteurs logiques AND, OR, NOT autorise la construction d'expressions complexes[1], comme :

```
(QUERY '((THE CENTRE OF CERCLE.1 IS POINT.3)
     AND (THE CENTRE OF CERCLE.2 IS POINT.3)))
```

1. Un OR dans une expression ASSERT est placé sur une liste alternative de faits "non-structurés", qui ne peuvent être traduits sous forme d'objets, mais qui sont gérés à part.

Cette requête renvoie TRUE si les deux cercles CERCLE.1 et CERCLE.2 sont concentriques.

Il ne semble pas impossible de lier plusieurs variables dans une requête composée du style :

```
(QUERY '((THE ENFANTS OF JEAN IS ?ENFANTS)
         AND (THE FILS OF ?ENFANTS IS ?PETITS.FILS)))
```

car dans les prémisses de règles KEE, qui sont également exprimées en langage TellAndAsk, il peut y avoir liaison de plusieurs variables (cf. 2.4.2).

CLASSIC

Le langage **CLASSIC** [Bor&89] est l'un des successeurs de KL-ONE. Ce langage de description de concepts permet de construire une base de connaissances, composée de concepts et d'instances de ces concepts, de définir des règles et de formuler des requêtes. Les concepts sont partiellement ordonnés entre eux par un classifieur.

L'originalité des requêtes en CLASSIC est qu'elles ne retournent pas forcément une réponse en termes d'une liste d'instances. Etant donné le caractère partiel des descriptions de concepts, fondé sur l'hypothèse du monde ouvert, la réponse peut être descriptive, en termes des propriétés nécessaires des objets satisfaisant la requête.

Le langage de description permet de définir des concepts (classes), à l'aide d'un ensemble de constructeurs, de déclarer leurs attributs (rôles), et de créer des instances (individus). Exemples de définition :

```
define-concept[FABRICANT-DE-BICYCLETTE
                (ONE-OF Peugeot Motobecane Gazelle)]

define-role[locomotion]

define-concept[ETUDIANT-SPORTIF
                (AND ETUDIANT
                     (ALL locomotion VELO)
                     (AT-LEAST 2 locomotion))].
```

Création d'instance :

```
create-ind[Laurent]
```

A l'aide d'assertions, de nouvelles informations peuvent être associées aux instances, telles que l'appartenance à une classe ou des restrictions de rôles. Exemples :

```
assert-ind[Laurent, ETUDIANT]

assert-ind[Laurent, (AT-LEAST 1 locomotion)]

assert-ind[Laurent, (FILLS locomotion Peugeot-Sprint)]
```

Des relations de coréférence entre rôles permettent d'inférer des valeurs, et des règles propagent les conséquences de nouvelles informations ajoutées à la base. Exemple de règle :

```
assert-rule[ETUDIANT-SPORTIF,
                (ALL etat-sante (ONE-OF Excellent))]
```

Cette règle exprime le fait que tous les étudiants sportifs sont en excellent état de santé. Pour toute instance reconnue comme appartenant au concept ETUDIANT-SPORTIF, la règle permet de déduire l'état de sa santé. Ce n'est pas la même chose que d'en faire une partie de la définition du concept ETUDIANT-SPORTIF. En effet, il n'est pas nécessaire d'avoir une excellente santé pour être reconnu comme un étudiant sportif. La règle exprime une

conséquence et non une condition nécessaire.

Il y a plusieurs types de requêtes (recherche d'informations dans la base) :

- celles qui portent sur les aspects d'un concept et qui retournent des informations sur sa définition
- la requête `concept-subsumes[C1, C2]`, qui teste si deux concepts sont en relation de subsomption, c.à d. ssi tous les individus de `C2` sont nécessairement (par définition) aussi instances de `C1`
- les requêtes qui portent sur les valeurs des rôles
- les requêtes concernant l'appartenance à un concept.

C'est ce dernier type de requêtes qui nous intéresse plus particulièrement ici, puisqu'il correspond au filtrage des individus correspondant à un modèle. En CLASSIC, l'expression d'un concept quelconque peut être considérée comme une requête concernant tous les individus qui la satisfont, c.-à-d. les individus représentants de la classe définie par le concept correspondant. Exemple :

```
ask-necessary-set[?:Personne]
```

renvoie tous les individus connus comme instances de la classe `Personne`. La requête :

```
ask-necessary-set[?:(AT-LEAST 1 locomotion)]
```

renvoie tous les individus qui ont au moins une valeur pour leur rôle `locomotion`, ou dont la description de concept implique qu'ils ont au moins 1 valeur pour ce rôle (par exemple : `(AT-LEAST 2 locomotion)` dans `ETUDIANT-SPORTIF` ou `(AT-LEAST 1 locomotion)` dans l'assertion sur `laurent`). La requête prend donc en compte la description structurale donnée dans la définition de concepts et dans les assertions sur les individus.

Les requêtes sont généralisées aux sous-expressions de concepts : le marqueur '`?:`' indique la sous-expression dont on cherche les instances la satisfaisant.

```
ask-description[(AND ETUDIANT-SPORTIF
                     (ALL locomotion
                     ?:(ALL fabricant (ONE-OF Gazelle))))]
```

retourne tous les moyens de locomotion des étudiants sportifs dont le fabricant est Gazelle. La requête peut ainsi porter sur des objets en valeur d'un rôle, sans introduire de variables intermédiaires, ni de rôles inverses.

```
ask-description[(AND ETUDIANT-SPORTIF (ALL etat-sante ?:THING))]
```

Cette requête retourne les états de santé d'un étudiant sportif, indépendamment des exemples connus. Des inférences par des règles ou la classification peuvent aider à trouver des réponses à la requête.

Le traitement d'une requête se fait de la façon suivante : le concept représentant la requête est "classifié" parmi les concepts de la base. Ensuite, les instances des concepts parents sont testées pour déterminer leur appartenance au concept représentant la requête. D'une part, cela évite de tester les instances des concepts subsumés par la requête, puisqu'elles la satisfont par définition. D'autre part on ne teste que les instances des parents immédiats.

Le traitement de la requête comme un concept, dont la place dans la hiérarchie est ensuite déterminée par le classificateur, est pratiqué par plusieurs systèmes d'information basés sur des langages de la famille KL-ONE [Pat&84] [Bec&89] [Dev&90].

YAFOOL

YAFOOL [Duc&86][Duc91] est un langage à base de frames (cf. 2.2.2) sans distinction classe/instance. Les objets sont définis par spécialisation suivant le lien *est-un* (multiple).

Le but du filtrage en YAFOOL est de trouver l'extension d'un filtre donné en intension, c.-à-d. de trouver les objets qui satisfont la fonction logique définissant le filtre [Cha88]. Contrairement aux langages qui distinguent classes et instances et où le filtrage renvoie des instances, le filtrage en YAFOOL s'applique indifféremment à tous les objets, puisqu'ils ont le même statut. Un objet filtré peut donc se situer à n'importe quel niveau de spécialisation dans le graphe.

Le filtrage fait une partition de l'univers des objets en trois groupes :

- les objets certains vérifiant le filtre
- les objets impossibles qui ne vérifient pas le filtre
- les objets possibles dont on ne peut affirmer ni l'un ni l'autre.

Les objets certains ont leurs (domaines de) valeurs inclus(es) dans les domaines spécifiés dans le filtre. Les objets possibles sont ceux dont l'intersection des domaines de valeurs avec les domaines du filtre est non-vide. Les objets impossibles sont ceux dont un domaine de valeurs a une intersection vide avec le domaine correspondant dans le filtre.

Le filtrage manipule donc les domaines des attributs. L'espace de filtrage d'un attribut est délimité par le sous-graphe de racine(s) le(s) maxima des objets qui possèdent cet attribut. Ces maxima sont connus immédiatement par la facette *liste-frame* de chaque attribut.

Le filtre est exprimé comme une liste de couples propriété-expression, sans variable. Les connecteurs logiques OU et NON permettent de composer les expressions de filtrage. Les filtres peuvent comporter plusieurs niveaux d'imbrication.

Considérons le graphe suivant, décrivant les personnes, étudiants, sportifs, moyens de transport et leurs fabricants (Fig. 22).

Avec les définitions d'objets suivantes :

```
(defmodele personne
  (locomotion (des . moyentransport)))

(defmodele etudiant (est-un personne)
  ...)

(defmodele sportif (est-un personne)
  (locomotion (des . bicyclette)))

(defmodele laurent (est-un etudiant sportif)
  (locomotion (value peugeot-sprint)))

...

(defmodele moyentransport
  (fabricant (un . fabricant)))

...


(defmodele peugeot-205 (est-un voiture)
  (fabricant (value peugeot)))

(defmodele peugeot-sprint (est-un bicyclette)
```

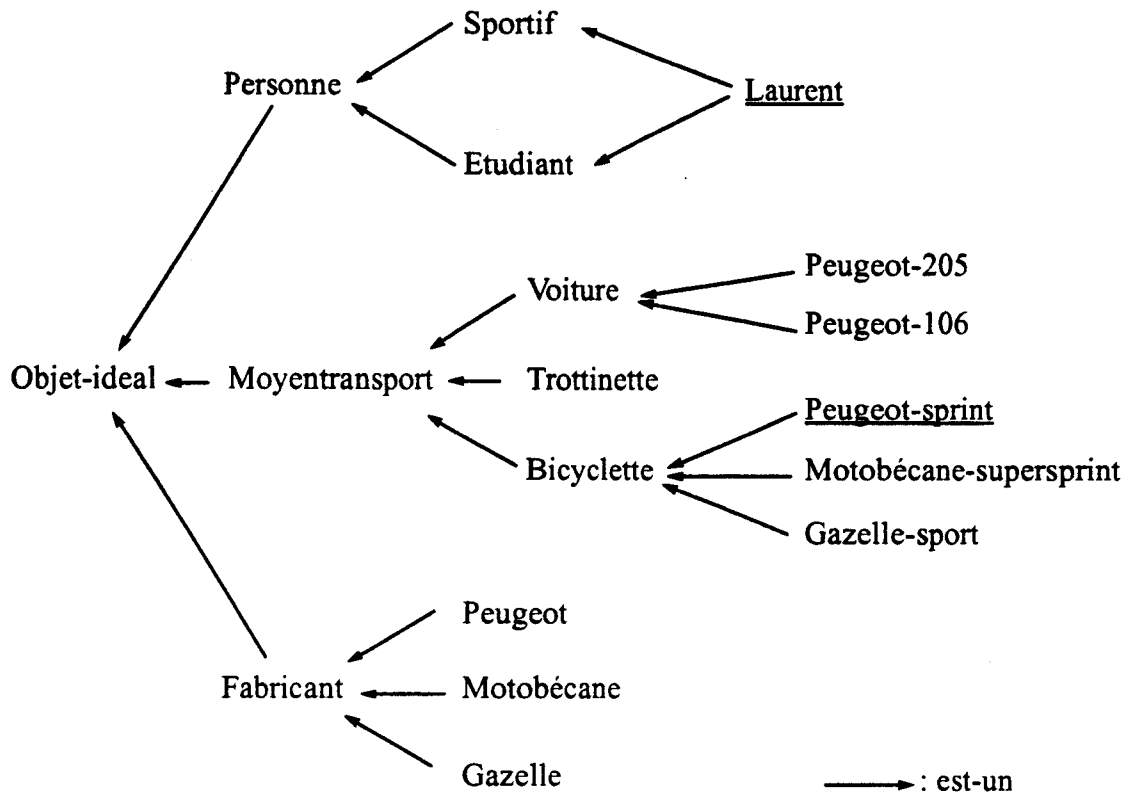


Fig. 22 Laurent et son vélo

```
  (fabricant (value peugeot)))
(defmodele motobécane-supersprint (est-un bicyclette)
  (fabricant (value motobécane)))
(defmodele gazelle-sport (est-un bicyclette)
  (fabricant (value gazelle)))
```

l'expression suivante va filtrer les objets ayant un attribut locomotion, dont la valeur est un objet fabriqué par peugeot :

```
(bfg ((locomotion (fabricant peugeot))
-->
  (|filtre2|
       (b-certain (value laurent))
       (b-possible (value personne))
       (b-prop (value (locomotion . |filtre3|)))))
  (|filtre3|
       (b-impossible (value motobécane-supersprint gazelle-sport))
       (b-certain (value peugeot-sprint peugeot-205 peugeot-106))
       (b-possible (value moyentransport))
       (b-prop (value (fabricant . peugeot))))
```

Le filtrage sélectionne tous les objets de l'espace de filtrage de l'attribut `locomotion`, c.-à-d. les objets de type `personne`, tels que la valeur de cet attribut vérifie `(fabricant peugeot)`. Le premier filtre sélectionne donc des personnes, et un deuxième filtre est chargé

de ramener les objets dont l'attribut `fabricant` a la valeur `peugeot`. L'espace de filtrage du deuxième filtre est `moyentransport`. Les objets impossibles pour le filtre imbriqué sont `motobécane-supersprint` et `gazelle-sport`. En effet, leurs fabricants respectifs sont `motobécane` et `gazelle` (intersection vide avec le domaine du filtre). Les objets certains sont tous les moyens de transport dont le fabricant est (exactement) `peugeot`, donc aussi quelques voitures. La propriété filtrée est donnée dans `b-prop`. L'étudiant sportif `laurent` est finalement le seul objet certain, puisqu'il possède (seulement) un `peugeot-sprint`, qui appartient aux objets certains du filtre imbriqué. L'objet `personne` est un objet possible, car le domaine de son attribut `locomotion` est un sur-type de celui du filtre (intersection non-vide).

La distinction entre objets certains, impossibles et possibles permet de fournir des informations plus fines que la simple liste d'objets (instances terminales) qui correspondent au filtre. Sans nécessairement donner une réponse précise, cette approche permet de cibler l'ensemble des solutions en donnant un objet générique, un peu comme les descriptions de concepts CLASSIC, qui donnent des informations en termes de conditions nécessaires (cf. page 62) à défaut de disposer des conditions suffisantes.

MERING I

MERING I définit un certain nombre de fonctionnelles (en VLisp) pour étendre les facilités de contrôle offertes par l'envoi de messages (possibilité d'envoi de messages répétitifs, conditionnels, à un ensemble d'objets destinataires,...). Parmi ces fonctionnelles figurent les structures de contrôle classiques IF, WHILE, AND, OR, et aussi des structures plus élaborées telles que FORALL, EXIST, TESTALL qui expriment les quantificateurs logiques "quel que soit" et "il existe" et qui servent directement dans les règles d'inférence. La syntaxe est la suivante :

```
(nom-fonctionnelle variable domaine expression).
```

Dans une application style base de données, des requêtes à la base peuvent facilement être formulées à l'aide des fonctionnelles :

```
(forall x (personnes allinst)
  (if  (x salaire > 8000)
       (x nom ?)))
```

La requête ci-dessus affiche le nom de toutes les instances de personnes dont le salaire est supérieur à 8000. Les requêtes peuvent être imbriquées et utiliser plusieurs variables :

```
: (forall X (revues allinst)
       (if  (exist Y (X journalistes ?)
            (if  (and (Y age < 35)
                      (Y articles n? | > 1))
                 X))))
= (ORDIN-INDIV MICRO-SYSTEMES MONDE-INFORMATIQUE)
```

En français : "Quelles sont les revues ayant au moins un journaliste agé de moins de 35 ans et qui a écrit plus d'un article ?" ('|' indique une cascade de transmissions, c.-à-d. un envoi de message au résultat du précédent).

Ces fonctionnelles sont utilisables directement dans les méthodes, et permettent de définir des méthodes de filtrage à l'aide des prédicats primitifs de comparaison d'entités (cf. 2.4.1.3). La méthode *match?* retourne tous les objets pris dans une liste qui, pour un sous-ensemble

d'attributs, sont compatibles avec l'objet receveur du message. Elle est définie avec la fonctionnelle *forall* et le prédicat de comparaison =?= :

```
match? methode: (lobj lattr) (forall X lobj (if (% =?= X lattr) X))
```

Plusieurs variantes de filtrage peuvent être définies en combinant les prédicats de comparaison de base et les fonctionnelles décrites ici.

2.4.2.3 Les facettes de filtrage

MERING

Les requêtes MERING sous la forme de fonctionnelles peuvent être évaluées directement. Par ailleurs, elles peuvent être placées dans n'importe quelle méthode MERING. En particulier, on peut les utiliser dans les facettes actives ou **réflexes** (qui sont les méthodes des attributs). Le réflexe *$si-req* est prévu spécialement pour calculer la ou les valeurs manquantes d'un attribut. A l'aide d'une fonctionnelle *forall* ou *exist* on peut donc spécifier un filtre pour calculer les valeurs de l'attribut possédant ce réflexe. On filtre les enfants d'une personne par l'attachement du réflexe `$si-req` de la façon suivante :

```
(type => $objet personne      ;définition du type personne
  enfants   $dom personne     ;l'attribut enfants de type personne
            $si-req   (forall X (personne allinst)
                                (if (X parent = %)
                                    X))  ;filtrage des personnes qui ont
                                         ;pour parent cette personne (%)
```

Le contenu du réflexe `$si-req` n'est pas obligatoirement une fonctionnelle de filtrage. La valeur d'un attribut peut être calculée par tout moyen (procédural), exprimé par une fonction Lisp, un envoi de message, une question à l'utilisateur, etc.

SHIRKA

En SHIRKA, deux facettes prédéfinies permettent d'exprimer des mécanismes d'obtention de valeurs d'attributs : `$sib-exec` et `$sib-filtre`. La facette `$sib-exec` ("si besoin exécuter") réalise un attachement procédural qui permet de calculer la ou les valeurs de l'attribut auquel elle est attachée. La facette `$sib-filtre` permet d'attacher un ou plusieurs filtres à un attribut. Un filtre porte le nom d'une classe et partage sa structure. Les conditions que doivent satisfaire les instances de cette classe pour être retenues comme valeur pour l'attribut, sont exprimées par des facettes de restriction de valeur sur les attributs du filtre.

Les attributs d'un filtre peuvent faire référence aux attributs de la classe dans laquelle il apparait. Des facettes spéciales de manipulation de variables (`$var-nom`, `$var->`, `$var<-`) ont été prévues pour gérer ces références. Ces facettes rendent possible l'imbrication de filtres sans mélanger les différents niveaux dans la cascade. Un exemple célèbre est celui du filtrage des petits-fils d'un homme [Rec&90] :

```
{ personne
  sorte-de       =            objet ;
  a-pour-pere    $un          personne ;
  pere-de        $liste-de    personne }

{ homme
```

```
sorte-de        =           personne ;
lui-meme        $var-nom    lui ;
a-pour-pere     $un         homme ;
petits-fils     $liste-de   homme
                $com        "les petits-fils d'un homme sont les "
                            "hommes qui ont pour pere les hommes [1]"
                            "qui ont pour pere l'homme dont il "
                            "est question"
                $sib-filtre
                     { homme
                       lui-meme      $var->      petits-fils ;
                       a-pour-pere   $var-nom    pere
                                     $sib-filtre
                                        { homme
                                          lui-meme      $var->   pere ;
                                          a-pour-pere   $var<-   lui }}}

{ Pierre
  est-un        =    homme ;
  a-pour-pere   =    Paul }

{ Paul
  est-un        =    homme ;
  a-pour-pere   =    Jean }

{ Jacques
  est-un        =    homme ;
  a-pour-pere   =    Paul }

{ Jean
  est-un        =    homme }
```

Les schémas `personne` et `homme` définissent la classe des personnes et la sous-classe des hommes. L'attribut `petits-fils` d'un homme est calculé en filtrant tous les hommes dont le père du père est cet homme (cf. la facette `$com` !). La facette `$var-nom` permet de créer une variable temporaire utilisée comme alias pour l'attribut auquel elle est associée, dans tout l'espace de définition du schéma. La variable temporaire `lui` associée à l'attribut `lui-meme`[2] du schéma `homme` permet ainsi de faire référence à l'instance d'`homme` pour lequel on filtre les petits-fils, dans le filtre (imbriqué) associé à son attribut `petits-fils`. De même, la variable temporaire `pere` associée à l'attribut `a-pour-pere` permet de faire référence au père d'un candidat petit-fils, lors du filtrage. Les facettes `$var->` et `$var<-` servent au transfert des valeurs calculées. Le filtre imbriqué parcourt toutes les instances de la classe `homme` et retient celles dont l'attribut `a-pour-pere` a comme valeur (la valeur de la variable) `lui`. La valeur de l'attribut `lui-meme` de chacune de ces instances est ensuite transférée dans l'attribut `a-pour-pere` du filtre englobant, par l'intermédiaire de la variable temporaire `pere`. Toutes les instances d' `homme` qui satisfont ce filtre englobant fournissent la valeur de leur attribut `lui-meme`, pour être transférée dans `petits-fils` : le résultat du filtre. La requête `val? Jean petits-fils` déclenche le filtrage et renvoie la liste des instances satisfaisant le filtre: `Pierre Jacques`. Du point de vue de l'implantation, un filtre est une sous-classe de la classe dont il porte le nom et spécialise celle-ci en spécialisant ses attributs, par l'ajout ou la spécialisation des facettes. Un filtre ne peut pas ajouter de nouveaux attributs.

1. On oublie ici les hommes qui ont pour *mère* les *femmes* qui ont pour père l'homme dont il est question.
2. L'attribut `lui-meme` est un attribut particulier qui a pour valeur l'instance dont il est l'attribut.

2.4.3 Le filtrage dans les règles d'inférence

De nombreux systèmes hybrides comportent un module de règles de production. Citons SMECI [ILO91], KOOL [Bul90], SHIRKA/CRIKA [Rec&85], MERING [Fer83], LORE/ ELOISE [Por&89], KEE [Fik&85], LOOPS [Bob&83], SMALLTALK/NEOPUS [Pac92], OBJLOG[Dug88], YAFOOL/YAFLOG [Duc91], ART[Inf85], CENTAUR[Aik83].

Les systèmes à base de règles généraux utilisent du filtrage pour déterminer les règles applicables en fonction des faits présents dans la base de faits. Dans les systèmes à base d'objets et règles, les règles sont exprimées sur les objets. Une règle filtre tous les objets dont l'état (valeurs des attributs, ou réponses à des messages) vérifie les conditions exprimées dans sa partie prémisse et/ou sa partie déclaration. Dans ce paragraphe, nous passons en revue le filtrage appliqué dans quelques systèmes à objets et règles.[1]

Dans KEE, le langage TellAndAsk utilisé dans l'expression des requêtes de filtrage est également employé pour l'expression des règles d'inférence. Le module de règles YAFLOG est un Prolog écrit en YAFOOL.

ELOISE

Le module de règles défini pour (en) LORE s'appelle ELOISE [Ben&89] [Por&89]. La base de faits est constituée de tous les objets présents dans le système (classes, instances, attributs, méthodes, et ... règles). Les règles ont une partie déclaration de variables (*filter*), une partie condition et une partie action. Les conditions et actions sont exprimées à l'aide de *patterns*, sorte de primitives comparables aux prédicats de correspondance de MERING (cf. 2.4.1.3), destinés à comparer des valeurs d'attributs, tester l'appartenance d'un objet à une classe ou ensemble, et à affecter, modifier ou supprimer des valeurs d'attributs ou créer des objets (partie action). De plus, tout calcul de valeur de variable peut être fait par un envoi de message. A part les patterns de condition, il y a les conditions de liaison et les conditions d'itération. Ces dernières permettent de lier une variable aux éléments d'un ensemble calculé. Une distinction stricte est faite entre attributs ou relations monovalués et multivalués, qui ne répondent pas aux mêmes patterns (sémantique d'ensemble ou valeur unique).

SMECI

Le langage de définition de règles SMECI résulte d'un effort de rapprochement au langage naturel[2]. Chaque règle comporte une partie déclaration de variables avec restrictions, une partie condition et une partie conclusion. Les objets filtrés sont ceux de la base, c.-à-d. les instances de catégories (classes). Les règles elle-mêmes sont implantées sous forme d'objet, donc peuvent faire l'objet de filtrage (contrairement aux classes). Le langage d'expression des règles est puissant : avec variables et quantificateurs universel et existentiel.

1. On ne détaillera pas les autres aspects de ces systèmes, tels que le fonctionnement en chaînage avant et/ou arrière, le contrôle, la gestion de tâches, de mondes multiples, l'héritage etc.
2. L'utilisation d'un langage quasi-naturel n'empêche pas l'utilisation dans ces mêmes règles de directives d'application particulière pour la règle, influant directement sur le fonctionnement du moteur d'inférence (instancier la règle une fois, plusieurs fois, en parallèle, en séquence, etc.).

CRIKA

Dans le but de réunir les qualités de la représentation "centrée objet" et celles de l'inférence d'un système de production, Vignard et Rechenmann ont conçu **CRIKA** issu d'une intégration de Shirka [Rec&90] et du système à base de règles CRIQUET [Vig84].

CRIKA utilise les règles de CRIQUET, mais le langage d'expression de ces règles a été adapté pour permettre l'utilisation des filtres (cf. le filtrage en Shirka 2.4.2.3), et ainsi faire référence aux schémas Shirka. Les filtres sont donc utilisés en partie déclaration d'une règle, qui s'exprime selon le format suivant :

```
Il-existe <nombre> <type> <variable> tel que <filtre>
```

Selon le nombre *$un* ou *$tous*, la variable <variable> prendra comme valeur la première ou toutes les instances vérifiant le <filtre>. Le filtre est un schéma de filtre Shirka. La classe concernée par le filtrage est donnée par <type>. Une notation pointée permet l'accès aux attributs d'un schéma en valeur d'une variable. Exemple de saisie (interactive) d'une règle :

```
(système)        Donnez une condition
(utilisateur)    il-existe $un personne ?x tel que
(système)        Donnez le filtre
(utilisateur)    (filtre1
                      sorte_de:
                        $valeur personne ;
                      nom:
                        $valeur "laurent" ;)
                 sachant ?y = ?x.père
(système)        Donnez une condition
(utilisateur)    predicat : ?z = ?y.père
(système)        Donnez une condition
(utilisateur)    -
(système)        Donnez une action
(utilisateur)    fonction : afatt '?x 'grand-père '?z
(système)        Donnez une action
(utilisateur)    -
```

Une fois la règle saisie, le système affiche son interprétation de la règle. Les règles peuvent être déclenchées par une commande qui lance le chaînage avant. On peut demander une explication qui consiste à montrer la règle avec ses variables instanciées.

OBJLOG

Objlog [Dug88] propose des règles d'inférence assez originales. Ce sont en fait des règles de classification. Une règle est modélisée comme une classe, qui contient deux ensembles de classes dans ses attributs *condition* et *conclusion*. Le premier ensemble correspond aux conditions et le deuxième aux conclusions. Les instances filtrées par la partie condition de la règle sont celles qui peuvent appartenir (ou qui appartiennent) aux classes de la condition. La conclusion de la règle consiste à ajouter des contraintes aux instances filtrées. Ces contraintes sont définies dans les classes de la partie conclusion et l'ajout des contraintes revient à classifier les instances filtrées dans ces classes (en modifiant les liens d'appartenance).

Il ne s'agit pas ici d'implanter un mécanisme de classification à l'aide de règles d'inférence, mais de l'inverse : l'inférence exprimée dans les règles fonctionne par

classification.

Pourtant, ce mécanisme ne semble pas vraiment adapté à une généralisation. Une règle permet bien de filtrer un n-uplet d'instances (pouvant appartenir à des classes différentes), mais les tests portant sur les champs de plusieurs instances doivent faire intervenir des procédures attachées à la facette *contrainte*. L'absence de variables rend difficile l'expression des liens entre la condition et la conclusion. La liaison de variables de la condition et de la conclusion telle qu'elle se fait dans les représentations classiques doit passer par la facette *contrainte* du champ conclusion de la règle. Une règle du type :

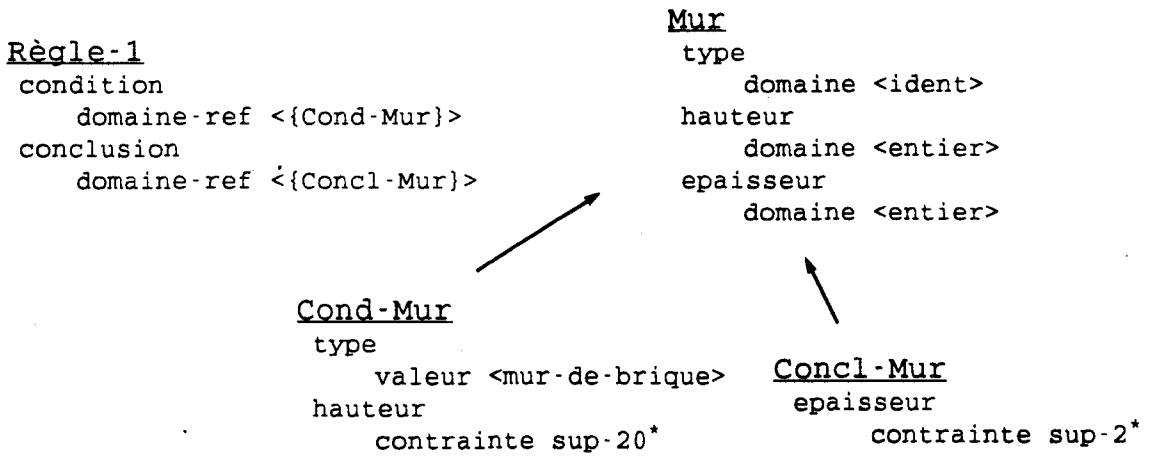
```
Si Y est-pere-de X et Z est-pere-de Y Alors Z est-grandpere-de X
```

semble difficile à exprimer en Objlog.

Exemple de règle en Objlog :

```
Si Mur(x) ∧ x.type = mur-de-brique ∧ x.hauteur > 20
Alors x.epaisseur > 2
```

Elle est représentée par la classe `Règle-1`, qui a deux champs : `condition` et `conclusion` (cf. Fig. 23). Le champ `condition` contient l'ensemble des classes condition, ici réduit à un seul élément : `Cond-Mur`. Le champ `conclusion` contient l'ensemble des classes conclusion, ici `Concl-Mur`. Les instances testées lors de l'exécution de la règle doivent pouvoir appartenir à la classe `Cond-Mur` pour être ensuite attachées à la classe `Concl-Mur`. Elles ne sont attachées à la classe condition que pendant le filtrage.



*) sup-20 et sup-2 sont des prédicats (Prolog) à vérifier par les valeurs des champs hauteur et epaisseur des instances filtrées.

Fig. 23 Représentation d'une règle en Objlog

MERING

MERING I propose une intégration de règles de production dans le sens où il s'agit d'une augmentation du langage et non pas d'une mise en relation de deux formalismes jugés

incompatibles (formalisme orienté objet et formalisme relationnel des systèmes de production).

Une règle est une entité MERING, issue d'un type *règle*, et manipulée par des envois de messages. Le moteur d'inférence classique est absent : une règle assure sa propre interprétation. Les règles sont "compilées" sous forme de transmissions MERING et attachées aux attributs. Lors du déclenchement d'une règle, par réflexe, les arguments de la règle sont passés à sa méthode d'exécution, qui a un comportement parfaitement procédural. Il n'y a donc pas de filtrage à proprement parler.

Les règles peuvent servir à propager le résultat d'une affectation d'attribut, et sont alors attachées dans la facette *$regle-ajout*, qui simule un comportement de chaînage avant, ou bien à calculer une valeur d'attribut, auquel cas elles sont attachées dans la facette *$regle-req*, qui simule un comportement de chaînage arrière.

Les règles MERING sont construites sur le schéma simple : Si <condition> Alors <action>. Elles manipulent deux types de variables : les arguments, qui reçoivent leurs valeurs lors du déclenchement de la règle (à l'appel, comme les arguments d'une méthode) et les variables proprement dites qui, elles, sont filtrées lors de l'exécution de la règle. Les variables sont typeés et quantifiées. Exemple de règle MERING calculant les petits-fils d'une personne :

```
(regle => petits-fils
  args      ::   X,
  soit      ::   (Y Z),
  dom       ::   (homme personne)
  quant     ::   (U U)
  condit    ::   (and
                   (or  (Z mere = X)
                        (Z pere = X))
                   (or  (Y pere = Z)
                        (Y mere = Z)))
  action    ::   (X petits-fils <= Y)
  ; lorsque l'on trouve un petit-fils, on le place
  ; sur l'attribut petits-fils de X
```

Cette règle peut ensuite être placée sur l'attribut `petits-fils` de la classe `personne`, dans la facette `$regle-req`, et sur les attributs `pere` et `mere` dans la facette `$regle-ajout` :

```
(type => personne
  pere        $dom          personne
              $regle-ajout  petits-fils
  mere        $dom          personne
              $regle-ajout  petits-fils
  petits-fils $dom          personne
              $regle-req    petits-fils)
```

Lors des accès à ces attributs, les règles sont déclenchées de la façon suivante :

```
(forall X (% # fac? '$regle-ajout)
   (X exec: %))
;pour chaque règle dans la facette $regle-ajout (ou $règle-req)
;de cet attribut (#) de cet objet (%) exécuter cette règle avec
;comme argument self (%)
```

NEOPUS

Le système NéOpus [Pac92] intègre les règles d'OPS-5 dans l'univers Smalltalk. Il modifie et étend Opus [Lau&87]. Les règles s'expriment sur tous les objets Smalltalk, y compris les classes, et peuvent utiliser toute expression Smalltalk (envoi de messages), dans les prémisses comme dans les parties conclusion.

2.4.4 La classification

Le raisonnement par classification existe dans de nombreux domaines et le terme même de classification a plusieurs acceptions [Hat&91]. En analyse de données la classification automatique consiste à élaborer un ensemble de classes pour regrouper des objets en fonction de leurs similarités. L'ensemble de ces classes obtenues peut également être qualifié de classification. La classification n'est pas nécesairement organisée de façon hiérarchique. En RPO, il y a deux acceptions de la notion de classification. Premièrement, la classification des classes consiste à organiser et maintenir une hiérarchie de classes, en insérant de nouvelles classes à leur place. Cette acception est aussi appelée catégorisation et est une technique d'acquisition des connaissances. La deuxième, la classification d'instances, consiste à trouver la classe d'appartenance d'un objet individuel (sa catégorie). Elle est aussi appelée classement ou identification. La classification d'instances par rapport à une hiérarchie de classes ou taxonomie sous-tend des raisonnements de diagnostic ou d'identification.

La partie II de ce mémoire est entièrement consacrée à la classification.

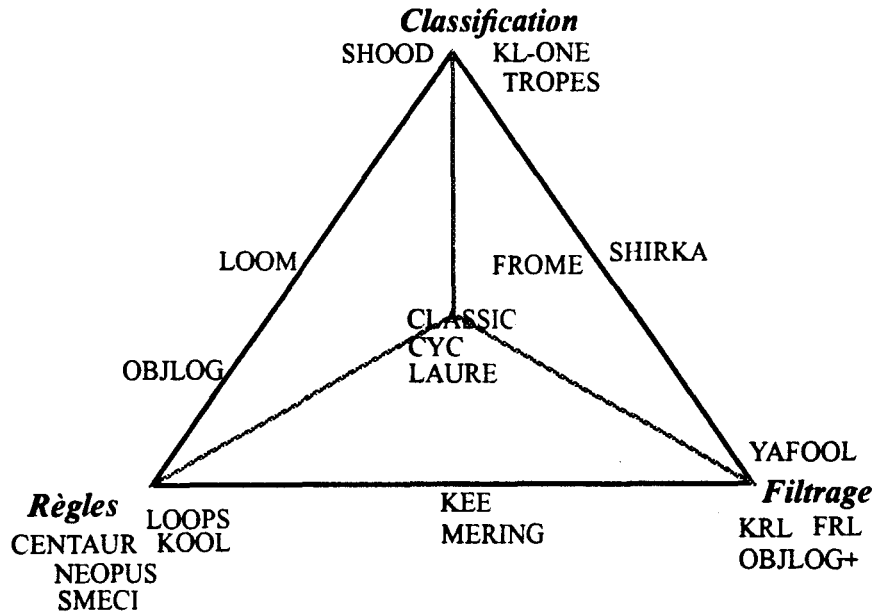


Fig. 24 Les formes de raisonnement

2.5 "Méta" et réflexivité

"Méta" et réflexivité sont deux notions étroitement liées qui se sont montrées puissantes d'abord dans les paradigmes procéduraux, logiques et à base de règles [Pit90] avant de faire leur apparition dans le paradigme orienté objet. Pattie Maes a réalisé une étude de ces notions dans le cadre de divers systèmes non-orientés objet et une expérimentation de la réflexivité dans un langage orienté objet (3-KRS), qui sont décrites dans sa thèse [Mae87a]. Reprenons brièvement quelques-unes de ses définitions :

- un système computationnel
  raisonne et agit sur une partie du monde, appelée son domaine.
- connexion causale :
  un système et son domaine sont liés de façon à ce qu'un changement dans l'un entraîne un effet dans l'autre.
- méta-système :
  système computationnel dont le domaine (son système-objet) est un autre système computationnel.
- système réflexif :
  un méta-système connecté causalement qui a pour système-objet lui-même.

[Mae87a] distingue les systèmes *réflexifs* des systèmes *à architecture réflexive*, les premiers ayant seulement un ensemble fixe de facilités méta, par opposition aux seconds, qui sont de véritables systèmes ouverts ("open-ended") et peuvent être étendus à volonté.

Tout système computationnel est constitué de données et d'un programme, où les données représentent le domaine et le programme explicite comment manipuler ce domaine. Les données d'un système réflexif constituent d'une part la représentation d'une partie du monde et d'autre part l'auto-représentation du système. Son programme est la computation sur le domaine et sur le système lui-même (computation réflexive).

L'un des principaux intérêts de la réflexivité pour un système est qu'elle lui fournit une grande capacité d'évolution et d'adaptation à des contextes variés d'application [Fer89].

On peut considérer le méta comme un niveau descriptif supérieur. La réflexivité est alors l'application circulaire du méta. Méta et réflexivité concernent la description et la manipulation des informations structurales (réflexivité structurale) et comportementales (réflexivité computationnelle ou opératoire) de leur objet.

2.5.1 Les méta-niveaux de description

Dans les systèmes à base de frames, la description des attributs au travers des informations contenues dans les facettes (documentation, historique, relations, valeurs par défaut, moyens de calcul, procédures) constitue déjà un niveau méta [Mae87a].

Dans l'approche classe/instance, un premier niveau méta est celui de la classe, qui détient la structure et le comportement de ses instances et qui sert de moule à leur création. Conformément à un souci d'uniformité (les classes sont des objets), on a vu apparaître un niveau méta supplémentaire pour les classes en tant qu'objets par l'introduction de métaclasses en Smalltalk et LOOPS.

Smalltalk-80 associe à chaque nouvelle classe (et de manière automatique) sa métaclasse.

Ces métaclasses sont elles-mêmes instances de la classe *Metaclass*. La définition est rendue circulaire au niveau de *Metaclass*, instance de sa propre métaclasse *Metaclass class*, qui est elle-même une métaclasse, donc instance de *Metaclass*.

Cointe [Coi87] a évoqué les limites de ce modèle méta-circulaire : la création des métaclasses est automatique et l'utilisateur ne peut pas les créer explicitement par envoi de message *new*, comme les autres objets. Chaque métaclasse n'a qu'une seule instance et le nombre de niveaux méta est limité.

Dans LOOPS, la création d'une (méta)classe se fait explicitement par le message *new*. Mais toute métaclasse est obligatoirement instance de *Metaclass*, donc le nombre de niveaux méta est également limité. Dans LOOPS comme dans Smalltalk-80, les 3 niveaux métaclasse, classe et instance sont séparés.

2.5.2 Méta-circularité

Dans un effort de minimalisation et d'uniformisation plus grande, Cointe et Briot ont défini ObjVlisp [Coi87], noyau méta-circulaire minimal pour tout langage à base de classes/ instances. Ce modèle considère toute classe comme un moule pouvant créer des instances. La classe en tant que moule détient deux fonctionnalités :

- la fonctionnalité de description (abstraction)
- la fonctionnalité de création d'instance

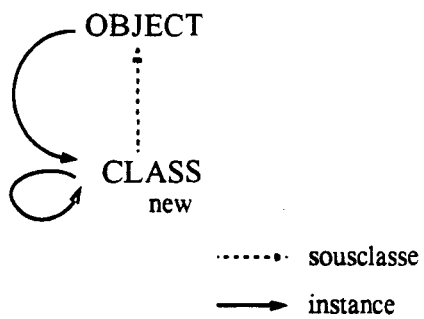


Fig. 25 Noyau ObjVlisp

Ces deux fonctionnalités sont associées dans un modèle tel qu'ObjVlisp. On peut dissocier ces fonctionnalités fondamentales d'une classe. On obtient ainsi deux catégories de classes :

- les classes abstraites détenant la fonctionnalité de description d'objets (abstraction).
- les classes instanciables détenant la fonctionnalité de génération d'objets

Cette distinction est faite dans le modèle ROME, où la classe R-CLASS est la méta-classe des classes abstraites ou classes de représentation. La classe I-CLASS est la méta-classe des classes instanciables et est une sous-classe de R-CLASS.

Le noyau devient[1] :

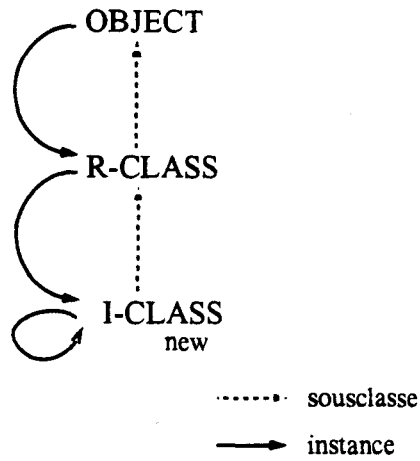


Fig. 26 Noyau Rome

Notons que la définition de classes abstraites peut être obtenue également en ObjVlisp par extension du noyau et par inhibition de la fonctionnalité d'instanciation pour ces classes, comme elle est décrite pour ClassTalk dans [Bri&89].

La définition méta-circulaire peut se faire de manière réflexive, par un bootstrap [Coi87]. Le noyau "s'auto-engendre" par l'application de ses propres méthodes qui le construisent selon sa propre définition de sa structure.

Cependant, le but de la définition réflexive n'est pas de modifier le noyau lui-même pour obtenir des comportements différents, mais plutôt de pouvoir l'étendre par la définition de nouvelles métaclasses à tout niveau. D'autres langages ont adopté des schémas métacirculaires extensibles, où chaque élément du langage est défini comme un objet (*réification*), afin de pouvoir accéder à sa description et la manipuler. C'est en particulier le cas pour CLOS [DeM&87][Moo89], dont la définition a été largement inspirée des expériences sur Smalltalk, LOOPS et ObjVlisp. Son Méta-Object-Protocol (MOP) fournit le niveau de description nécessaire à l'extension du langage [Kic&91][Pae93][Att&89][Rat91].

2.5.3 La réflexivité opératoire

Les travaux de Maes sur 3-KRS mettent en avant la réflexivité opératoire qui a pour but d'influer sur le comportement du système par l'intermédiaire du niveau méta. Chaque objet (concept) en 3-KRS possède (potentiellement) un méta-objet (et récursivement), qui garde des informations sur l'implantation et l'interprétation de son *référent* et qui gère son comportement. Changer de méta-objet, c'est changer de comportement. Par opposition aux classes de l'approche classe/instance, le méta-objet est privé. Le changement de méta-objet pour n'importe quel objet peut modifier temporairement le comportement du système pour des buts de debugging ou trace par exemple. C'est en ce sens que le système est pleinement réflexif.

1. Il s'agit du noyau méta-circulaire, avant extension aux objets pour la Représentation Multiple et Evolutive par la classe R-OBJECT, instance de R-CLASS et sous-classe de OBJECT.

D'autres travaux sur l'expérimentation de la réflexivité dans un environnement à base de prototypes incluent les travaux sur SELF. SELF possède un mécanisme d'introspection à travers des objets miroirs, qui détiennent des informations sur l'implantation de chaque objet [Age&92]. [Coi&92] décrit une tentative d'extension de SELF avec des méta-objets qui gèrent le comportement, pour étudier la réflexivité opératoire dans les langages à prototypes (cf. aussi [Mul&93]).

2.5.4 Séparation de la réflexivité structurale et de la réflexivité opératoire

MERING IV [Fer89] combine les méta-objets à la KRS avec un modèle classe/instance. Les différentes fonctionnalités réflexives sont prises en charge par des entités différentes. Ainsi, chaque objet a un méta-objet privé, qui s'occupe des communications (gestion des messages). Le méta-objet concerne donc la réflexivité opératoire. En ce qui concerne la description de la structure, elle est détenue par les *Smodèles*, qui s'occupent de la réflexivité structurale et qui permettent au méta-objet d'instancier l'objet.

La classe, contrairement au modèle classique, n'est plus le modèle structurel de l'instance, mais une abstraction intensionnelle d'un ensemble d'entités. Elle constitue le modèle conceptuel de l'instance. Elle ne sert donc plus de moule à l'instanciation, car elle ne contient pas la représentation des objets en mémoire. Elle est l'abstraction des propriétés communes d'un ensemble d'individus. Ainsi, les classes servent à l'abstraction, tandis que les méta-objets et les Smodèles servent à la représentation des objets. Le niveau méta associé à ce niveau conceptuel (les classes des classes) donne lieu aux "hyperclasses". Les hyperclasses décrivent les classes au niveau conceptuel. Chaque objet (classe, méta-objet, hyperclasse...) a son Smodèle qui détermine la structure, et la méthode de création d'instance.

2.6 Conclusion

Au travers l'étude de l'état de l'art, nous avons relevé plusieurs aspects qui nous paraissent primordiaux pour un système de représentation à objets souple. D'abord, des objets de type frame semblent fournir des structures riches, bien adaptées à la représentation. La multiplicité des représentations est une caractéristique dont aucun système de représentation ne devrait être privé, ainsi que la représentation dynamique.

Dans le chapitre suivant, nous décrivons notre proposition du système FROME. Pour ce système hybride, nous retenons une représentation à base de frames avec méthodes en adoptant l'approche classe/instance avec héritage multiple. La représentation multiple et évolutive avec points de vue de ROME constitue l'axe central du système. Nous définissons également un mécanisme de filtrage d'objets. Les capacités réflexives qui permettent l'extension du système sont obtenues par son implantation méta-programmée en ROME (chapitre 4).

3
La représentation en FROME

3.1 Objectifs

Nous présentons le système FROME. L'objectif est d'offrir une plate-forme d'expérimentation pour la représentation multi-point de vue des connaissances. Celle-ci allie les avantages des frames (typage des attributs, expression de contraintes, programmation par réflexes) à ceux des langages objets à base de classes (distinction classe/instance, définition possible de nouvelles méta-classes pour l'extension du langage, méthodes, envoi de message). FROME retient la notion de point de vue du langage ROME, pour la description et la manipulation multi-point de vue de concepts. La notion de point de vue est utilisée avantageusement pour résoudre des types de conflits survenant dans les frames.

La représentation évolutive des frames répond à la nécessité de souplesse d'un système de représentation, pour adapter la description des instances par généralisation ou affinement. Cette capacité d'évolution est exploitée par le mécanisme de filtrage d'instances, qui sélectionne des objets pour les faire évoluer.

3.2 Classes et instances de frames

FROME est un langage de frames fondé sur la distinction classe/instance. Les classes de frames décrivent la structure et le comportement de leurs instances. Les classes sont reliées entre elles par la relation superclasse, et forment ainsi un graphe (orienté sans circuit), sur lequel opère un mécanisme d'héritage multiple. Un frame particulier est une instance terminale d'une classe de frames.

3.2.1 Structure

La structure d'un frame est décrite par sa classe comme un ensemble de *slots*, chaque slot étant défini par un ensemble de *facettes*.

Comme nous avons vu en 2.1, les facettes permettent de typer les slots des frames et d'y associer des moyens de calcul, des valeurs par défaut, et aussi de provoquer des effets de bords lors des accès. La description d'un slot dans une classe de frames consiste à lui donner un nom et un certain nombre de facettes. Les facettes prédéfinies en FROME sont données dans le schéma Fig. 27 (page 80).

Les facettes peuvent être classifiées suivant plusieurs critères : nous distinguons d'une part, les facettes actives (procédurales) et les facettes passives (déclaratives, à valeur de constantes), d'autre part les facettes contraignantes et non-contraignantes. Les facettes contraignantes expriment une contrainte sur la valeur admissible du slot.

Quelques facettes passives ne sont pas concernées par la distinction contraignant/ non-contraignant. Ce sont des facettes telles que $frame, $name,... qui sont initialisées et

manipulées par le système.

Une distinction supplémentaire peut être faite pour les facettes passives entre facettes de classe et facettes d'instance. En effet, la plupart des facettes sont des facettes de classe. Leurs valeurs concernent toutes les instances de la classe. La valeur d'une facette d'instance est propre à une instance de frame particulière.

Les détails concernant les diverses facettes sont décrits dans [Dek89][Car&91a].

Facettes	actives	passives	classe	instance	système	contraignantes	non-contraignantes
$type		✗	✗			✗	
$supercl		✗	✗			✗	
$domain		✗	✗			✗	
$interval		✗	✗			✗	
$default		✗	✗				✗
$com		✗	✗				✗
$frame		✗	✗		✗		
$val		✗		✗			✗
$name		✗		✗	✗		
$frame		✗		✗	✗		
$verify	✗		✗			✗	
$if-needed	✗		✗				✗
$before-X*	✗		✗				✗
$after-X*	✗		✗				✗

*) avec X = get, put, rem (avant-, et après-, lecture, écriture, destruction de valeur)

Fig. 27: Les différents types de facettes en FROME

Exemple de description de frames :

Une description générale d'appartements est donnée par la classe `Appartement`. Quelques slots associés sont `proprietaire`, de type `Personne` (décrit par une autre classe de frames), `etage`, un entier, `loyerTCC`, qui est le loyer plus les charges. Le calcul du `loyerTCC` et des `charges` se fait à l'aide des facettes `$if-needed`. La définition d'une classe en FROME se fait par un envoi de message de création à sa métaclasse. Voici le message de création pour la classe `Appartement`:

```
[Meta-i-frame 'new 'Appartement 'supercl '(Frame)
 'slots '( proprietaire
                ($type Personne)
           etage ($type integer
                  $interval (0 20))
           jardin ($type boolean)
           loyer ($type real
```

```
            $interval (500 1000))
      charges
            ($type real
             $if-needed (lambda()
               (+ %entretien
                  (* 0.05 %loyer))))
      entretien
            ($type real
             $default 50)
      loyerTCC
            ($type real
             $if-needed (lambda()
               (+ %loyer %charges))))]
```

L'appartement `mon-appart` est instancié dans la classe Appartement :

```
[Appartement 'new 'mon-appart
 'etage  3
 'jardin false
 'loyer  800]
```

3.2.2 Comportement

Le comportement des frames en FROME est assuré, d'une part, par les méthodes et d'autre part, par les réflexes. Les méthodes peuvent être privées ou publiques, suivant qu'il y a ou non un sélecteur associé, faisant partie de l'interface du frame. Les réflexes sont comparables à des méthodes privées, dans le sens où l'on ne peut les déclencher directement. Elles sont déclenchées lors des accès aux slots.

Les accès aux slots se font par un envoi de message au frame, qui déclenche la méthode d'accès adéquate.

Deux styles de programmation sont donc possibles en tirant profit des différents types de comportement des frames : la programmation orientée objet, par envoi de messages entre frames, et la programmation par réflexes, utilisant la propagation par effets de bord lors des accès aux slots. Cette dernière approche est aussi appelée "programmation orientée par les accès" [Ste&86]. Les deux approches peuvent être combinées. Envoi de messages dans un réflexe, accès aux slots d'un frame à l'intérieur d'une méthode, déclenchant les réflexes...

Les réflexes comme les méthodes s'expriment par des lambda-expressions lisp.

3.2.3 Héritage et affinement

L'héritage (multiple) est un héritage entre classes de frames, et concerne les slots et les méthodes. Une sous-classe hérite de tous les slots et méthodes de ses superclasses. La description d'un slot dans une classe de frames consiste à lui donner un nom et un certain nombre de facettes. Cette description est entièrement héritée par les sous-classes.

L'affinement dans une approche classe/instance est réalisé par un sous-classement. La sous-classe est une spécialisation du modèle dont elle hérite (ses superclasses), ajoutant de nouveaux attributs et méthodes et redéfinissant des méthodes héritées. En FROME, où les classes de frames définissent slots et méthodes pour leurs instances, la spécialisation concerne aussi bien l'ajout et la redéfinition des slots que l'ajout et la spécialisation des méthodes.

L'affinement des slots est gouverné par des règles précises qui prennent en compte les différents types de facettes. Les facettes contraignantes expriment une contrainte sur la valeur admissible du slot. L'affinement d'un slot entraîne alors l'ajout de nouvelles contraintes par la restriction du domaine des valeurs admises, ou l'ajout ou la redéfinition de facettes non contraignantes, qui peuvent réaliser l'ajout ou l'extension d'effets de bords lors des accès. L'affinement des réflexes est réalisé comme celui des méthodes : un réflexe redéfini dans une classe masque le réflexe de même nom de ses superclasses. La définition peut néanmoins faire appel aux définitions masquées à l'aide de l'application de *super*, ce qui permet d'enrichir les réflexes au fur et à mesure, selon l'affinement. L'affinement des contraintes se fait en respect des règles d'affinement généralement reconnues : (cf. [Dug88] et [Car&91a])

Le $type d'un slot affiné doit être un sous-type du $type hérité.

Les valeurs des facettes $domain et $interval doivent être incluses dans les valeurs héritées des superclasses.

L'exemple de l'agence de location immobilière gérant des appartements nous permet de montrer la spécialisation par sous-classement [Dek&92]. Des appartements spécifiques sont ceux qui disposent d'un jardin, ou ceux situés aux étages supérieurs, disposant d'un ascenseur. Ces appartements sont décrits par les sous-classes d'`Appartement`, `AppartAvecJardin` et `AppartEtageSup` :

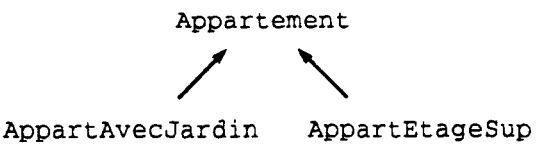


Fig. 28: Sous-classement

```
[Meta-i-frame 'new 'AppartAvecJardin 'superclasses '(Appartement)
 'slots '(etage ($domain (0))
         jardin ($domain (true))
         loyer  ($interval (700 1000)))]

[Meta-i-frame 'new 'AppartEtageSup 'superclasses '(Appartement)
 'slots '(etage ($interval (1 20))
         jardin ($domain (false))
         fraisAscenseur
                ($type real
                 $if-needed (lambda()
                   (* %etage 12.50)))
         charges
                ($if-needed (lambda()
                   (+ (super)
                     %fraisAscenseur)))))]
```

La spécialisation est exprimée à l'aide des diverses facettes. Un appartement avec jardin restreint le domaine de son slot `etage` à 0, en ajoutant la facette `$domain`. La facette `$domain` est restreinte dans les deux sous-classes. Le loyer d'un appartement avec jardin s'élève à 700 au minimum (facette `$interval`). Un appartement à l'étage a des charges supplémentaires, dues aux frais d'ascenseur. La façon de calculer ces charges est spécifiée dans la facette `$if-needed`. Il s'agit d'une facette affinée : `super` appelle la facette `$if-needed` du même slot définie dans la classe `Appartement`.

3.3 La représentation de frames selon plusieurs points de vue

En 2.3, nous avons exposé le besoin d'exprimer des représentations multiples, reflétant plusieurs points de vue sur la connaissance décrite. Différentes solutions ont été discutées. En FROME, nous avons retenu la notion de point de vue fondée sur la représentation multiple du langage ROME [Car89]. La manière dont cette notion est héritée et étendue pour les frames est exposée dans le chapitre 4, qui a pour objet l'implantation de FROME en ROME.

En ROME, la sélection d'un point de vue permet de lever des conflits entre caractéristiques (attributs et méthodes) qui portent le même nom, mais qui sont issues de sous-classifications différentes. Ces conflits surgissent tant en héritage multiple qu'en représentation multiple. Contrairement aux techniques de linéarisation, qui privilégient une seule des caractéristiques en conflit, les expressions de points de vue permettent d'hériter toutes les caractéristiques. La résolution des conflits n'est pas effectuée de façon statique, en faisant des choix arbitraires (selon l'ordre des superclasses par exemple), mais se fait de façon dynamique en prenant en compte le contexte induit par le choix d'un point de vue.

3.3.1 Principes de multiplicité pour les slots

Dans le cadre des frames, de nouveaux types de conflits sont distingués. Il s'agit du conflit de nom entre slots distincts et du conflit de définitions multiples d'un même slot, ces définitions entraînant des interprétations multiples du slot suivant les points de vue.

Réécrivons les principes de l'héritage multiple de ROME [Car89], appliqués aux slots. Ces principes sont illustrés sur la Fig. 29.

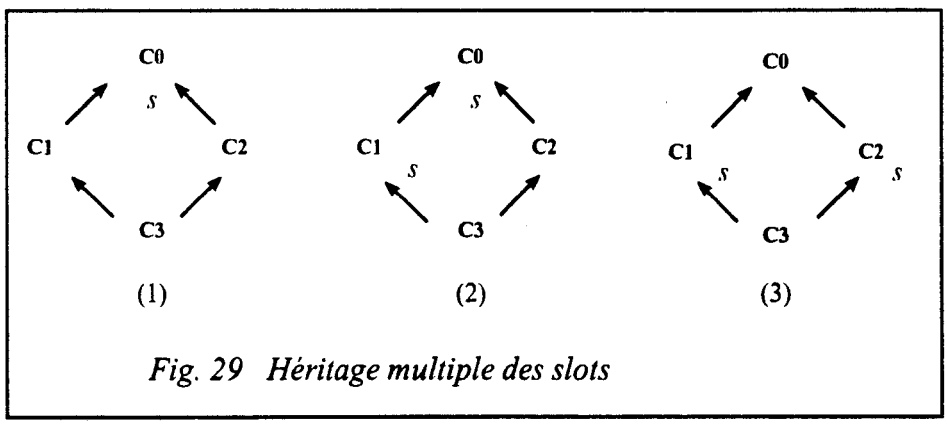


Fig. 29 Héritage multiple des slots

(1) Les slots qui sont accessibles suivant plusieurs chemins d'héritage sont hérités **une seule fois** (pas d'héritage répété). Les instances de C3 ont une seule occurrence de *s*.

(2) Une classe qui redéfinit un slot affine le slot hérité. Les instances de C3 ont un seul slot *s*, celui de C1. C'est le **principe d'affinement**.

(3) Le **principe d'indépendance** : il n'y a pas de conflit d'héritage entre slots homonymes hérités de classes indépendantes. Les instances de C3 ont deux slots *s*. La notion de **point de vue** intervient pour exprimer la préférence pour les slots d'une sous-classification par rapport à une autre.

Le cas (3) est le cas du "conflit de noms" entre slots distincts.

Un nouveau cas est à signaler : le cas du "conflit de définitions" d'un même slot, par extension du cas (2) :

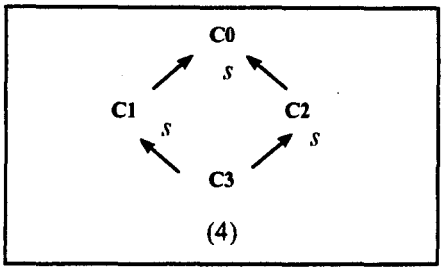


Fig. 30 Affinements multiples d'un slot

(4) Affinement multiple d'un même slot. Par (1), les instances de C3 ont un seul slot *s*. Ce cas est régi par le **principe de multiplicité des contraintes** et par la sélection de point de vue pour choisir un affinement (non-contraignant) particulier.

principe de multiplicité des contraintes :

> Si plusieurs classes spécialisent le même slot <s>, en ajoutant de nouvelles contraintes correspondant à leur propre point de vue, <s> doit satisfaire toutes ces contraintes.

Les paragraphes 3.3.2, 3.3.3, et 3.3.4 étudient les conflits sur les slots et montrent comment les expressions de point de vue servent à les résoudre. L'exemple de l'agence de location immobilière illustrera ces paragraphes ([Dek&92]).

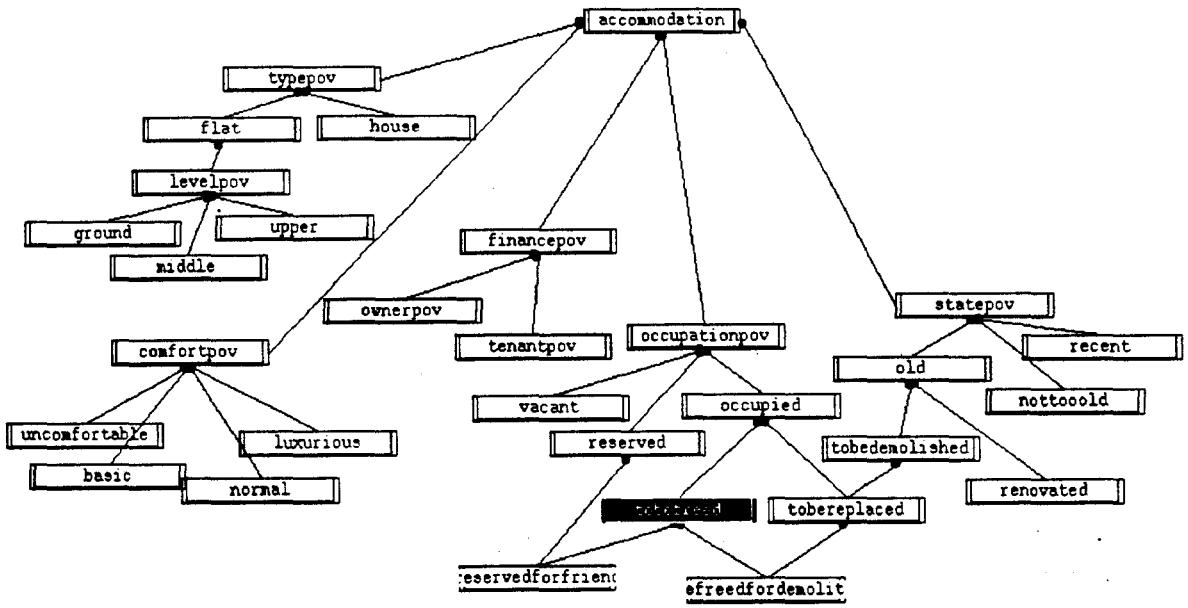


Fig. 31: La classification multi-points de vue des logements, réalisée avec l'environnement graphique GEROME [Leb&92].

Les logements gérés par l'agence sont décrits selon plusieurs points de vue, induisant chacun une sous-classification particulière. Nous pouvons distinguer le point de vue de l'occupation, décrivant les caractéristiques relatives à l'occupation du logement, c. à d. le locataire actuel, la date d'arrivée de ce locataire, une liste des anciens locataires. Sous ce point de vue, les logements sont classifiés comme occupés, vacants ou réservés. Parmi les autres points de vue possibles, citons celui du Type de logement, qui sous-classifie les appartements et les maisons, celui du Secteur, et celui des Finances, regroupant des caractéristiques telles que les taxes, les frais de réparations, etc. Des sous-points de vue peuvent induire des sous-classifications à l'intérieur d'un point de vue. Suivant l'étage par exemple, les appartements peuvent être partitionnés en appartements situés aux étages supérieurs, moyens ou au rez-de-chaussée.

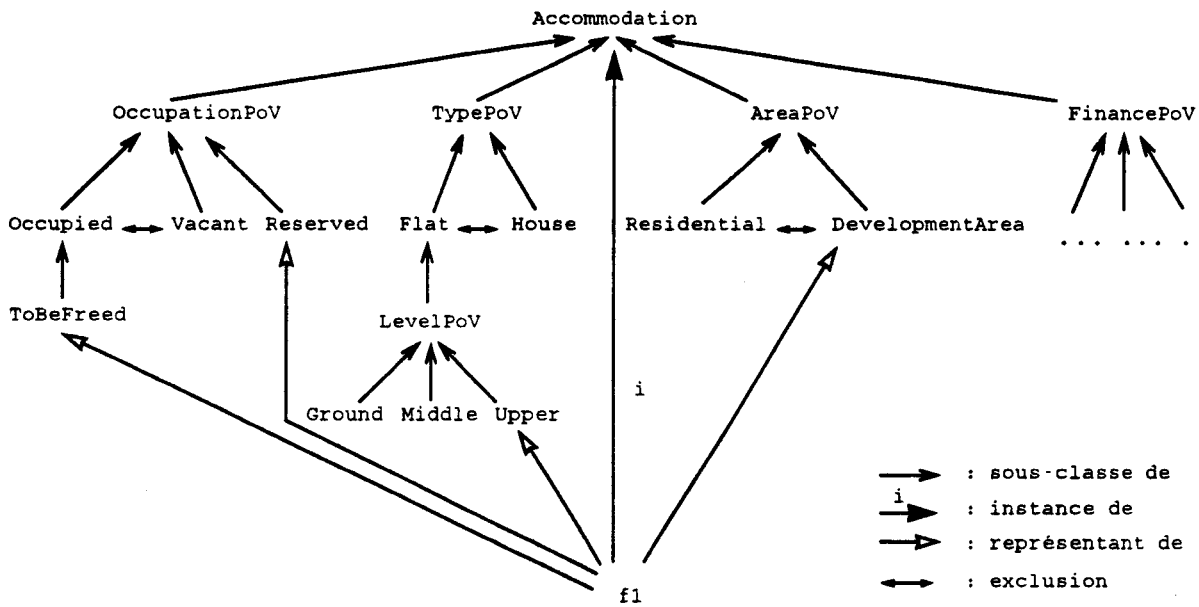


Fig. 32 Représentation multiple

Un logement particulier est par exemple `f1`, un appartement au 16ème étage d'un immeuble HLM, qui va être libéré le 1er février par son locataire actuel et qui est réservé pour un futur locataire. Cet appartement est décrit en tant que logement en instanciant la classe `Accommodation` (cf. Fig. 32). Les différentes caractéristiques énoncées ci-dessus sont décrites dans les spécialisations de la classe `Accommodation` sous ses différents points de vue. Du point de vue du type de logement, `f1` est un appartement, situé aux étages supérieurs. Du point de vue de l'occupation, il est décrit par les classes `ToBeFreed` et `Reserved`. Du point de vue du secteur (`AreaPoV`) il est décrit par la classe `DevelopmentArea`. La réprésentation multiple est exprimée à travers les liens de représentation qui attachent l'instance à ses classes sous les différents points de vue.

La description de l'objet est essentiellement partielle. L'objet n'est pas nécessairement rattaché à tous les points de vue représentés dans le graphe. L'appartement `f1` n'est pas représenté selon le point de vue financier dans l'exemple ci-dessus.

La représentation multiple est contrôlée par des contraintes de cohérence entre classes de

représentation. Quand deux classes Ci et Cj s'excluent mutuellement, un objet ne peut être représentant des deux simultanément. Ces contraintes sont exprimées dans des **relations d'exclusion** :

```
[Ci 'excl Cj]
```

A la Fig. 32, les classes `Flat` et `House` sont en relation d'exclusion, car un logement ne peut être à la fois un appartement et une maison. Les propriétés de la relation d'exclusion sont ([Dek89]) :

- symétrie
- héritabilité : si Ci exclut Cj, alors Ci exclut toutes les sous-classes de Cj.

Exemple : `Occupied` exclut `Vacant`, alors `Vacant` exclut `Occupied` (par symétrie) et aussi `ToBeFreed` (par héritage).

3.3.2 Conflit de nom entre slots distincts

La représentation multiple d'un frame autorise sa manipulation tantôt sous un point de vue, tantôt sous un autre, suivant l'intérêt que l'on porte à l'objet représenté à un moment donné.

Ainsi, l'agent de location chargé de l'attribution des logements s'intéresse à l'aspect occupation d'un logement, au moment où il doit contacter le locataire occupant et le futur locataire pour arranger une visite. La personne qui s'occupe de la comptabilité s'intéresse avant tout au point de vue financier. La sélection d'une sous-classification particulière correspond au choix d'un point de vue sur l'objet représenté par le frame.

Intuitivement, un point de vue correspond à une partie de la hiérarchie des classes, décrivant un ensemble de caractéristiques ayant trait à une partie spécifique du domaine de connaissances représentées. Plus radicalement, chaque classe FROME peut être interprétée comme un point de vue sur les frames qu'elle décrit. La définition de la notion de point de vue est celle de ROME (cf. 2.3.4.3) :

> Soit l'**ensemble de représentation** d'un frame f, l'ensemble des classes de représentation du frame, i.e. toutes les classes auxquelles est relié le frame par un lien de représentation ou un lien d'instanciation, et leurs super-classes. Le graphe de représentation est noté Rep(f).
>
> Soit la **dépendance** de C l'ensemble de toutes les super-classes et des sous-classes et C elle-même. La dépendance de C est notée Dep(C).
>
> Le **point de vue** d'une classe C sur un frame f est défini :
>
> $$PdV(C,f) = Rep(f) \cap Dep(C)$$

Par exemple (Fig. 32) :

PdV(Occupied, f1)= {ToBeFreed, Occupied, OccupationPoV, Accommodation}

La sélection d'un point de vue est exprimée par la spécification d'une classe lors de l'accès à un frame dans une **expression de point de vue** :

```
[<unFrame> '? '(<slot> as <classe>) ]
```

en lecture, et

```
[<unFrame> '<- '(<slot> as <classe>) <valeur>]
```

en écriture d'un slot.

Sur l'exemple Fig. 33, le frame `f1` est représentant des classes `Occupied` et `Reserved`, du point de vue de l'occupation. Ces classes constituent elles-mêmes des points de vue sur `f1`. Lors de l'accès à `f1`, l'association d'une expression de point de vue permet de choisir les caractéristiques liées à ce point de vue. On accède alors au slot `contact` tel que défini dans la classe `Occupied` ou au slot `contact` (différent) tel que défini par la classe `Reserved`, suivant le point de vue spécifié :

```
[f1 '? '(contact as occupied)]
; lecture du slot contact sous le point de vue de la classe occupied :
; retourne le numéro de téléphone du locataire

[f1 '<- '(contact as reserved) 19331214]
; écriture du slot contact sous le point de vue de la classe reserved :
; affecte le numéro de téléphone de l'aspirant locataire.
```

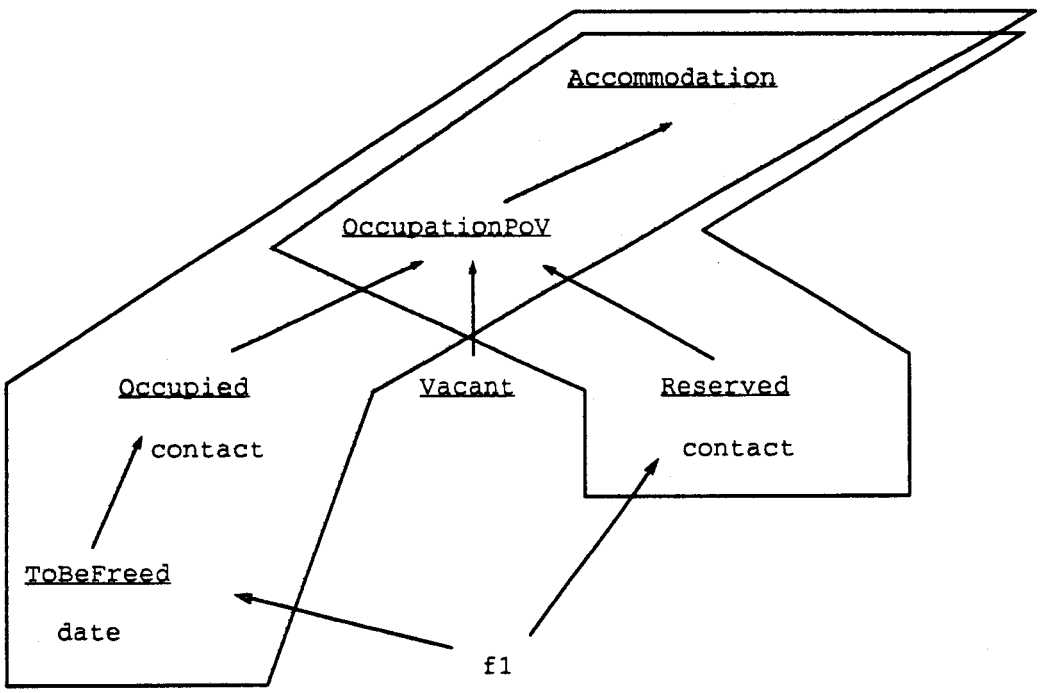


Fig. 33 Sélection d'un slot par la sélection d'un point de vue

De façon générale, l'expression de point de vue peut être associée à n'importe quel envoi de message à un frame, pour activer une de ses méthodes sous un point de vue particulier :

```
[<unFrame> '(<nom-méthode> as <classe>) <arguments>]
```

L'expression d'un point de vue a pour effet de lever les conflits entre caractéristiques homonymes (slots ou méthodes) définies sous des points de vue différents.

3.3.3 Conflit de définitions d'un même slot

La sélection d'un point de vue sur un frame s'intéresse à la description du frame telle qu'elle est représentée par la sous-classification correspondante. Le frame, ses méthodes et ses slots sont interprétés selon le point de vue choisi. Cela a deux conséquences. D'une part, l'attention est portée sur certaines caractéristiques plutôt que sur d'autres, comme on a vu dans le cas des slots au paragraphe précédent. D'autre part, telle interprétation d'un slot est choisie plutôt que telle autre, quand un slot a de multiples interprétations suivant différents points de vue.

L'interprétation d'un slot est donnée dans sa description, à travers les différentes facettes non-contraignantes associées. La sélection d'un point de vue lors d'un accès en lecture déclenche les réflexes *$before-get*, *$after-get* et éventuellement *$if-needed* (et/ou la lecture de la facette *$default*) associés au slot sous ce point de vue. Les facettes contraignantes, par contre, ne font pas partie de l'interprétation du slot, mais sont intrinsèquement liées à sa définition[1]. La sélection d'un point de vue ne permet jamais de contourner des contraintes spécifiées sous un autre point de vue. La cohérence est assurée par le respect du principe de multiplicité des contraintes (cf. page 84).

Considérons de nouveau notre appartement `f1` (Fig. 34). Sa description est enrichie par le point de vue financier (`FinancePoV`). Les intérêts financiers du propriétaire et ceux du locataire sont décrits dans les classes `OwnerPoV` et `TenantPoV` et correspondent à des "sous-points de vue" sur `f1`. En effet, le propriétaire et le locataire ont chacun leurs propres taxes à payer ; certains frais de réparation incombent au propriétaire, d'autres au locataire.

Quant au loyer, il n'est pas vu de la même façon par le propriétaire et par le locataire. Pour le propriétaire, c'est de l'argent qui rentre, pour le locataire, c'est de l'argent qui sort. Le point de vue du locataire s'intéresse à la date où le locataire a payé le dernier loyer. Du point de vue du propriétaire, par contre, la date d'encaissement du chèque est plus importante. L'information associée lors de la lecture du loyer dépend alors du point de vue : selon le point de vue du locataire, la date du chèque le plus récent est retournée, et selon le point de vue du propriétaire, la date de dernière réception. Respectivement :

```
> [f1 '? '(monthlyrent as tenantpov)]
  Rent of 2275 paid (12 - 29 - 1991)
  = 2275

> [f1 '? '(monthlyrent as ownerpov)]
  Rent of 2275 received (1 - 3 - 1992)
  = 2275
```

Les réflexes *$after-get* qui retournent ces informations en fonction du point de vue sont définis comme des affinements du slot `monthlyrent`, qui est défini dans `FinancePov`. Un représentant de la classe `FinancePov`, ou de ses sous-classes, a un seul slot `monthlyrent`. Ce slot est spécialisé de manière multiple dans les sous-classes. Cette spécialisation multiple traduit la multiplicité des interprétations du slot suivant les différents points de vue.

Ce cas est à distinguer des slots homonymes, définis dans plusieurs classes indépendantes, comme les slots `repaircosts` et `rates`. Il s'agit alors de slots différents, avec des valeurs

1. Voir la table des facettes contraignantes et non-contraignantes Fig. 27 (page 80).

différentes, et non pas d'un même slot interprété de manière multiple.

Le principe de multiplicité des contraintes garantit le respect de toutes les contraintes, même en cas de sélection de point de vue. La tentative de modification du loyer :

```
> [f1 '<- '(monthlyrent as ownerpov) 3000]
  Rent of 2275 paid (12 - 29 - 1991)
  ** ROME error :monthlyrentf1g195: constraints not verified
```

est soldée par un échec. Une contrainte spécifiée dans la facette *$verify* du slot `monthlyrent` sous le point de vue du locataire contrôle la hausse des loyers. Celle-ci ne peut excéder 5%. Même en sélectionnant le point de vue du propriétaire, cette contrainte est appliquée. En effet, les contraintes de tous les points de vue sont toujours appliquées, et ne peuvent être contournées en sélectionnant un point de vue.

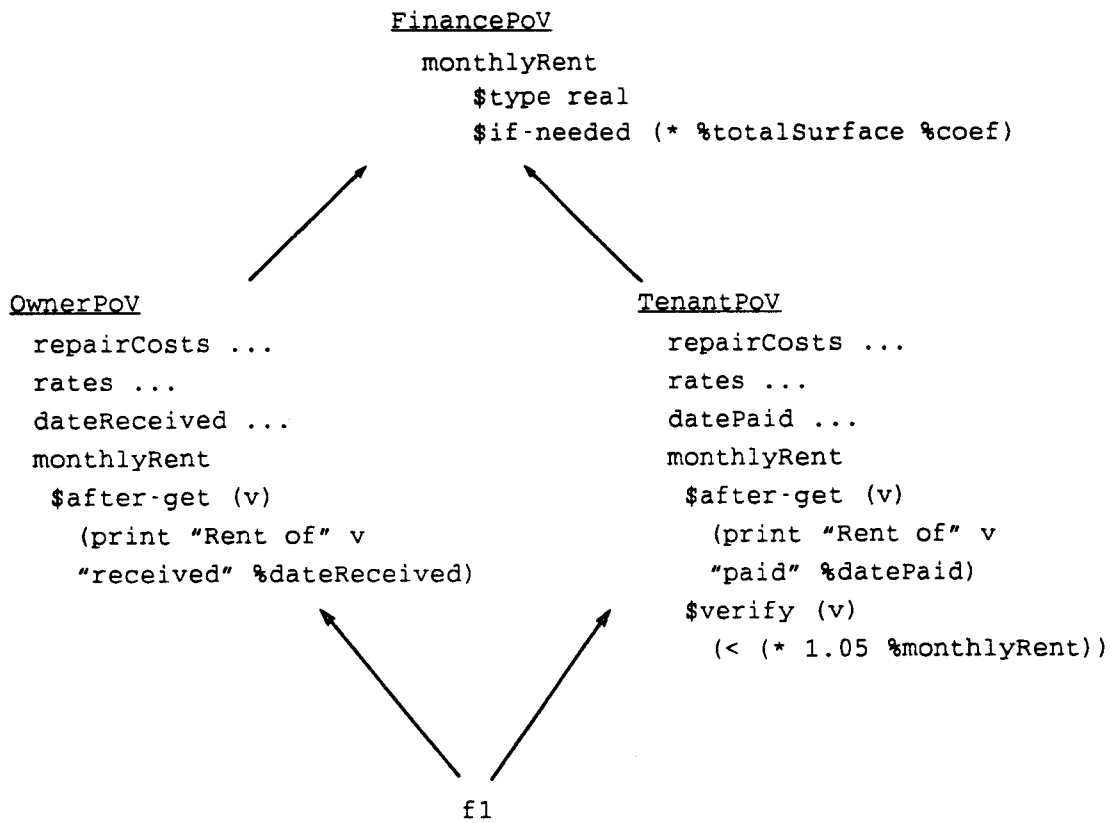


Fig. 34 Interprétations multiples d'un même slot

3.3.4 Héritage multiple

La spécialisation d'une description (classe de frame) est exprimée par une sous-classe, qui ajoute de nouvelles caractéristiques ou affine les caractéristiques héritées (cf. 3.2). De la même façon, la spécialisation de plusieurs classes est obtenue par création d'une sous-classe, qui spécialise alors plusieurs super-classes, dont elle hérite toutes les caractéristiques. La spécialisation multiple (induisant l'héritage multiple) est limitée par les relations d'exclusion qui peuvent exister entre classes (cf. page 86). La création d'une sous-classe commune à deux

classes en relation d'exclusion est interdite.

La spécialisation multiple introduit le même genre de situations que celles exposées en 3.3.2 et 3.3.3.

3.3.4.1 Héritage multiple de slots homonymes

Reprenons la sous-classification `OccupationPoV` des appartements. Certains locataires souhaitent laisser l'appartement qu'ils libèrent à un ami. Ces appartements sont décrits par la classe `ReservedForFriend`, qui spécialise les deux classes `Reserved` *et* `ToBeFreed` (cf. Fig. 35).

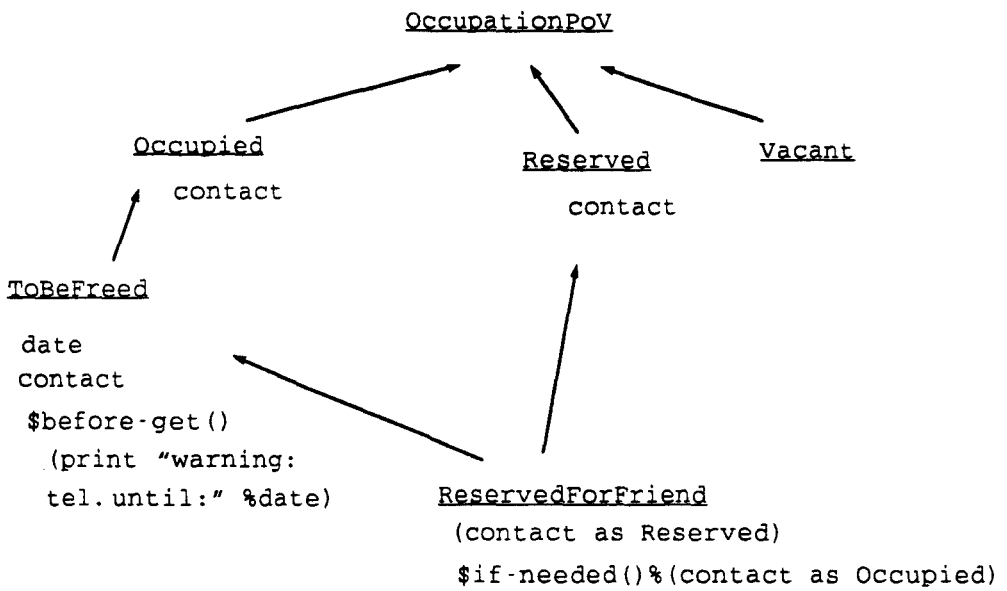


Fig. 35 Héritage multiple = Spécialisation multiple

Une sous-classe hérite toutes les caractéristiques de ses super-classes, en particulier, les deux slots `contact` sont hérités ici. Chacun des deux peut être redéfini indépendamment. Ici, le slot `contact` hérité de `Reserved` est spécialisé avec une facette de calcul *$if-needed*. Au cas où le numéro de téléphone de la personne qui réserve le logement est inconnu, on a recours a celui de son ami, qui saura certainement le trouver :

```
(contact as Reserved)
        ($if-needed () %(contact as Occupied)).
```

Cet exemple montre que la sélection de point de vue est également un moyen pour choisir un slot à affiner parmi plusieurs slots homonymes hérités. Il est à noter que l'affinement d'un slot est toujours respecté (cf. le respect du principe d'affinement pour les méthodes [Car&90b]). Le respect de l'affinement implique la prise en compte de la facette *$before-get*

associée à la spécialisation du slot `contact` dans `ToBeFreed`, lors de la lecture du slot `contact` "as occupied".

3.3.4.2 Héritage de définitions multiples d'un même slot

L'interaction de plusieurs points de vue identifiés sur le graphe de classification peut résulter en un héritage multiple traduisant la spécialisation de plusieurs descriptions confrontées les unes aux autres. Le graphe de la Fig. 36 donne un exemple de l'interaction entre deux points de vue explicitement identifiés dans la conception. Le point de vue de l'état (`StatePoV`) classe les logements dans les classes `Recent`, `NotTooOld` et `Old`, en fonction de leur âge, et partitionne les représentants de la classe `Old` en `Renovated` et `ToBeDemolished`. Par ailleurs, nous avons le point de vue de l'occupation, qui sous-classifie les logements en occupés, vacants, réservés,... comme on a vu (Fig. 33, Fig. 35).

Maintenant, les logements à démolir qui sont encore occupés par un locataire peuvent être décrits par une classe particulière `ToBeReplaced`, classe inter-point de vue. En effet, il faudra reloger les personnes occupant des logements à démolir. Cette classe est donc une spécialisation des deux classes `ToBeDemolished` et `Occupied`, affinant (par exemple) le slot

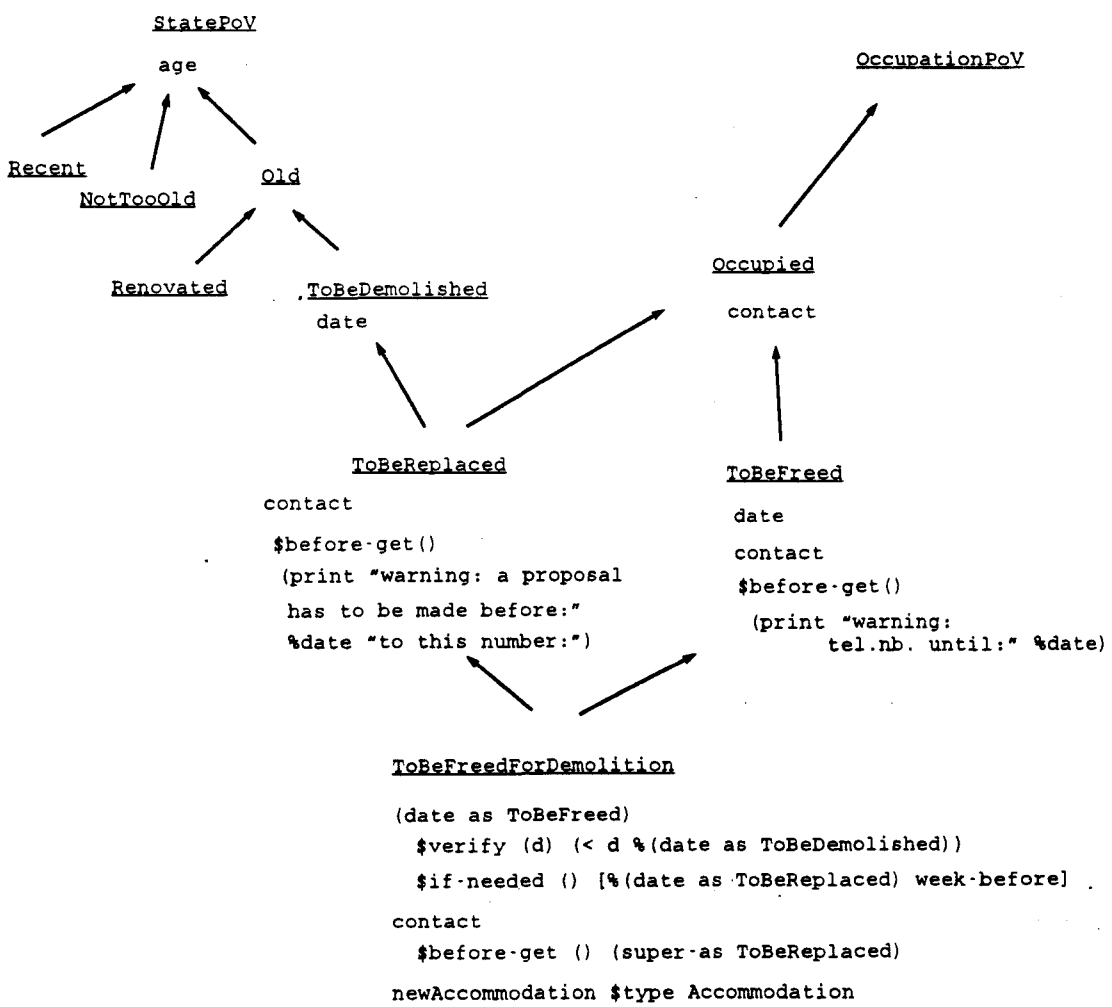


Fig. 36: Spécialisation inter-point de vue

contact avec un réflexe qui rappelle l'urgence de trouver un autre logement pour ce locataire :

```
contact
        ($before-get ()
              (print "warning: a proposal has to be
               made before" %date "to this number:").
```

La sous-classe ToBeFreedForDemolition décrit enfin tous ces logements qui doivent être libérés pour cause de démolition prochaine. Ce sont des logements à la fois à libérer (ToBeFreed) et, avant tout, à remplacer (ToBeReplaced). Cette classe hérite de toutes ses super-classes, et en particulier de leurs différents slots date. Il s'agit de la date de démolition et de la date de libération du logement. Le slot correspondant à cette dernière est spécialisé par l'ajout d'un moyen de calcul, prenant en compte la date de démolition, et d'une contrainte : le logement doit évidemment être libéré avant d'être détruit :

```
(date as ToBeFreed)
        ($verify (d) (< d %(date as ToBeDemolished))
         $if-needed ()
               [%(date as ToBeReplaced) week-before]).
```

Il est à noter ici que les deux expressions %(date as TobeDemolished) et %(date as ToBeReplaced) sélectionnent le même slot date ; la sélection d'un point de vue n'est pas la sélection d'une classe, mais bien de la sous-classification entière reliée à cette classe.

Quant au slot contact, spécialisé de manière multiple : l'interprétation du slot peut être choisie dans l'affinement, en utilisant la sélection de point de vue. Ici, l'une des deux facettes *$before-get* est sélectionnée par *super-as* expression :

```
contact
        ($before-get () (super-as ToBeReplaced))
```

En effet, pour un logement à démolir, il est plus important de rappeler l'urgence de faire une proposition de nouveau logement, que de rappeler que le numéro de téléphone n'est plus valable après la date de libération.

3.4 La représentation évolutive de frames

Nous avons discuté des avantages offerts par la représentation dynamique en 2.3.2. En effet, le caractère essentiellement partiel et incrémental de la description d'un domaine dans une hiérarchie de classes implique la nécessité d'affinements ultérieurs. L'affinement par sous-classement induit une spécialisation des instances des classes affinées. De nouveaux points de vue ajoutent des classifications supplémentaires d'un concept, enrichissant les frames existants avec une nouvelle "dimension". La description de notre appartement f1 du point de vue financier peut être ajouté a posteriori (cf. Fig. 32). Ces enrichissements de description requièrent des changements de type incrémental.

Un autre type de changement, plus radical, est celui qui représente une modification de l'objet représenté, ou une modification des connaissances que nous avons sur un objet.

Un logement vacant peut devenir occupé quand un nouveau locataire emménage. L'appartenance à la classe vacant n'est alors plus pertinente pour le frame qui représente ce

logement. Un appartement rattaché à la classe `Upper` n'est plus correctement représenté à partir du moment où l'on apprend qu'il se situe au 1er étage.

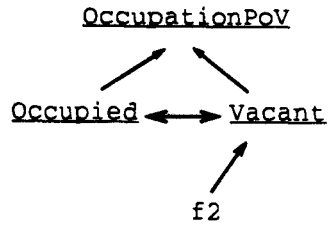


Fig. 37

La représentation évolutive des frames se traduit par des liens dynamiques de représentation entre un frame et ses classes (cf. ROME [Car&90a]).

L'ajout d'un lien de représentation rattache un frame à une nouvelle classe :

```
[<aFrame> 'r+ <C>]
```

tandis que la destruction de ce lien le détache de la classe :

```
[<aFrame> 'r- <C>].
```

Le frame reste un représentant des superclasses de `C`.

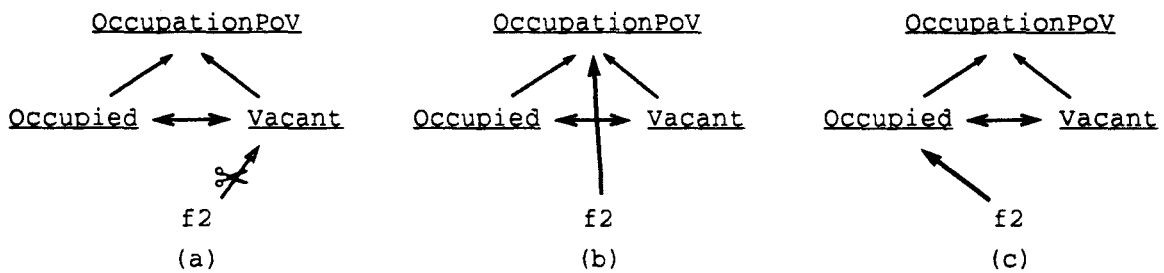


Fig. 38: Evolution

Exemple (Fig. 38):

```
[f2 'r- 'Vacant] (a puis b)
[f2 'r+ 'Occupied] (c)
```

Ces messages expriment l'évolution de f2 de la classe `Vacant` vers `Occupied`.

Un frame ne peut être représentant d'une classe dont il ne respecte pas les contraintes. Les contraintes imposées par une classe concernent les définitions des slots ou la relation d'exclusion entre la classe et une des classes de représentation du frame. La représentation évolutive est donc contrôlée par ces contraintes.

Dans l'exemple (Fig. 38), `f2` ne peut être rattaché à `Occupied` tant qu'il est représentant de `Vacant`, à cause de la relation d'exclusion entre ces deux classes. Les contraintes dues à la relation d'exclusion et celles imposées par la spécialisation des slots, sont vérifiées lors de l'évolution du frame. En effet, la spécialisation d'un frame peut entraîner la restriction des

domaines de ses slots, comme on peut en voir un exemple à la Fig. 39. Cet exemple représente la maison `h1` sous le nouveau point de vue `ComfortPoV`. La spécialisation sous ce point de vue est contrainte par la restriction des domaines de divers slots. Etant donné les valeurs des slots de `h1`, l'évolution vers la classe `Luxurious` est interdite. En effet, `h1` n'a pas de chauffage, alors que l'admissibilité pour le slot `heating` de la classe Luxurious est `central`. La seule classe vers laquelle `h1` pourrait être spécialisée dans cet état est la classe `Uncomfortable`.

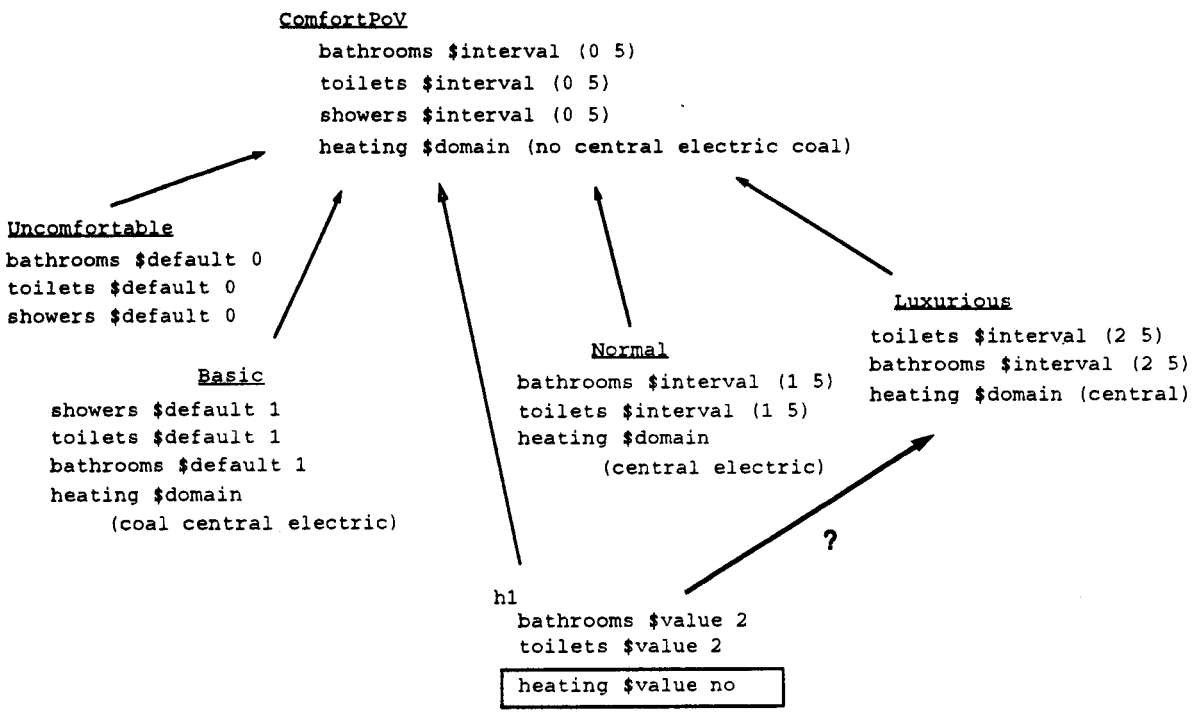


Fig. 39: Contraintes de spécialisation

3.5 Le filtrage

Dans plusieurs systèmes de représentation par objets offrant du filtrage, la notion de classe intervient pour implanter le filtre. La classe étant la description (l'abstraction) de ses représentants, elle peut servir d'implantation pour un filtre, qui décrit un ensemble d'objets — ceux vérifiant le filtre —, parfois appelés ses instances. La classe implantant le filtre joue rarement un rôle autre que ce rôle d'implantation. Elle n'est pas présente de manière explicite dans la base de connaissances et disparaît après l'opération de filtrage.

En FROME, les filtres ne sont pas seulement implantés par des classes (de manière orientée objet), mais ils possèdent le statut de classe pour étendre la base de connaissances et en faire partie ([Dek93]). Le filtre est représenté comme une classe abstraite qui filtre ses représentants. Le filtre devient alors un moyen de définir de nouvelles classes et d'y rattacher leurs représentants en utilisant le filtrage (cf. [Cas85])[1]. Il constitue de cette façon une aide à la conception incrémentale de la base de connaissances par filtrage. De plus, il rend possible le filtrage incrémental par réutilisation de filtres partiels.

3.5.1 Filtres volatils et filtres rémanents

Le filtrage en FROME permet de sélectionner toutes les instances de frames correspondant à un ensemble de critères, spécifiés dans un filtre (cf. 2.4 sur le filtrage). Ces filtres peuvent être utilisés soit indépendamment, sous forme de requêtes, soit attachés aux slots d'un frame au moyen de la facette `$if-needed-match`.

Nous distinguons les filtres conjonctifs et les filtres disjonctifs, un filtre disjonctif étant la disjonction de plusieurs filtres conjonctifs.

Pour disposer aussi bien de filtres temporaires et anonymes que de filtres nommés destinés à persister comme nouvelle connaissance, nous introduisons les deux notions suivantes :

- le filtre volatil
- le filtre rémanent

Le filtre volatil, anonyme, existe le temps du filtrage et disparaît ensuite. Le filtre rémanent, nommé, continue d'exister en tant que classe dans la hiérarchie. Les instances filtrées sont ses représentants.

3.5.2 Les filtres conjonctifs

Dans un contexte de représentation multiple, une requête conjonctive autorise la sélection de frames appartenant à plusieurs classes et vérifiant certaines conditions exprimées comme des restrictions des domaines de leurs attributs (*slots*).

1. Cette technique peut également être rapprochée de celle qui consiste à dériver des *vues* dans les bases de données, par exemple [Sch&91] [Sou&93].

L'expression générale d'une requête conjonctive se présente comme suit :

```
(match '(<classe 1> ... <classe n>)
 'in <classe-filtre>
 'not '(<classe m> ... <classe p>)
 'where '(<slot 1> (<facette 11> <val 11> ... <facette 1q> <val 1q>)
          ...
          <slot r> (<facette r1> <val r1> ... <facette rs> <val rs>))
 'do '(<app-meth>))
```

Cette requête filtre tous les frames représentants des classes <classe 1> ... <classe n>[1], qui ne sont pas représentants des classes <classe m> ... <classe p>[2] et dont les slots <slot 1> ... <slot r> satisfont les contraintes exprimées dans les facettes associées sous la clause *where*. La formulation des slots contraints s'exprime dans le langage de description des slots par les classes, avec le même ensemble de facettes contraignantes[3]. La partie *do* permet de spécifier un ensemble de méthodes à appliquer par les objets filtrés. Les parties *in*, *where*, *not* et *do* sont optionnelles.

La partie *in* sert à spécifier que le filtre est rémanent. L'argument est le nom de la classe de filtre qui recueille les instances filtrées. Sans la partie *in*, le filtre est volatil.

La forme abrégée sans la partie *where* sélectionne simplement les représentants des classes, sans contraintes supplémentaires.

```
(match '(<classe 1> ... <classe n>))
```

3.5.2.1 Les filtres conjonctifs volatils

Dans les requêtes conjonctives, le filtrage se fait sur les représentants appartenant à **toutes** les classes passées en argument. Seuls les représentants dont les slots satisfont les contraintes supplémentaires imposées par la partie *where* sont retenus.
Exemple de filtre conjonctif volatil :

```
(match '(Etudiant Etranger)
 'where '(nationalite ($domain (b nl l)))).
```

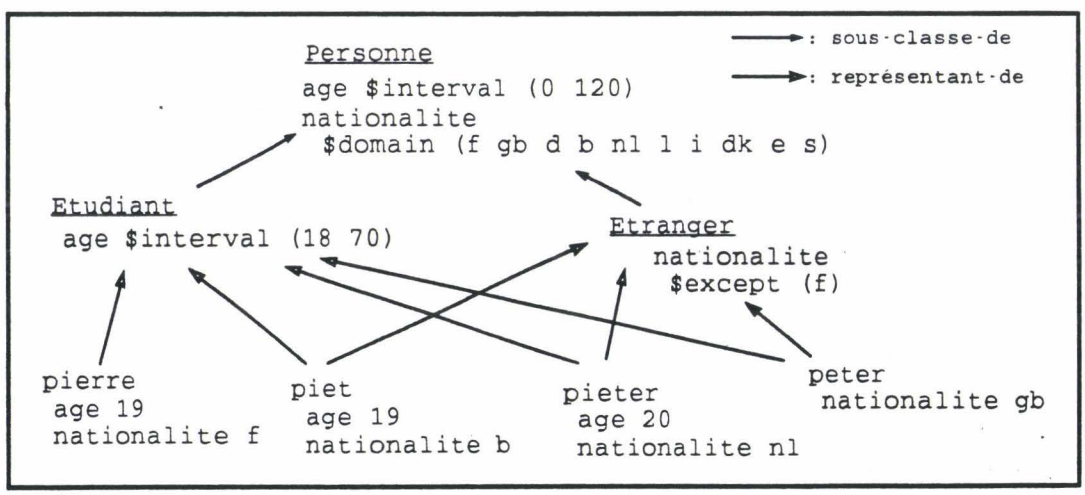


Fig. 40: Filtrage d'étudiants étrangers

1. Les frames qui sont représentants de *toutes* ces classes.
2. Ils ne sont représentants d'*aucune* de ces classes.
3. Toute expression de slot, en particulier paramétrée par un point de vue, est autorisée (cf. 3.5.2.5).

Avec les classes et instances de la Fig. 40, cette requête filtre `piet` et `pieter`. En effet, `peter` ne satisfait pas la contrainte imposée par le filtre sur la nationalité et `pierre` n'est pas étranger. Le résultat d'une requête avec filtre volatil est l'ensemble des instances de frames satisfaisant le filtre.

3.5.2.2 Les filtres conjonctifs rémanents

Le filtre rémanent est destiné à persister en tant que classe dans la base de connaissances. Les instances résultat du filtrage lui sont rattachées et appartiennent désormais à cette classe. La classe correspondant à un filtre conjonctif est placée dans la hiérarchie comme sous-classe directe des classes candidates spécifiées dans le filtre.

Voici la même requête que celle de 3.5.2.1, cette fois-ci avec filtre rémanent :

```
(match '(Etudiant Etranger)
 'in 'Etudiant-du-Benelux
 'where '(nationalite ($domain (b nl l))))
```

Le résultat du filtrage est le même, avec la différence qu'une nouvelle classe a été créée dont les représentants sont les instances filtrées (cf. la Fig. 41). Les instances `piet` et `pieter` qui étaient représentants des classes `Etudiant` et `Etranger` **se sont spécialisées** pour devenir représentants de la classe `Etudiant-du-Benelux`.

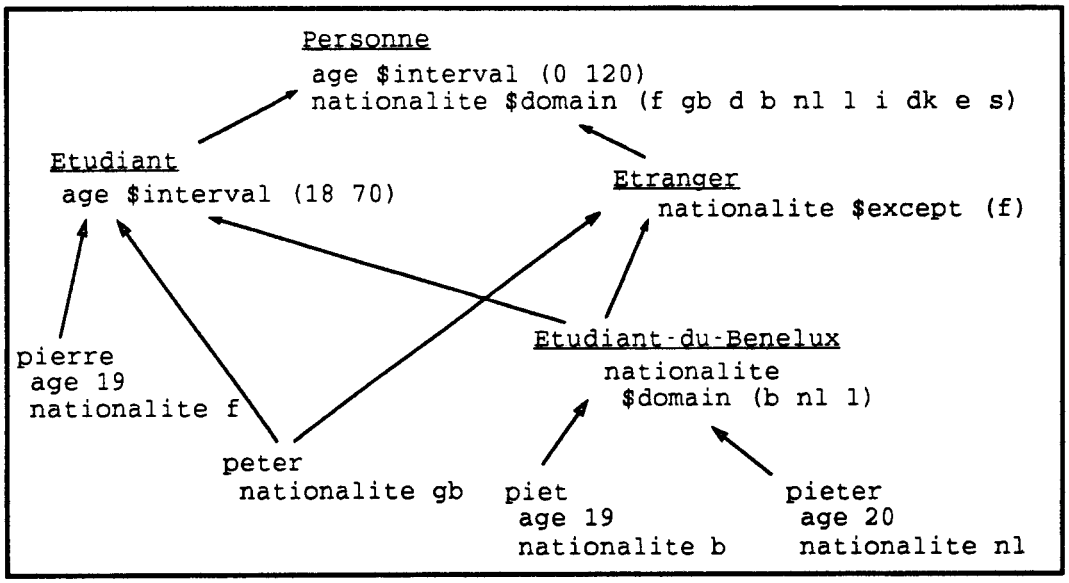


Fig. 41: Filtrage d'étudiants du Benelux

Cette nouvelle classe constitue un enrichissement de la base et montre ainsi l'intérêt du filtrage pour la conception incrémentale de la hiérarchie des classes.

L'héritage permet d'opérer des filtrages successifs par la réutilisation de filtres rémanents, représentés par des classes dans la hiérarchie.

Une requête plus spécialisée qui réutilise la classe du filtre `Etudiant-du-Benelux` est par exemple celle qui filtre tous les étudiants Belges agés de moins de 20 ans :

```
(match '(Etudiant-du-Benelux)
 'in 'Jeune-Etudiant-Belge
 'where '(nationalite ($domain (b))
          age ($interval (18 20))))
```

Notons qu'il ne s'agit pas ici d'un placement de la classe filtre par calcul de subsomption de classes, tel que réalisé par les classificateurs des langages de la famille KL-ONE. En effet, ces classificateurs comparent des concepts *définis* et les placent les uns par rapport aux autres suivant la relation de subsomption. En Frome, seuls les filtres peuvent être considérés comme des concepts définis, les classes s'apparentant plutôt à des concepts *primitifs* dans le cas général (description des conditions nécessaires et non suffisantes d'appartenance à la classe). Le mécanisme de filtrage en Frome comprend une opération de comparaison d'instances à la classe filtre. La comparaison de classes les unes par rapport aux autres n'est pas prise en compte [Cap92][Cas91a]. Une telle comparaison serait nécessaire pour faire un placement plus fin de la classe filtre[1] que son placement comme sous-classe directe des classes spécifiées dans le filtre.

3.5.2.3 L'exclusion de classes

Le *not* prend comme argument une liste de classes dont les représentants sont à exclure du filtrage. Considérons l'exemple de l'agence de location de logements (cf. Fig. 36 (page 91)). Une demande de logement peut s'exprimer sous la forme d'une requête à la base de logements gérée par l'agence. Exemple :

"Filtrer tous les logements à libérer avant le 1er juillet 1993 à l'exclusion des logements à détruire et des logements réservés".

```
(match '(ToBeFreed)
 'not '(Reserved ToBeDemolished)
 'where '(date ($verify (lambda(d)
                [d '< [date 'new* 'day 1 'month 7 'year 1993]] ))))
```

Le *not* élague le graphe des classes candidates, en rejetant les classes spécifiées dans la clause *not* et leurs sous-classes, ici : `Reserved`, `ReservedForFriend`, `ToBeDemolished`, `ToBeReplaced` et `ToBeFreedForDemolition` (cf. Fig. 36).

3.5.2.4 La clause do

La partie *do* d'une requête permet de spécifier une opération à effectuer sur tous les objets filtrés, sous la forme d'un ensemble d'applications de méthodes. En particulier, l'évolution de tous les objets filtrés vers une classe particulière, de même que l'ajout d'un point de vue supplémentaire sur ces objets, peuvent être pris en charge par la partie *do*.

Exemples :

```
(match '(Old)
 'where '(age ($interval (30 50))
          state ($domain ("bad" "very bad")))
 'do '({r+ ToBeDemolished}))

(match '(Accommodation)
 'do '({r+ ComfortPoV}))
```

Le premier exemple filtre tous les vieux logements qui ont entre 30 et 50 ans et dont l'état est mauvais ou très mauvais et les fait évoluer vers la classe `ToBeDemolished`. Le deuxième exemple ajoute le point de vue `ComfortPoV` à tous les logements existant dans la base.

1. Sous son/ses subsumant(s) le(s) plus spécifique(s).

3.5.2.5 La sélection de point de vue dans un filtre

Une requête cherchant tous les logements à démolir avant la fin de l'année et occupés par des locataires, par exemple dans le but de proposer rapidement à ces derniers un nouveau logement, peut s'exprimer comme suit :

```
(match '(ToBeDemolished Occupied)
 'where '((date as ToBeDemolished)
          ($verify (lambda(d)
               [d '< [date 'new* 'day 31 'month 12 'year 1994]] ))))
```

Remarquons que la sélection de points de vue est appliquée naturellement dans les filtres pour sélectionner un slot à contraindre (`date as ToBeDemolished`).

3.5.3 Les filtres disjonctifs

L'utilisation du *or* dans une requête introduit les filtres disjonctifs, représentant une disjonction de sous-filtres, chaque sous-filtre étant un filtre conjonctif :

```
(match 'in <classe-filtre>
 'or <classes-1>
          'in <classe-filtre-1>
          'not <classes-exclues-1>
          'where <contraintes-slots-1>
          'do <app-meth>
 'or <classes-2>
          'in <classe-filtre-2>
          'not <classes-exclues-2>
          'where <contraintes-slots-2>
          'do <app-meth>
 ...
 'or <classes-n>
          'in <classe-filtre-n>
          'not <classes-exclues-n>
          'where <contraintes-slots-n>
          'do <app-meth>)
```

Cette requête filtre tous les frames représentants d'une des parties *or*. [1] Le résultat du filtrage est la réunion de tous les résultats partiels des sous-filtres.
Toutes les parties *in*, *where*, *not* et *do* sont optionnelles.

3.5.3.1 Les filtres disjonctifs volatils

La disjonction de filtres autorise les requêtes du style :

"Filtrer tous les appartements vacants, non-reservés et non à démolir, qui ne sont pas situés aux étages supérieurs et dont le loyer est inférieur à 2000, et les maisons vacantes et ni réservées ni à démolir, et dont le loyer est inférieur à 2800" (cf.Fig. 43).

Cette requête peut correspondre à une demande de logement prenant en compte des choix alternatifs. Le filtre disjonctif permet de réunir des instances appartenant à différentes classes, qui peuvent être en relation d'exclusion (comme ici `Flat` et `House`).

1. Ils sont filtrés par *au moins un* des sous-filtres.

```
(match
 'or '(Flat Vacant FinancePov)
          'not '(Upper)
          'where '(monthlyRent ($interval (0 2000)))
 'or '(House Vacant FinancePov)
          'where '(monthlyRent ($interval (0 2800))))
```

3.5.3.2 Les filtres disjonctifs rémanents

La classe représentant un filtre disjonctif rémanent est placée dans la hiérarchie comme superclasse directe de toutes les classes représentant ses sous-filtres. Sa (ses) superclasse(s) est (sont) calculée(s) comme la (les) plus petite(s) superclasse(s)[1] commune(s) de tous les sous-filtres. Algorithme (en $O(n^2)$) :

(1) calculer la liste des superclasses (fermeture transitive), sans doublons, pour chaque sous-filtre, par message de sélecteur *superclg*
(2) calculer l'intersection de toutes les listes obtenues
(3) supprimer dans la liste obtenue toutes les classes qui sont superclasse (générale) d'une ou plusieurs autres classes dans la liste

Les classes de cette liste sont indépendantes deux à deux (conséquence de (3)) et sont superclasses de tous les sous-filtres (par (2)). De plus, il s'agit bien des plus spécifiques (par (3)). Placer le filtre disjonctif comme sous-classe directe de toutes ces classes et comme superclasse directe de tous les sous-filtres le rend donc plus petit majorant (pour l'héritage) des sous-filtres.

Les sous-filtres d'un filtre rémanent peuvent être volatils ou rémanents[2].

Les représentants des sous-filtres calculés par filtrage sont aussi représentants de leurs superclasses, en particulier du filtre englobant. Par conséquent, l'ensemble des représentants du filtre englobant est constitué par la réunion des représentants de ses sous-classes.

Il faut noter que le filtre disjonctif a bien été construit pour représenter une disjonction, mais que la classe résultante n'a pas une vraie sémantique de "classe ou". Elle représente la disjonction en extension (par construction), car ses représentants sont la réunion des représentants de ses sous-classes, mais non en intension, car elle ne décrit pas les caractéristiques communes de ces représentants. Cf. aussi [McA&86] sur les problèmes liés à la sémantique des classes "ou".

Nous proposons d'introduire un nouveau concept qui est celui des logements situés au rez-de-chaussée, donc accessibles aux personnes handicapées par exemple. La requête correspondante est exprimée à l'aide d'un filtre rémanent disjonctif, qui est inséré comme nouvelle classe dans la hiérarchie, sous le nom de `GroundLevelAccommodation` :

1. La ou les plus petite(s) au sens de "la (les) plus spécifique(s)".
2. Les sous-filtres d'un filtre volatil sont volatils, car les sous-classes ne peuvent exister sans leur superclasse.

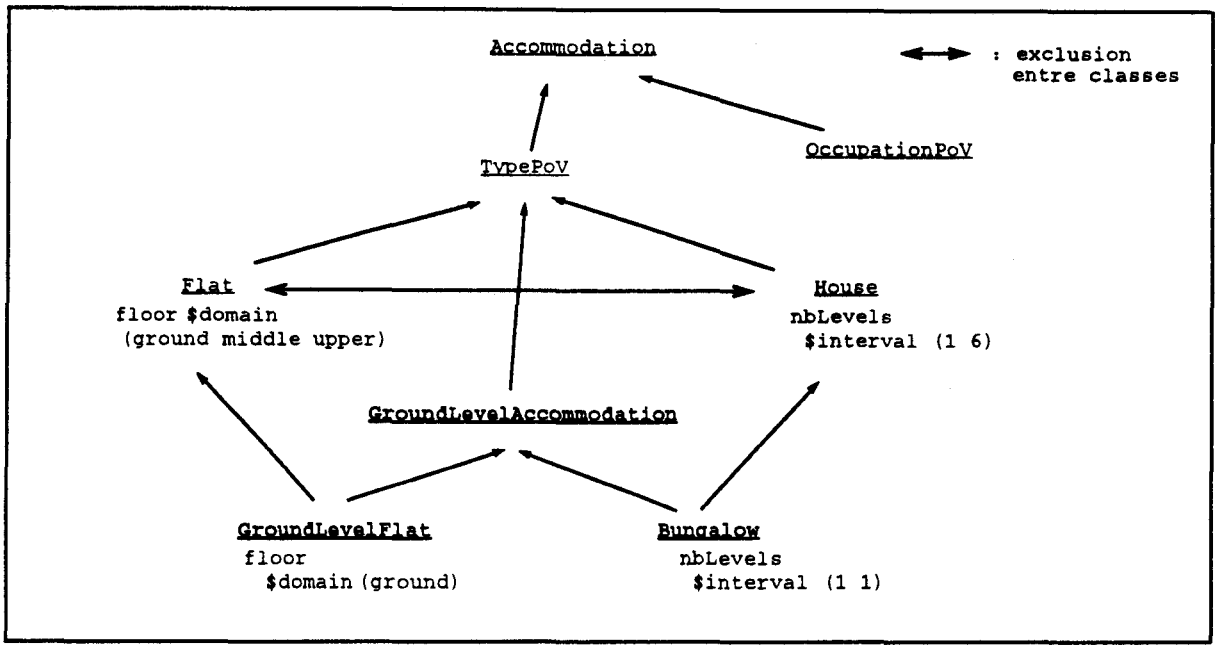


Fig. 42: Conception incrémentale avec filtre disjonctif rémanent

"Filtrer tous les appartements qui se situent au rez-de-chaussée, et les maisons de plain-pied" (cf. Fig. 42)

```
(match 'in GroundLevelAccommodation
 'or '(Flat)
          'in GroundLevelFlat
          'where '(floor ($domain (ground)))
 'or '(House)
          'in 'Bungalow
          'where '(nbLevels ($interval (1 1))))
```

Les deux sous-filtres sont également des filtres rémanents, représentant les concepts de `Bungalow` et `GroundLevelFlat`. Ces trois filtres rémanents viennent enrichir la base avec trois nouvelles classes. Les deux sous-filtres `Bungalow` et `GroundLevelFlat` sont placés comme sous-classes de la classe `GroundLevelAccommodation`. Cette dernière est placée sous la superclasse commune la plus spécifique des classes `Flat` et `House`, ici `TypePoV`.

Le filtrage constitue donc un moyen d'enrichir la base avec de nouvelles classes, définies à l'aide d'un filtre. Tous les objets de la base qui correspondent à leur description y sont rattachés par filtrage.

Conservation de résultats

Une exploitation plus opératoire du filtrage est également possible. Elle consiste à utiliser les filtres rémanents pour la simple conservation du résultat du filtrage en vue d'une réutilisation future. Le filtre pourrait être réactivé à tout moment pour recueillir d'éventuelles nouvelles instances le satisfaisant. La requête correspondant à la demande de logement ci-dessus (en 3.5.3.1) pourrait ainsi être conservée comme la demande de logement de Laurent, qui s'est vu attribuer le numéro de dossier 167.

Elle peut alors se formuler de la façon suivante :

```
(match 'in 'Request-no.167
 'or '(Flat Vacant FinancePov)
          'in 'Flats-no.167
          'not '(Upper)
          'where '(monthlyRent ($interval (0 2000)))
 'or '(House Vacant FinancePov)
          'in 'Houses-no.167
          'where '(monthlyRent ($interval (0 2800))))
```

Les classes correspondant au filtre disjonctif rémanent (`Request-no.167`) et à ses différents sous-filtres (`Flats-no.167` et `Houses-no.167`) sont représentées ci-dessous.

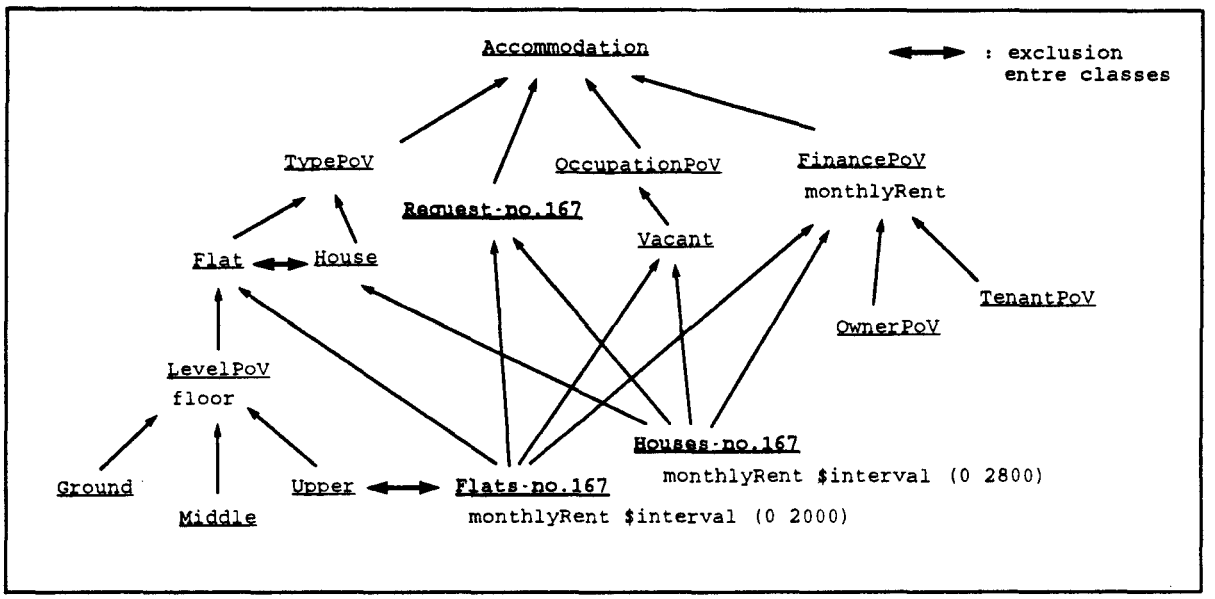


Fig. 43: Représentation multi-points de vue de logements (b)

Il s'agit d'une utilisation des filtres rémanents à un autre niveau que celui de la définition de nouvelles classes par conception incrémentale. Elle demande une gestion adaptée de la hiérarchie des classes ainsi obtenue.

3.5.4 La facette de filtrage

Le filtrage peut être utilisé comme moyen d'inférence de la valeur d'un slot, comme en Shirka. Une facette spécifique, appelée `$if-needed-match`, introduit le filtre à utiliser (cf. la facette `$sib-filtre`). La syntaxe du filtre suit celle des requêtes et a la même structure que la liste de description des slots d'une classe de frames :

```
[<meta-classe> 'new <classe-de-frames> 'supercl <super-classes>
 'slots '(<slot-1>
              (<facette-1.1> <valeur-1.1>
              ...
               #if-needed-match
                ((<classe 1> ... <classe n>)
                'in <classe-filtre>
                'not '(<classe m> ... <classe p>)
                'where
                  '(<slot 1>
                      (<facette 11> <val 11> ... <facette 1q> <val 1q>)
                    ...
                    <slot r>
                      (<facette r1> <val r1> ... <facette rs> <val rs>))
                'do '(<app-meth>))
              ...
              <facette-1.w> <valeur-1.w>)
         ...) ...]
```

Ces facettes de filtrage offrent donc le même filtrage que les requêtes. Elles n'ont toutefois pas la puissance des facettes de filtrage de Shirka, qui autorisent l'utilisation de variables et l'imbrication des filtres.

3.6 Les règles de production

FROMER est un module de règles défini pour FROME. Cet outil est décrit dans [Ham91]. Il s'agit d'une implantation des règles selon l'approche de MERING I (cf. 2.4.3).

L'exemple de la règle petits-fils s'écrit en FROMER :

```
[metarule 'new 'petits-fils 'supercl 'rule
 'slots '( x ($type personne $supercl argument)
           y ($type homme $supercl variable)
           z ($type personne $supercl variable))
 'si       '(and (?= x [z '? 'parent])
                 (?= z [y '? 'parent]))
 'alors '[x 'addval 'petits-fils y] ]
 ; si x est le parent de z et z est le parent de y
 ; ajouter y aux petits-fils de x
```

Comme en MERING, l'exécution d'une règle se fait par un envoi de message directement à la règle, ou par attachement dans un réflexe de slot. A la différence de MERING, on peut passer des valeurs en argument au déclenchement d'une règle (autres que l'argument proprement dit de la règle). Ainsi, dans la règle ci-dessus, l'information "Jean est le parent de Jacques" est utile quand on applique la règle à Jean (ou à un parent de Jean). Cette information peut être associée au déclenchement de la règle. Exemple :

```
[petits-fils 'applyfor 'parent '(Jacques Jean)]
```

3.7 La classification d'instances

La représentation évolutive rend possible la classification d'instances de façon primitive. En effet, un objet (frame) évolutif est spécialisé par rattachement à une ou plusieurs sous-classes de sa classe d'instanciation, à travers les liens de représentation dynamiques. La classification d'une instance par spécialisation peut être traduite par son évolution de proche en proche. En l'absence d'un moteur de classification dédié à la gestion de ce processus, le contrôle de la classification peut être réalisé primitivement de plusieurs façons : par attachement de réflexes, à l'aide de règles FROMER ou en utilisant le filtrage.

3.7.1 Classification par réflexes

Dans la mesure où la valeur d'un attribut peut être déterminante pour l'appartenance d'un objet à une classe, il suffit d'associer un réflexe à l'écriture sur cet attribut, chargé d'effectuer l'évolution correspondante. Le placement adéquat des réflexes peut ainsi enchaîner plusieurs évolutions élémentaires et résulter en un classement d'instances.

Si Chein et Cogis ont montré dans [Che&89] que la puissance de calcul de la programmation orientée réflexe est équivalente à celle de la programmation selon le modèle "tant-que", la bonne pratique des réflexes réclame néanmoins une certaine virtuosité du programmeur. L'approche de la programmation orientée réflexe est une approche procédurale complètement distribuée, ce qui rend difficile le contrôle de la cohérence.

Aucune méthodologie de la programmation orientée réflexe ne semble avoir été élaborée.

Deux approches sont envisageables pour programmer la classification d'instances par réflexes :

(1) programmation explicite : réflexes ad-hoc adaptés aux classes du domaine et attachés "à la main"
(2) programmation générique : réflexes implantant entièrement un algorithme de classification général ; les réflexes sont attachés automatiquement.

Programmation explicite

La première approche est relativement simple à mettre en œuvre, mais demande le maximum de coordination du concepteur de la hiérarchie. Pour chaque classe, il faut spécifier explicitement les conditions d'évolution de ses instances vers des sous-classes et les exprimer sous forme de réflexes attachés aux slots. Le concepteur doit donc avoir une vue suffisamment globale de la hiérarchie pour garantir la cohérence. Dans cette approche, il n'y a pas de façon standard de décider l'appartenance à une classe. Si la condition de classification dans une sous-classe dépend de contraintes sur plusieurs slots, chacun des slots doit reproduire la vérification de ces contraintes dans son réflexe associé, ce qui entraîne l'introduction de redondances. L'expression des critères de classification avec des réflexes est entièrement procédurale, donc ad-hoc, mais elle permet d'exprimer ces critères de façon précise et souple.

Programmation générique

La seconde approche est applicable quel que soit le domaine représenté. Par conséquent, elle pose le problème de l'analyse de ce domaine, et nécessite un moyen générique de

distinguer les critères de classification (propriétés discriminantes) des caractéristiques descriptives n'intervenant pas dans la classification (propriétés accidentelles ou facultatives). Nous appelons attribut discriminant un attribut dont le domaine est affiné dans plusieurs sous-classes de celle qui le définit, et dont la valeur est déterminante pour décider de l'appartenance de l'objet à une ou plusieurs de ces sous-classes, à condition de correspondre au domaine restreint spécifié dans celle(s)-ci. Ainsi, la valeur de l'attribut discriminant sexe dans la classe Personne suffit à elle seule de décider que l'objet peut être classifié dans la sous-classe Homme ou dans la sous-classe Femme, qui affinent chacune le domaine des valeurs possibles. Un deuxième attribut discriminant âge permet la classification automatique dans des sous-classes représentant des tranches d'âge. Cette classification automatique est toutefois soumise à la condition de non-violation de contraintes éventuelles sur les autres attributs valués.

L'identification d'attributs discriminants suffit pour attacher à chacun automatiquement des réflexes qui vont prendre en compte la propagation vers les sous-classes qui affinent cet attribut, dès lors qu'une valeur lui est affectée. La propagation vers les sous-classes consiste à leur demander de tester l'appartenance possible en fonction de la valeur de l'attribut discriminant et d'accepter l'objet comme représentant le cas échéant.

Etant donné le fait que la progression de la classification repose entièrement sur la valuation des attributs discriminants, les valeurs de ceux-ci peuvent éventuellement être obtenues en interaction avec l'utilisateur.

L'expression des critères de classification à l'aide d'attributs discriminants est déclarative, mais fixe et plus rigide.

Pour une illustration des deux approches, voir l'implantation discutée en 4.4.3.1.

3.7.2 Classification avec FROMER

Dans une classification programmée à l'aide de règles, deux approches peuvent encore être envisagées :

(1) règles ad hoc adaptées aux classes du domaine
(2) algorithme de classification général entièrement implanté par des règles.

Tout ce qui peut être fait avec des réflexes de manière ad-hoc, peut l'être également avec des règles FROMER.

Par contre, une approche générique à base de règles réclame des règles d'un niveau suffisamment général pour pouvoir exprimer un programme. Les règles de FROMER s'expriment sur des instances de frames, mais ne permettent pas la prise en compte de la structure d'une classe de frames, des domaines de valeurs des attributs etc. Elles ne sont pas adaptées pour définir un algorithme de classification.

A l'heure actuelle, aucun outil de classification entièrement basé sur des règles ne semble exister.

3.7.3 Classification par filtrage

Nous avons vu que les filtres étant assimilés à des classes dotées d'un comportement de filtrage, ces dernières avaient la faculté de faire évoluer des instances pour en faire leurs

représentants. Le filtrage est effectué en principe lors de la définition du filtre. La définition de filtres en cascade, comme sous-classes d'autres filtres, permet de faire évoluer leurs représentants de proche en proche. Pour prendre en compte des modifications survenant dans la base, telles que l'affectation d'attributs ou la création de nouvelles instances, la capacité de filtrage des classes de filtres doit être maintenue. Il s'agit d'activer chaque classe filtre lors de changements qui peuvent l'intéresser. La définition d'une classe filtre doit alors être accompagnée par son "installation" sur ses superclasses, afin que celles-ci puissent l'avertir d'éventuels changements. Il s'agit alors plus d'un processus de filtrage permanent que d'un processus de classification d'instances.

Il faut toutefois noter la spécificité des classes filtres par rapport aux autres classes de la base. Les classes en FROME, dans le cas général, ne peuvent pas être vues comme des définitions de concepts, comme nous l'avons remarqué en 3.5.2.2. Les contraintes exprimées dans les descriptions de slots constituent des conditions nécessaires d'appartenance à la classe, mais la non-violation de ces contraintes ne suffit pas pour que l'objet y soit rattaché. Les filtres, par contre, jouent le rôle de concepts définis. Les contraintes imposées sur les slots dans leur définition sont non seulement des conditions nécessaires, mais aussi suffisantes d'appartenance. La valuation de ces attributs est nécessaire au respect des contraintes. Le comportement des filtres prend en charge la classification des instances vérifiant leur définition. Cette classification est réalisée sous les conditions suivantes :

(1) appartenance de l'instance à toutes les superclasses du filtre
(2) appartenance de l'instance à aucune classe exclue par le filtre
(3) valuation nécessaire de tous les attributs contraints par le filtre
(4) respect des contraintes sur ces attributs

La classification par filtrage réalisée de cette manière (même par "installation" du filtre, évoquée ci-dessus) est assez restreinte a priori. En particulier, elle ne permet la classification d'instances que dans les classes "définies" que sont les filtres. De plus, les filtres n'ajoutent aucune description supplémentaire sous la forme de nouveaux attributs. Ce sont donc des classes purement définies et non des classes descriptives.

Les filtres comme règles de classification

Nous pouvons néanmoins étendre les possibilités du filtrage pour la classification en tirant profit du paramètre *do* prévu dans les filtres. De façon générale, la clause *do* spécifie des méthodes à appliquer aux instances filtrées. L'exécution de ces méthodes est donc une *conséquence* du filtrage des instances. Nous avons vu dans 3.5.2.4, qu'une application possible du *do* est l'évolution des instances filtrées. L'utilisation des filtres de cette manière peut être rapprochée des règles de CLASSIC (cf. page 62)[1]. Les règles de CLASSIC représentent une conséquence d'appartenance à un concept, et ont pour effet le rattachement d'un individu au concept en conclusion. Chaque règle fonctionne alors comme une condition (éventuellement complexe) nécessaire et suffisante d'appartenance au concept en conclusion. Dans le cas où plusieurs règles concluent sur le même concept, on dispose d'alternatives de décision pour l'appartenance : une disjonction de conditions nécessaires et suffisantes.

Les filtres pourvus de clauses *do* avec spécification d'évolution peuvent remplir le rôle de "règles de classification", attachées aux superclasses adéquates. Un processus d'activation

1. Il y a également des similitudes avec les passerelles de TROPES (cf. page 49).

doit être mis en place par l'installation des filtres sur leurs superclasses. Comme avec toutes les approches ad-hoc examinées, le problème du maintien de la cohérence se pose, ainsi que celui du statut (en tant que classes dans la hiérarchie) de ces filtres qui expriment en fait des règles.

3.7.4 Limitations

Les différentes tentatives de classification à l'aide de moyens simples tels que les réflexes, les règles et les filtres, montrent rapidement les limites de ces moyens. Le problème de la décision de l'appartenance à une classe peut se poser de multiples façons. Dans les solutions esquissées ci-dessus, les critères de décision dépendent tantôt d'attributs discriminants (combinés avec des réflexes), tantôt de conditions nécessaires et suffisantes (exprimées dans les filtres), s'ils ne sont pas donnés de façon ad-hoc (approche par réflexes ou règles explicites). Une étude plus approfondie des critères de classification, ainsi que des mécanismes mis en œuvre, s'impose. C'est l'objet de la partie II de ce mémoire.

Pour l'instant, nous allons nous pencher sur ...

4
La métaprogrammation de FROME en ROME

4.1 Introduction

L'objet de ce chapitre est la conception du langage de frames FROME, tel qu'il a été présenté au chapitre 3, comme une extension du langage orienté objet ROME ([Dek89]). Cette extension est obtenue de manière orientée objet, par métaprogrammation. Nous montrons que la conception par métaprogrammation orientée objet conserve les "bonnes propriétés" du langage sous-jacent : ses qualités orientées objet et en particulier l'extensibilité. De plus, l'orientation objet facilite l'enrichissement de notions spécifiques au langage sous-jacent. L'héritage multiple, les points de vue et la représentation multiple et évolutive de ROME sont conservés et étendus dans leur application au concept de frame ([Car&91b]).

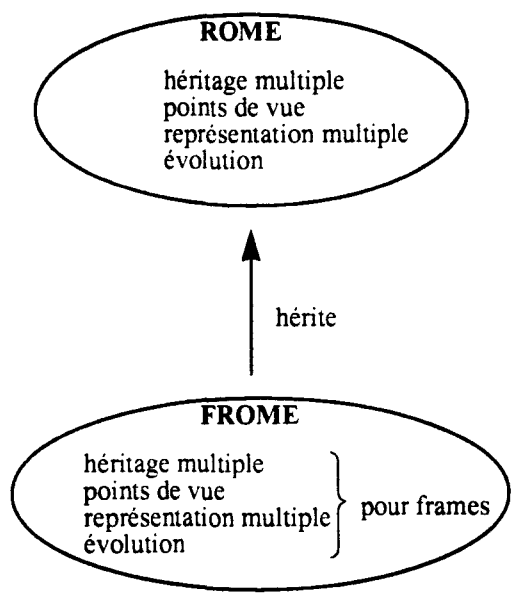


Fig. 44 Frome hérite de Rome

4.2 Extension par métaprogrammation orientée objet

Nous avons vu (cf. chapitre 2) que parmi les langages de réprésentation à base d'objets, on compte des langages de frames "purs" tels que FRL, KRL, SHIRKA, aussi bien que des langages appelés hybrides, tels que MERING, KRS, KEE, YAFOOL. Ces derniers combinent souvent la notion d'objet issue des langages de programmation orientée objet, du type Smalltalk, avec la notion de frame (schéma, unité,...). Une approche bien connue pour obtenir

un langage hybride, disposant de méthodes et d'attributs structurés est celle de la conception de PTITLOO [Fer89]. Cette approche est fondée sur l'extension d'un langage orienté objet à base de classes. Une telle extension est obtenue assez facilement par métaprogrammation si le langage sous-jacent est réflexif [Mae87a/b](cf. 2.4.4). En effet, elle requiert la définition de nouvelles classes et métaclasses. Dans les paragraphes 4.2.1 à 4.2.4, nous appliquons cette approche à ROME pour obtenir un langage hybride. Dans la section 4.3, nous montrons que le langage obtenu bénéficie des caractéristiques de ROME, qui sont spécialisées par extension du langage de frames obtenu. Dans la section 4.4, nous montrons que FROME est un langage extensible à son tour : de nouvelles fonctionnalités pour la représentation des connaissances y sont définies pour les objets de structures plus riches que sont les frames.

4.2.1 Principe de l'extension

L'idée de base de l'extension est la description des frames et des slots comme des objets du langage sous-jacent. Etant donné que les objets sont décrits par des classes (sous-classes de la classe OBJECT), les frames sont décrits par des sous-classes d'une classe appelée Frame, et leurs slots sont décrits par des sous-classes d'une classe appelée Slot. Les slots d'une instance de frame sont des instances d'une classe de slots associée. Cette idée est illustrée à la Fig. 45. L'analyse orientée objet de ce problème conduit à la définition de deux grandes familles d'objets : celle des frames et celle des slots, qui remplissent des rôles symétriques. Cette symétrie entre le "Monde des frames" (MoF) et le "Monde des slots" (MoS) forme un fil directeur tout au long de la conception de l'extension. Chacun des deux "mondes" contient ses propres instances terminales, classes et métaclasses, avec leurs responsabilités propres au sein du système de frames obtenu.

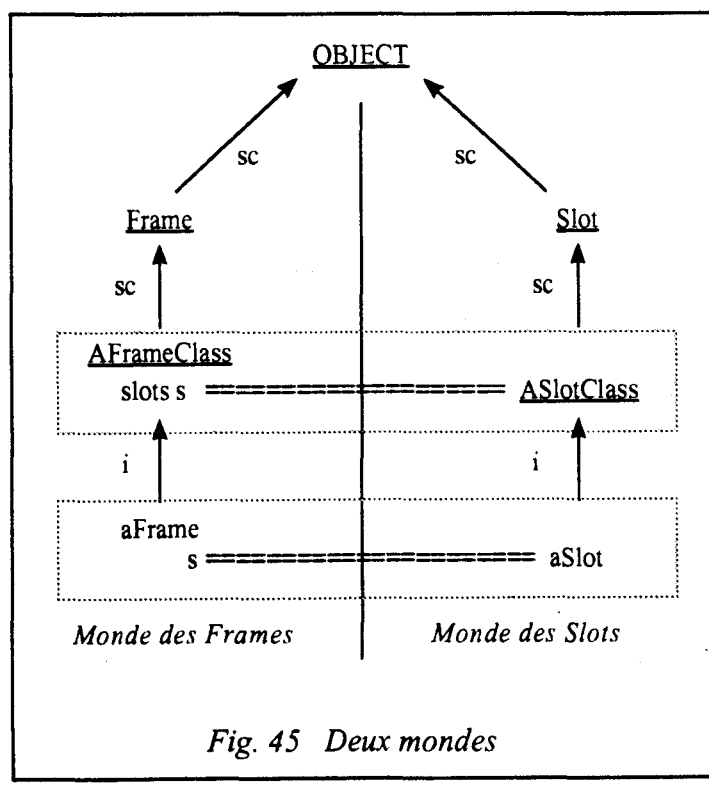


Fig. 45 Deux mondes

4.2.2 Les slots sont des objets

Les facettes des slots peuvent être partitionnées en **facettes passives** et **facettes actives** (cf. Fig. 27 (page 80)). L'analyse orientée objet conduit à la définition des slots comme des objets, caractérisés par leurs :

- attributs = facettes passives
- méthodes = facettes actives.

Les slots sont alors définis par des classes qui définissent ces caractéristiques. La classe de slots la plus générale est la classe **Slot**. C'est une classe abstraite.

Les facettes $type, $domain, $interval sont partagées par toutes les instances d'une même classe de slots. Par conséquent, elles sont assimilées à des attributs de classe et appelées **facettes de classe** (cf. page 80). Les facettes de classe sont définies comme des attributs d'une classe de slots, par sa métaclasse. La métaclasse des classes abstraites de slots est appelée **Meta-R-Slot**, par analogie avec la métaclasse des classes abstraites du noyau de ROME : R-CLASS.

La classe Slot définit :

- l'attribut $val (facette d'instance)
- les méthodes qui implantent les réflexes
- les méthodes d'accès à la valeur du slot (*get*, *put*) qui appliquent les réflexes.

Il est important de noter que l'accès à la valeur du slot relève de la responsabilité de l'objet slot qui applique ses propres méthodes *get* et *put*. Ces méthodes contrôlent les contraintes sur la valeur et appliquent les réflexes. Elles correspondent à des stratégies particulières de lecture et d'écriture du slot.

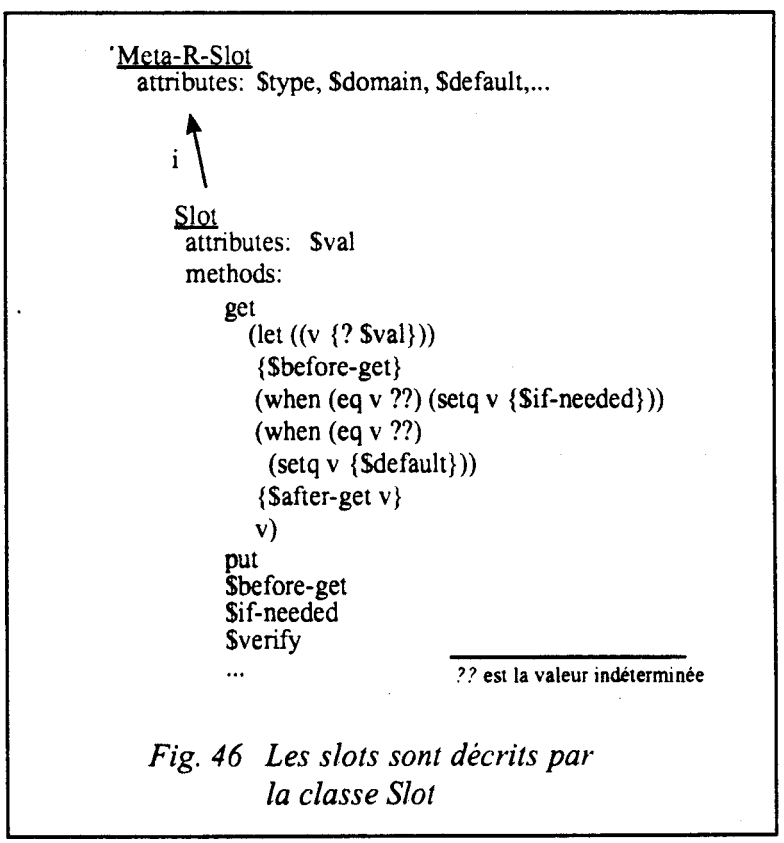


Fig. 46 Les slots sont décrits par la classe Slot

4.2.3 Les frames sont des agrégats d'objets slots

Les frames sont définis comme des objets, et donc décrits par des classes, dont la plus générale est une classe abstraite qui s'appelle **Frame**. Par conséquent, les frames en FROME disposent de méthodes. Leurs slots sont des objets décrits par des classes de slots. Chaque slot est associé à un attribut ROME, dont la valeur est l'objet slot correspondant. Au niveau de la classe de frames, chaque attribut se voit associer une classe de slots. Cette association est exprimée dans l'attribut *slots*, défini pour les classes de frames par la métaclasse **Méta-R-Frame**. C'est la métaclasse des classes abstraites de frames.

L'instanciation d'une classe de frames nécessite l'instanciation des classes de slots associées aux attributs, afin de construire l'instance de frame comme un agrégat des instances de slots. La méthode d'instanciation de base (*new*) est redéfinie pour les classes d'instanciation de frames, pour répondre à ce besoin. Elle est définie dans la métaclasse des classes d'instanciation de frames : **Méta-I-Frame**, sous-classe de Méta-R-Frame (et de I-CLASS).

Les méthodes d'accès aux attributs sont redéfinies par la classe Frame pour rediriger l'accès vers l'objet slot.

```
[aFrame ? s]      ⇒ [{? s} get]
[aFrame <- s v]   ⇒ [{? s} put v]
```

Fig. 47 Accès aux slots

La Fig. 48 montre un exemple de classe de frame (la classe `Appartement`) avec une classe de slots associée pour son slot `proprietaire`.

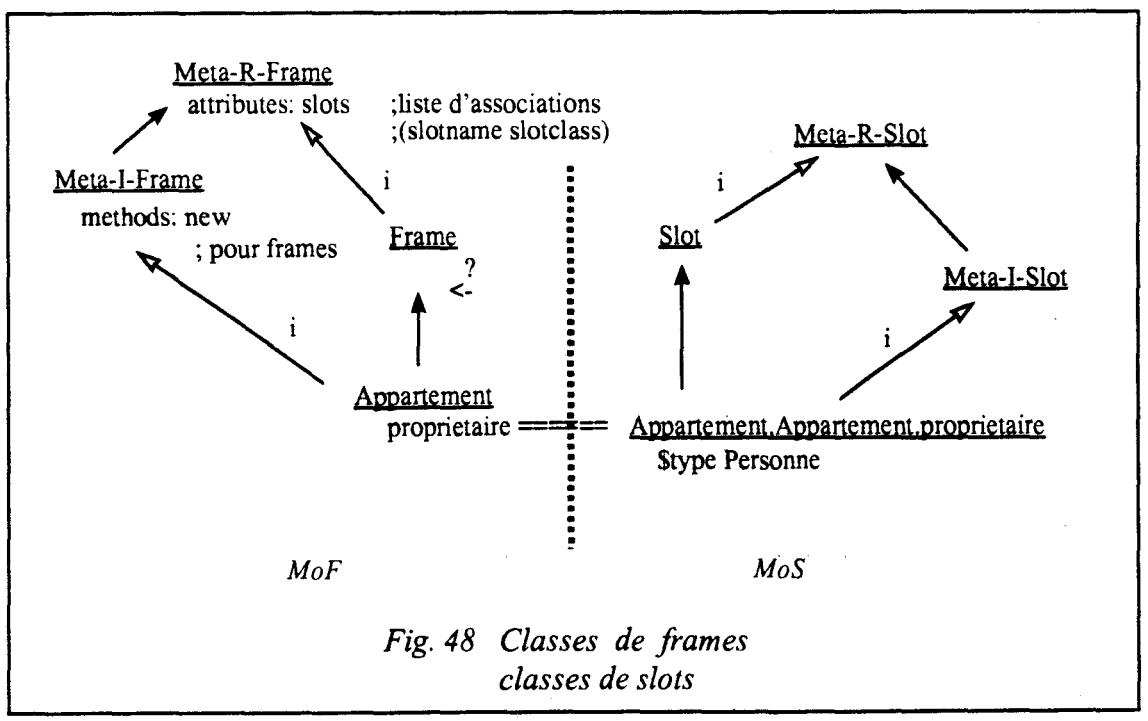


Fig. 48 Classes de frames
classes de slots

La création d'une instance de frame représentant un appartement, par exemple `mon-appart`, est réalisée par un envoi de message à sa classe :

```
[Appartement 'new 'mon-appart]
```

ce qui entraîne l'envoi du message suivant pour créer l'objet correspondant à son slot `proprietaire` :

```
[Appartement.Appartement.proprietaire 'new 'proprietairemon-appartg101]
```

La redirection de message de MoF vers MoS provoquée par un accès en écriture au slot `proprietaire` est illustrée Fig. 49.

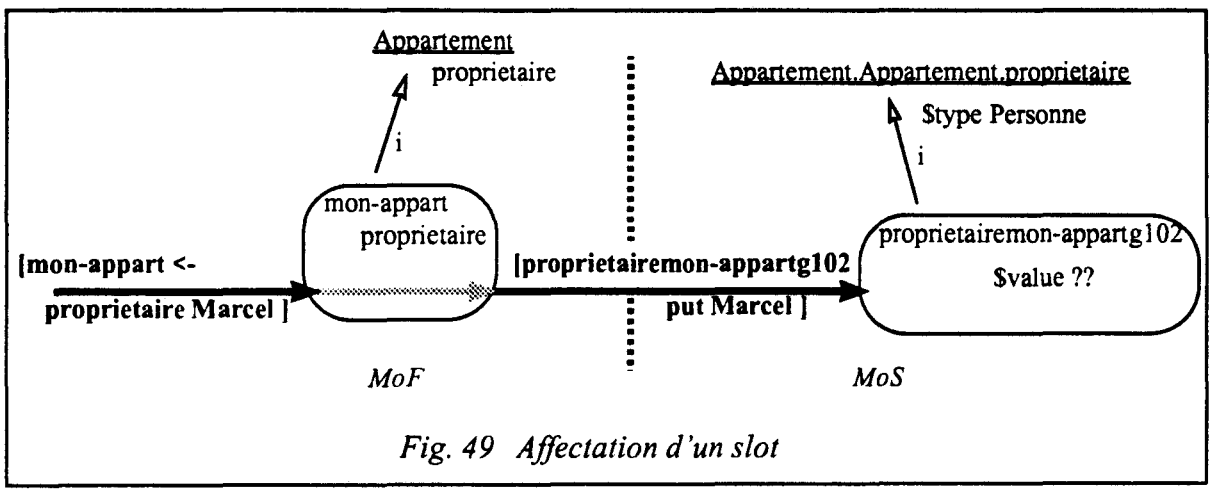


Fig. 49 Affectation d'un slot

4.2.4 Classes de frames, classes de slots et sous-classement

La création d'une classe de frames entraîne la création de classes de slots pour tous les slots qu'elle définit dans son attribut *slots*. Cette action particulière lors de la création d'une classe de frames est accomplie par la méthode d'instanciation de classes, redéfinie à cet effet par la classe des métaclasses de frames : **MétaMétaFrame**.

La création de la classe `Appartement` du paragraphe précédent est obtenue par un envoi de message à la classe Méta-I-Frame, qui applique sa méthode *new*, dans MoF. Il en résulte une série de messages, envoyés à Méta-I-Slot, avec pour but la création d'une classe pour chaque slot décrit, dans MoS, appelée `Appartement.Appartement.<slot>`[1] :

```
[Meta-I-Frame 'new 'Appartement 'supercl '(Frame)
 'slots '(proprietaire
                ($type Personne)
          etage ($type integer
                 $interval (0 20))
          jardin($type boolean)
          loyer ($type real
                 $interval (500 1000))
          charges
                ($type real
                 $if-needed (lambda()
                   (+ %entretien
                      (* 0.05 %loyer))))
          entretien
                ($type real
                 $default 50)
          loyerTCC
                ($type real
                 $if-needed (lambda()
                   (+ %loyer %charges))))]
```

⇒

```
[Meta-I-Slot 'new 'Appartement.Appartement.proprietaire 'supercl '(Slot)
 '$type 'Personne ...]

[Meta-I-Slot 'new 'Appartement.Appartement.etage 'supercl '(Slot)
 '$type 'integer
 '$interval '(0 20)...]

[Meta-I-Slot 'new 'Appartement.Appartement.jardin 'supercl '(Slot)
 '$type 'boolean ...]

[Meta-I-Slot 'new 'Appartement.Appartement.loyer 'supercl '(Slot)
 '$type 'real
 '$interval '(500 1000)...]

[Meta-I-Slot 'new 'Appartement.Appartement.charges 'supercl '(Slot)
 '$type 'real
 '$if-needed '(lambda() (+ (entretien (* 0.05 %loyer)))) ...]
```

1. Tout objet en FROME a un nom, qui est un symbole lisp. Les noms des classes et instances de slots sont générés automatiquement.

...

Dans un langage à base de classes, la spécialisation est exprimée par la création de sous-classes. La sous-classe spécialise sa superclasse par l'ajout de nouveaux attributs et de nouvelles méthodes ou par la redéfinition de méthodes héritées. Le sous-classement des classes de frames autorise de plus la spécialisation par affinement de slots. Le sous-classement étant un moyen d'expression de l'affinement, l'affinement des slots entraîne :

(1) le sous-classement des classes de slots (dans MoF)
(2) la vérification des règles d'affinement (dans MoS)

(1) : Les sous-classes de slots sont générées automatiquement lors de la création d'une sous-classe de frames. Les calculs de sous-classement et les redirections de messages vers MoS sont réalisés par la méthode *new* de Méta-I-Frame ou de Méta-R-Frame[1], définie par leur classe MétaMétaFrame. En effet, cette partie tombe sous la responsabilité de MoF, car le sous-classement des classes de slots est déduit de celui des classes de frames. Le sous-classement des classes de slots est réalisé selon la règle suivante :

Soit	*F une classe de frames définissant le slot s associé à la classe de slots S,*
et	*F' une classe de frames sous-classe de F, affinant le slot s.*
Alors	*la classe de slots S' associée à s dans F' est créée comme une sous-classe de S.*

Un exemple de sous-classement avec affinement de slots est donné dans le message suivant :

```
[Meta-I-Frame new AppartEtageSup superclasses (Appartement)
 slots ( etage  ($interval (1 20))
         jardin ($domain (false))
         fraisAscenseur
                ($type real
                 $if-needed (lambda()
                   (* %etage 12.50)))
         charges
                ($if-needed (lambda()
                   (+ (super)
                     %fraisAscenseur))))]
```

Les messages suivants sont générés pour la création de nouvelles classes de slots :

```
[Meta-I-Slot 'new 'Appartement.AppartEtageSup.etage
 'supercl '(Appartement.Appartement.etage)
 '$interval '(1 20)]

[Meta-I-Slot 'new 'Appartement.AppartEtageSup.jardin
 'supercl '(Appartement.Appartement.jardin)
 '$domain '(false)]
```

1. Selon qu'il s'agisse d'une classe d'instanciation de frames ou d'une classe abstraite.

```
[Meta-I-Slot 'new 'AppartEtageSup.AppartEtageSup.fraisAscenseur
 'supercl '(Slot)
 '$type 'real
 '$if-needed '(lambda() (* %etage 12.50))]

...
```

Si la sous-classe de frames définit de nouveaux slots (comme ici `fraisAscenseur`), la classe de slots associée est créée comme une sous-classe de la classe Slot. La classe Slot est la superclasse par défaut s'il ne s'agit pas d'un slot affiné et si aucune autre superclasse n'est spécifiée pour ce slot. En effet, l'utilisateur peut spécifier explicitement une (ou plusieurs) superclasses d'un slot, lors de sa description dans une classe, à l'aide de la facette $supercl. La classe MultiSlot par exemple, peut être utilisée comme superclasse des slots multivalués.

Après le calcul du sous-classement dans MoF, la tâche de création effective de la classe de slots est redirigée vers MoS à travers des messages à Méta-I-Slot.

(2) : La vérification des règles d'affinement :
L'affinement d'un slot entraîne l'ajout de nouvelles contraintes, la restriction du domaine des valeurs possibles, etc. Le respect des règles d'affinement doit être assuré :

- Le $type d'un slot affiné doit être un sous-type du $type hérité.
- Les valeurs des facettes $domain et $interval doivent être incluses dans les valeurs héritées des superclasses.

Ces règles d'affinement contrôlent la création d'une classe de slots comme sous-classe d'une autre classe de slots. Par conséquent, elles sont vérifiées lors de la création d'une classe de slots dans MoS. Cette vérification est prise en charge par une méthode d'instanciation spécifique, définie pour les métaclasses de slots par leur classe : **MétaMétaSlot**.

Les responsabilités respectives de MoF et MoS concernant le sous-classement sont illustrées dans la Fig. 50.

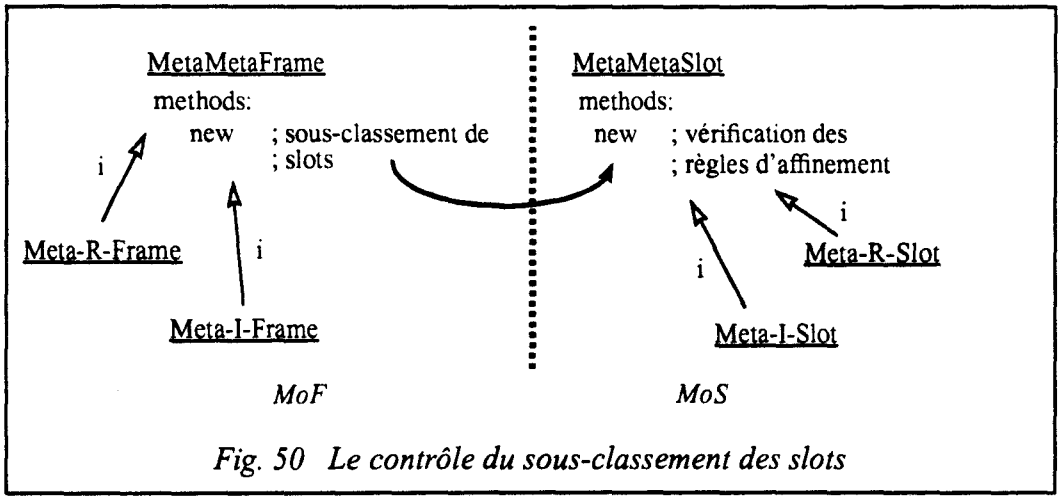


Fig. 50 Le contrôle du sous-classement des slots

Le résultat de la création de sous-classes par les messages de la page précédente est illustré par la Fig. 51.

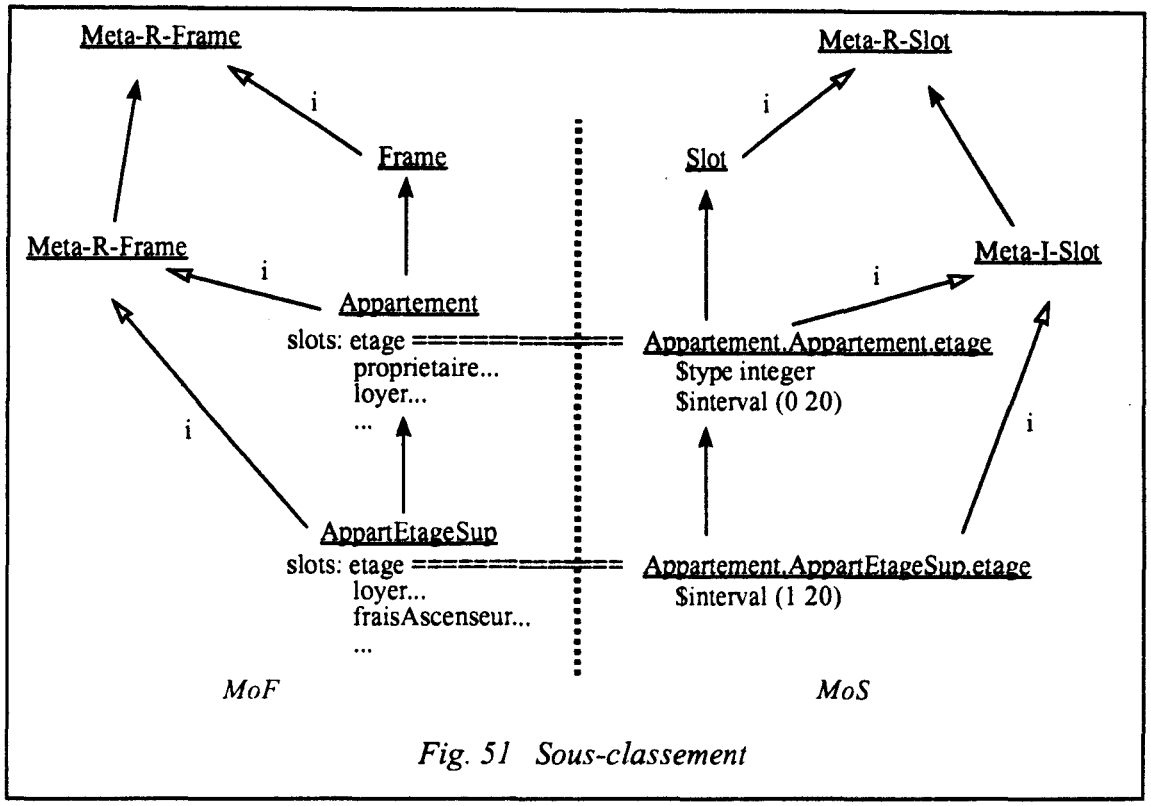


Fig. 51 Sous-classement

4.3 L'héritage des notions spécifiques de ROME

La conception d'un langage de frames suivant l'approche exposée en 4.2 préserve toutes les notions spécifiques du langage orienté objet sous-jacent. Nous montrons dans les paragraphes 4.3.1 à 4.3.3 que l'extension FROME hérite de l'Héritage Multiple, la Représentation Multiple et Evolutive de son langage sous-jacent ROME. Les notions de Représentation Multiple et Evolutive sont définies en ROME pour une classe d'objets particulière : R-OBJECT.

L'héritage de ces notions pour les frames est obtenu par la création de deux sous-classes spécifiques de R-OBJECT : R-Frame pour les frames et R-Slot pour les slots. Cette extension aisée est dûe à l'extensibilité, une des qualités majeures résultant de la conception orientée objet du système.

On examinera successivement l'Héritage Multiple, l'Evolution et la Représentation Multiple.

4.3.1 L'héritage multiple et les points de vue pour les frames et les slots

L'héritage multiple en FROME obéit aux mêmes principes que celui de ROME, comme nous l'avons exposé en 3.3.1. Le principe d'affinement, qui ne s'applique pas aux attributs de ROME[1], s'applique aux slots. Nous l'avons vu dans le cadre de l'héritage simple.

1. car ils ne s'affinent pas : ce sont de simples variables d'instance à la Smalltalk.

Les cas (1) et (2) (cf. Fig. 29 (page 83)) peuvent être interprétés de façon triviale pour les slots. Les instances de la classe (de frames) C3 ont un seul slot *s*. Cela correspond exactement à une instance de slot créée à partir de la classe de slots correspondante : celle associée à *s* de C0 pour (1) et celle associée à *s* de C1 pour (2).

Dans le cas d'un véritable héritage multiple de slots, la sélection de point de vue opère tant dans MoF que dans MoS. Une sélection de point de vue lors de l'accès à un slot est propagée de MoF vers MoS. La redirection s'exprime sous la forme suivante :

```
[aFrame ? (s as AFrameClass)]  ⇒
(let ((oslot {? (s as AFrameClass)}))
  [oslot (get as [AFrameClass slotClassFor? oslot])])
```

où *slotClassFor?*, demandé à AFrameClass renvoie la classe de slots associée à s dans AFrameClass. On est alors amené à distinguer deux cas. L'un correspond à l'interprétation des caractéristiques *s* en (3) comme des slots, et l'autre est un nouveau cas résultant de (1) et (2) dans le cas d'un affinement multiple d'un slot.

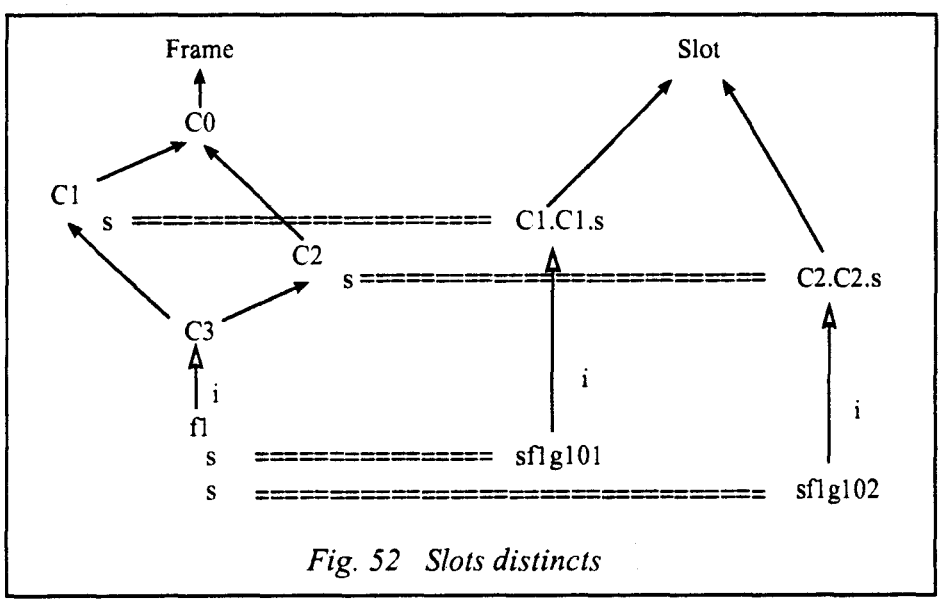


Fig. 52 Slots distincts

Cas I : Héritage multiple de slots homonymes

La version frame/slot de (3) est illustrée dans la Fig. 52. Les slots *s* sont définis par les classes indépendantes C1 et C2 et sont hérités tous les deux par C3. L'instance de frame f1 possède alors deux slots *s* distincts, correspondant à deux instances de slots de deux classes de slots distinctes. La sélection d'un point de vue permet d'en choisir un. Par exemple :

```
[f1 ? (s as C1)]
⇒ oslot = {? (s as C1)} = sf1g101
    [C1 slotClassFor? oslot] = C1.C1.s
⇒ [sf1g101 (get as C1.C1.s)]
```

La sélection de point de vue dans l'expression `(get as C1.C1.s)` n'a pas d'effet ici, car nous n'avons pas d'héritage multiple dans MoS. Le dernier message est alors équivalent à

```
[sf1g101 get].
```

Le cas I correspond au cas de l'héritage multiple de slots homonymes du chapitre précédent. La Fig. 53 reprend l'exemple de la Fig. 35 (page 90) pour montrer les instances distinctes, `contactf1g102` et `contactf1g103`, correspondant aux slots distincts `contact` hérités par la classe `ReservedForFriend`. Le message

```
[f1 '? '(contact as reserved)]
 ⇒
 [contactf1g103 'get]
```

demande au frame `f1` le numéro de téléphone de la personne qui réserve le logement.

Comme nous avons affaire à des slots distincts, l'affinement de chacun d'eux correspond à un sous-classement indépendant dans MoS

.

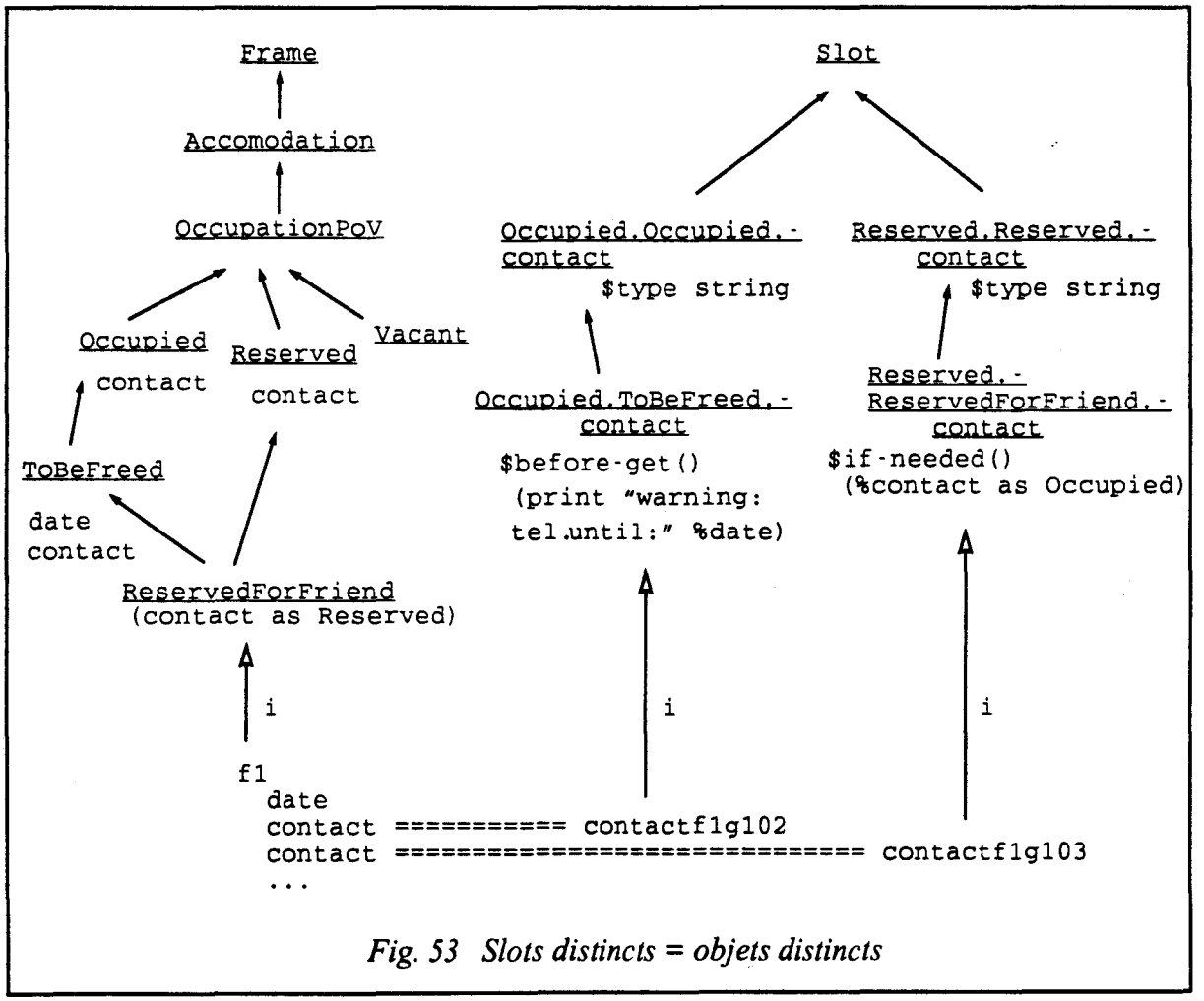


Fig. 53 Slots distincts = objets distincts

Cas II : Redéfinitions multiples d'un même slot

Dans le cas d'un affinement multiple du même slot, nous obtenons un nouveau cas de figure combinant les cas (1) et (2). Ce cas est représenté dans la Fig. 54. Les classes C1 et C2 spécialisent toutes les deux le slot *s* hérité de la classe C0. Il s'agit bien d'un seul slot *s*, affiné par deux sous-classes, selon le principe d'affinement. C3 hérite un seul slot *s*, qui doit

satisfaire toutes les contraintes héritées, aussi bien celles ajoutées par C1 que celles ajoutées par C2, par respect du **principe de multiplicité des contraintes** (cf. 3.3.1). On effectue une **fusion**, qui consiste en une intersection des facettes contraignantes ($domain, $interval, etc.). La classe de slots correspondant à cette fusion est créée (C0.C3.s) et associée au slot dans la classe de frames C3.

Comme dans le cas de l'héritage simple, le sous-classement des slots hérités est contrôlé par MoF, tandis que l'opération de fusion, qui fait intervenir la vérification des règles d'affinement, est effectuée dans MoS.

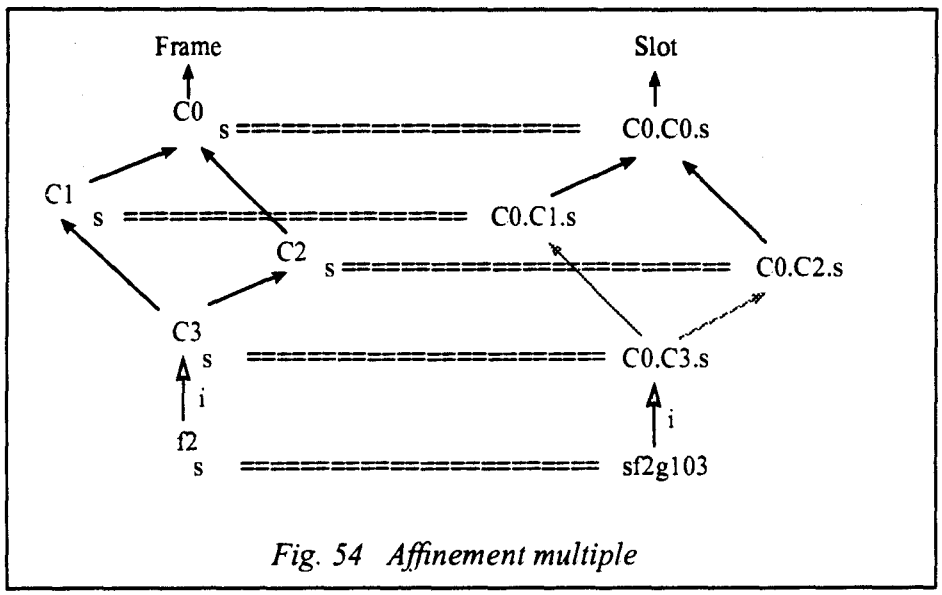


Fig. 54 Affinement multiple

La fusion concerne les facettes contraignantes, mais ne concerne pas les facettes non-contraignantes. Le cas II introduit l'héritage multiple des facettes non-contraignantes dans le graphe des classes de slots, dû à l'affinement multiple. La propagation d'une sélection de point de vue de MoF vers MoS entraîne maintenant une sélection parmi les réflexes hérités. Nous avons vu en 3.3.3 que la sélection d'un point de vue permet de déclencher des réflexes sous un point de vue correspondant à une *interprétation* particulière d'un slot. L'exemple de la Fig. 55 montre que cette sélection de point de vue est implantée comme une propagation de MoF vers MoS et résulte en une application de *méthode* (implantation de réflexe) sous un point de vue dans MoS. La sélection d'un point de vue sur f1 est propagée sur le slot interrogé, et résulte en une application de réflexe selon le point de vue de la classe de slots associée :

```
[f1 '? '(contact as ToBeReplaced)]
 ⇒
 oslot = {?(contact as ToBeReplaced)} = contactf1g102
 [ToBeReplaced slotClassFor? oslot]
  = Occupied.ToBeReplaced.contact
 ⇒
 [contactf1g102 (get as Occupied.ToBeReplaced.contact)]
 ⇒
 {($before-get as Occupied.ToBeReplaced.contact)}
 ⇒
 Warning: a proposal has to be made before: 2 - 15 - 1993 to this number:
 = 20.30.40.50
```

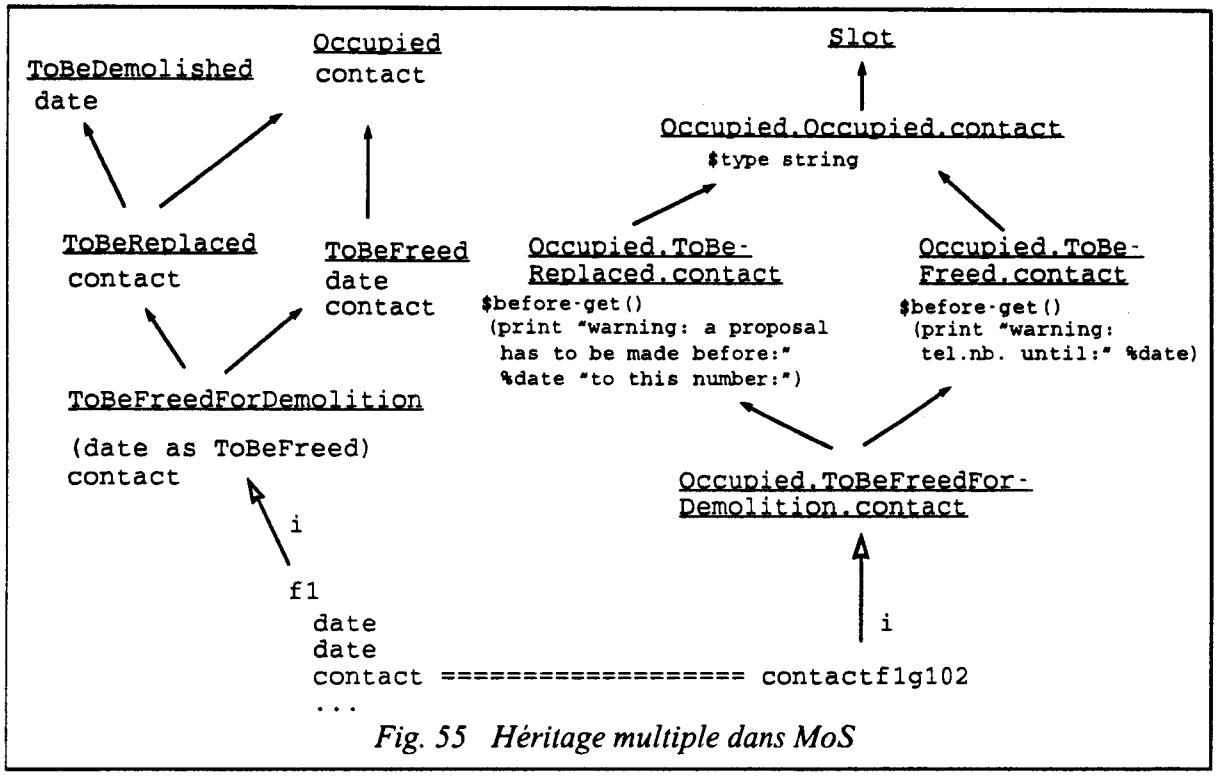


Fig. 55 Héritage multiple dans MoS

Rien n'empêche la spécialisation d'un slot dans une classe qui hérite de plusieurs affinements de ce slot. Par défaut, la classe de slots fusion des affinements lui est associée, cumulant toutes les contraintes et héritant toutes les facettes non-contraignantes. Des affinements supplémentaires restent néanmoins possibles, par ajout de nouvelles contraintes, de nouvelles facettes non-contraignantes ou la redéfinition de réflexes (ou valeurs par défaut) hérités. Dans la classe `ToBeFreedForDemolition`, on aurait pu redéfinir le slot `contact`, en redéfinissant le réflexe `$before-get`, de la manière suivante :

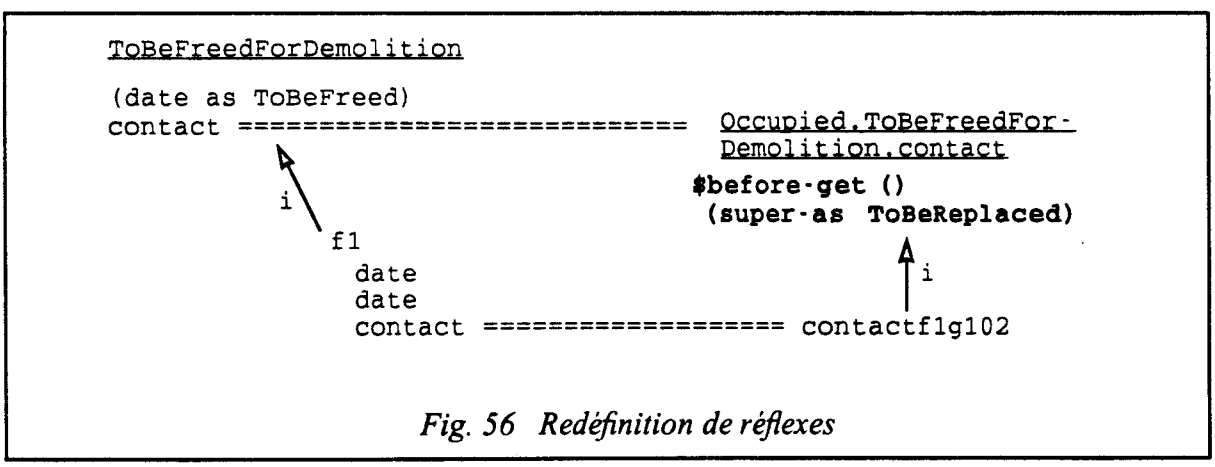


Fig. 56 Redéfinition de réflexes

La méthode `$before-get` de `Occupied.ToBeFreedForDemolition.contact` est maintenant la plus affinée, quel que soit le point de vue sur `f1`.

L'expression **(super-as <classe-de-frames>)** se transforme en **(super as <classe-de-slots>)**, avec <classe-de-slots> la classe de slots associée au slot dans la classe de frames. Ceci permet l'utilisation de "super as" expressions dans les réflexes, sans connaître les noms des classes de slots, qui sont générés automatiquement.

4.3.2 La représentation évolutive des frames

La représentation évolutive de ROME est la liaison dynamique d'un objet avec des classes de représentation potentielles. Initialement, l'objet est instance de sa classe d'instanciation. Il restera essentiellement un représentant de cette classe, qui a servi de moule à sa création. Sa description peut ensuite évoluer de façon incrémentale, en le rattachant à des sous-classes de sa classe d'instanciation. Les objets capables d'évoluer de cette manière sont décrits en ROME par la classe R-OBJECT. Cette classe définit les opérations qui lient un objet à une classe ou l'en détachent, à travers les méthodes d'évolution associées aux sélecteurs *r+* et *r-*.

Pour disposer de frames évolutifs à la manière des objets évolutifs de ROME, on utilise l'héritage multiple et l'affinement. Les frames évolutifs sont décrits par la classe **R-Frame**, sous-classe de R-OBJECT et de Frame. La classe R-Frame hérite donc en particulier des méthodes primitives d'évolution.

L'évolution d'un frame peut entraîner l'acquisition de nouveaux slots, définis par des classes dont le frame devient un représentant. Elle peut également résulter en un affinement de slots que le frame possède déjà. Nous avons vu que l'affinement de slots est exprimé par un sous-classement de classes de slots. L'affinement d'une instance de slot s'exprime alors naturellement par l'évolution vers une sous-classe de slots.

Pour rendre compte des particularités de l'évolution des frames, les méthodes *r+* et *r-* sont redéfinies dans R-Frame :

(1) **MoF** :
r+ doit
- instancier de nouveaux slots
- faire évoluer les slots existants affinés, en envoyant des messages *r+* aux instances correspondantes.

r- doit
- détruire les slots perdus lors de l'évolution
- propager *r-* vers les slots qui doivent évoluer

De façon symétrique, la classe **R-Slot** décrit les "slots évolutifs". Cette classe est créée comme une sous-classe de R-OBJECT et de Slot. La méthode associée à *r+* est redéfinie pour les slots :

(2) **MoS** : *r+* doit vérifier les nouvelles contraintes sur la valeur présente du slot, en respect des règles d'affinement.

La méthode *r-* n'est pas redéfinie pour les slots, car les contraintes des superclasses sont moins restrictives que celles de la classe d'appartenance avant l'évolution. Si les contraintes sont satisfaites avant l'évolution, elles le seront donc après, et aucune vérification supplémentaire n'est nécessaire.

Reprenons l'exemple Fig. 36 (page 91) et montrons la propagation de l'évolution de MoF vers MoS. Si nous avons le logement `f3`, représentant des classes `ToBeDemolished` et `ToBeFreed`, son évolution vers la classe `ToBeFreedForDemolition` est déclenchée par le message suivant :

```
[f3 'r+ 'ToBeFreedForDemolition]
```

Cette évolution entraîne :

- l'instanciation d'un nouveau slot `newAccommodation` (création de l'objet slot)
- l'évolution du slot `date` (*as ToBeFreed*) par le message `[datef3g106 'r+ 'ToBeFreed.ToBeFreedForDemolition.date]`, vérifiant la nouvelle contrainte exprimée dans la facette `$verify`.
- l'évolution du slot `contact` par le message `[contactf3g105 'r+ 'Occupied.ToBeFreedForDemolition.contact]`.

4.3.3 La représentation multiple des frames

La représentation multiple des frames (cf. 3.3) est définie par la classe des frames évolutifs, R-Frame. Elle entraîne les mêmes types de conflits entre slots que l'héritage multiple. Le cas des slots homonymes (cas I) est similaire au cas rencontré en héritage multiple. Le cas d'affinements multiples d'un slot (cas II) introduit **la représentation multiple des slots**. Ces affinements sont décrits dans des sous-classes de slots et l'objet slots y est rattaché par des liens multiples de représentation.

Reprenons l'exemple de 3.3.3, où le logement `f1` est représentant multiple de `TenantPoV` et `OwnerPoV`. L'affinement multiple du slot `monthlyRent`, défini dans `FinancePoV`, correspond à des sous-classes de slots multiples décrivant chacune un affinement particulier. Le slot `monthlyRent` du logement `f1` est un objet représentant des classes de slots `FinancePoV.OwnerPoV.monthlyRent` et `FinancePoV.TenantPoV.monthlyRent`, toutes deux sousclasses de la classe d'instanciation du slot, `FinancePoV.FinancePoV.monthlyRent`. Nous avons vu que le principe de multiplicité des contraintes garantit la vérification des contraintes imposées sous tous les points de vue. Comme la vérification de ce principe relève des responsabilités de MoS, son implantation dans le cadre de la représentation multiple des slots passe par la redéfinition de la méthode *put* dans la classe R-Slot :

```
R-Slot
 methods:
 put  (λ (val)
       (mapc
          (λ(rc) {(check-constraints as rc) val})
          {? rclasses})
      {$before-put val}
      {<- $value val}
      {$after-put val})
```

Enfin, la propagation de la sélection de point de vue de MoF vers MoS fonctionne de la même manière qu'en héritage multiple.

```
[f1 '? '(monthlyRent as OwnerPoV)]
 ⇒
 oslot = {?(monthlyRent as OwnerPoV)} = monthlyRentf1g104
 [OwnerPoV slotClassFor? oslot]
  = FinancePoV.OwnerPoV.monthlyRent
 ⇒
 [monthlyRentf1g104 (get as FinancePoV.OwnerPoV.monthlyRent)]
 ⇒
 {($before-get as FinancePoV.OwnerPoV.monthlyRent)}
 ⇒
 Rent of 2275 received (1 - 3 - 1992)
 = 2275
```

4.3.4 Variante à la représentation multiple : les mixins de représentation

Comme variante à la représentation multiple, nous définissons une classe particulière de mixins. Classiquement, les mixins sont des classes qui contiennent un comportement (ou des attributs) particulier(s), qui peuvent être utilisés comme ingrédients à mélanger avec des classes plus "conceptuelles", afin de spécialiser ces dernières en y ajoutant le comportement/ les attributs en question. Le mélange est obtenu par héritage multiple. Les mixins ne doivent pas être instanciées : ce sont des classes abstraites. Exemple (tiré de [Bra&90]):

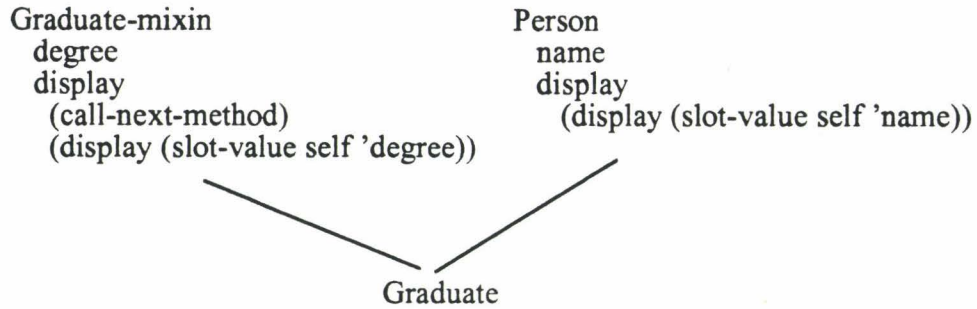


Fig. 57 Héritage multiple avec mixin

Dans cet exemple (la syntaxe est celle de CLOS), la classe Graduate est définie comme sous-classe de la mixin Graduate-mixin et de la classe Person. La linéarisation des ancêtres en CLOS a pour effet l'application de la méthode display de Graduate-mixin, qui appelle la suivante, c. à d. celle de Person, lors d'un envoi de message display a une instance de Graduate[1]. La méthode display enrichie affiche le nom d'une personne suivi de son grade.

Le type de mixin que nous proposons est différent, en ce qu'il n'est pas mélangé par héritage multiple, mais par représentation multiple. La représentation multiple classique de ROME [Car89] obéit à une contrainte de convergence sous la classe d'instanciation, qui garantit la cohérence de représentation d'un objet. L'objet ne peut être représentant direct que des sous-classes strictes[2] de sa classe d'instanciation. Ceci empêche par exemple une instance de la classe Personne d'évoluer vers les classes Table ou Bicyclette[3].

Tout en étant une propriété légitime pour la représentation des connaissances, la contrainte de convergence empêche en même temps l'enrichissement des objets avec des comportements particuliers (éventuellement temporaires), définis dans des classes utilitaires. De tels enrichissements relèvent plutôt d'un souci PPO, et peuvent servir notamment aux buts de mise au point, de trace, et à d'autres traitements particuliers.

Une classe d'objets (ROME) supportant un tel enrichissement peut être définie (de manière minimale) comme dans la Fig. 58. Le mixage est opéré par l'application de la méthode `mix+`. Le rejet du mixin est obtenu par la méthode `mix-`.

La classe des frames supportant les mixages est obtenue directement par héritage multiple :

```
[meta-r-frame 'new 'rframe+mix 'supercl '(r-object+mix rframe)]
```

Les classes mixins à définir doivent être identifiées en tant que telles et sont donc

1. Clairement, la mixin ne doit pas être instanciée ici car sa méthode display provoquerait une erreur
2. Aucune de leurs superclasses ne doit être indépendante de la classe d'instanciation de l'objet.
3. ...si toutefois ces dernières sont définies comme classes indépendantes de la première.

```
On définit une classe d'objets (r-object+mix) acceptant des mixins de
représentation, par l'application de la méthode mix+. Ces mixins ne
sont pas soumis à la contrainte de convergence sous la classe
d'instanciation des objets. La seule contrainte pour le paramètre est
d'être classe de mixin. Les objets sont reliés à leur(s) classe(s) de
mixin par r+d (r+ direct sans test de convergence). Ces classes font
donc partie de la liste des rclasses de l'objet (attribut rclasses).
Elles sont également stockées sur une liste spécifique (attribut
mixclasses).

[r-class 'new 'r-object+mix 'supercl 'r-object
 'attributes '(mixclasses)
 'methods '(mix+ (lambda(c)
                   (cond ((not (memq c (rome-obj)))
                          (applymeth 'rome-error c " not an object"))
                         ((not [c 'repg? 'mixin-class])
                          (applymeth 'rome-error c "not a mixin"))
                         (t (applymeth 'before-mix+ c)
                            (applymeth '<- 'mixclasses
                             (cons c
                               (let ((mc (applymeth '? 'mixclasses)))
                                (if (eq mc '??) () mc))))
                            (applymeth 'r+d c)
                            (applymeth `(after-mix+ as ,c) c)
                            self)))
            mix- (lambda(c)
                   (cond ((not (memq c (rome-obj)))
                          (applymeth 'rome-error c "not an object"))
                         ((not (memq c (applymeth '? 'mixclasses)))
                          (applymeth 'rome-error c
                              " not in mixclasses of " self))
                         (t (applymeth `(before-mix- as ,c) c)
                            (applymeth '<- 'mixclasses
                                (remq c (applymeth '? 'mixclasses)))
                            (applymeth 'r-d c)
                            (applymeth 'after-mix- c)
                            self)))
            before-mix+ (lambda(c))
            after-mix+ (lambda(c))
            before-mix- (lambda(c))
            after-mix- (lambda(c)))
 'selectors '(mix+ mix+ mix- mix-)]
```

Fig. 58 Objets prêts au mixage

nécessairement représentantes de la métaclasse prévue à cet effet `mixin-class`, sous sa définition la plus simple :

```
[i-class 'new 'mixin-class 'supercl 'r-class]
```

et la classe de frames correspondante :

```
[metametaframe 'new 'mixin-frameclass 'supercl '(meta-r-frame mixin-class)]
```

Notons que les instances de `mixin-class` et `mixin-frameclass` sont des classes abstraites.

Une utilisation des mixins de représentation pour les frames est proposée dans la partie II du mémoire.

4.4 Extensions spécifiques à FROME

La possibilité de définir de nouvelles classes et méta-classes dans un langage à noyau méta-circulaire a été mise en avant par de nombreux auteurs (cf. 2.5.2). FROME a été obtenu par la définition de nouvelles classes et méta-classes en ROME. Le langage obtenu, spécialisé pour la représentation multiple et évolutive des frames, est lui-même extensible. L'implantation d'un module de règles, appelé FROMER, obtenue par méta-programmation objet, est décrite dans [Ham91]. Les fonctionnalités de filtrage spécifiques à FROME, sont obtenues également par extension méta-programmée.

4.4.1 Nouvelles classes de slots = nouvelles spécificités

Ferber, dans l'implantation orientée objet de MERING [Fer83], a montré que la réification des attributs permet de définir toutes sortes d'attributs spécialisés.

Un premier exemple de slot spécialisé pour FROME a été donné par la classe des slots à représentation multiple et évolutive RSlot.

L'implantation des slots comme des instances de classes de slots nous a permis de la même façon de définir facilement différents types de slots multivalués : Multislot avec une sémantique de multi-ensemble, et Setslot, avec une sémantique d'ensemble de valeurs. Chaque type de slots particulier réclame ses modes d'accès particuliers, définis comme des méthodes spécialisées dans la classe de slots correspondante. Ainsi, l'utilisateur lui-même peut définir de nouveaux types de slots, incluant par exemple de nouvelles facettes[1].

En guise d'exemples, nous avons expérimenté de nouvelles facettes de classe :

- $value, fixant la valeur d'un attribut pour toutes les instances de la classe
- $coref, qui indique des relations d'égalité de valeurs entre slots
- $min et $max, cardinalités minimum et maximum d'un slot multivalué (cf. Fig. 59).

Nous pensons également à un type de slot descriptif DescriptiveSlot, avec de nouvelles facettes d'instance, en plus de la facette $val, pour restreindre localement la valeur d'un slot, à la manière des langages à prototypes :

- $type, $domain, $interval ...

D'autres types de slot peuvent être HistorySlot, qui conserve ses valeurs antérieures, ou DiscriminantSlot, utilisé pour la classification. Différents types de relations peuvent être obtenus par des classes de slots particulières.

Notons la définition de slots dédiés aux règles dans FROMER : Variable, Argument, et des facettes particulières d'attachement de règles $if-needed-rule, $after-put-rule, ... [Ham91].

1. La sémantique de spécialisation des nouvelles facettes doit être précisée dans une méthode de méta-classe, qui opère le sous-classement des classes de slots en vérifiant l'affinement.

```
[i-class 'new 'metametaminmax-slot 'supercl 'metametaslot
 'methods '(inter$min (lambda(v1 v2)      ; règle d'affinement pour $min
             (cond ((eq v1 '??) v2)
                   ((eq v2 '??) v1)
                   (t (if (> v1 v2) v1 v2))))
            inter$max (lambda(v1 v2)      ; règle d'affinement pour $max
             (cond ((eq v1 '??) v2)
                   ((eq v2 '??) v1)
                   ((eq v1 '+inf) v2)
                   ((eq v2 '+inf) v1)
                   (t (if (< v1 v2) v1 v2 ))))]

[metametaminmax-slot 'new 'meta-minmax-slot 'supercl 'meta-r-multislot
 'pfacets '($min $max)
 'methods '($min= (lambda(m) (:$min m))
            $max= (lambda(m) (:$max m)))]

[metametaminmax-slot 'new 'meta-i-minmax-slot    ; il faut une i-classe
 'supercl '(meta-i-multislot meta-minmax-slot)] ; pour les nouveaux slots

[meta-minmax-slot 'new 'minmax-slot 'supercl 'setslot
 'methods '(put (lambda(v) (mapc (lambda(c)
                 (let ((min [c '? '$min])
                       (max [c '? '$max]))
                  (cond ((and (not (eq max '+inf))
                              (> (length (if (atomp v) (list v) v)) max))
                         {rome-error "number of values for "
                          {$name} " must not exceed " max})
                        ((< (length (if (atomp v) (list v) v)) min)
                         {rome-error "number of values for "
                          {$name} " must not be inferior to " min})
                        (t ())))))
                  {rclasses})
                  (super))
            addval (lambda(v)…
            remval (lambda(v)…)))]


 ; son utilisation :
[meta-i-frame 'new 'personne 'supercl 'frame
  'slots '(boulot ($type string $supercl minmax-slot $max 5))]

[meta-i-frame 'new 'travailleur 'supercl 'personne
  'slots '(boulot ($min 1))]
```

Fig. 59 Exemple : MinMaxSlot avec des facettes de cardinalité $min et $max

4.4.2 Implantation des filtres

Les filtres étant identifiés par des classes, les classes qui les décrivent sont des *métaclasses*, dont la plus générale est `Filter`. Elle décrit les filtres conjonctifs, rémanents (car une classe est rémanente par défaut). Ensuite nous distinguons `OrFilter` — la classe des filtres disjonctifs —, `OrSubFilter` — celle des sous-filtres d'un filtre disjonctif—, et `VolatileFilter` décrivant les filtres volatils. Par héritage multiple nous obtenons de plus `VolatileOrFilter` et `VolatileOrSubFilter`.

Les requêtes de filtrage (`match`) sont traduites par des envois de messages de création de filtres adressés à une de ces classes selon le type de requête (requête conjonctive ou disjonctive, avec filtre volatil ou rémanent).

La méthode de création d'un filtre est définie par sa métaclasse[1]. Elle crée la classe filtre et lui envoie le message "filter". La méthode associée à "filter" est définie ou redéfinie dans chaque (méta)classe de filtre pour réaliser le comportement de filtrage approprié.

En résumé :

- **`Filter`**
  méthode `filter` : recueil des instances et spécialisation de celles-ci
- **`VolatileFilter`**
  méthode `filter` : recueil des instances et spécialisation de celles-ci
  auto-destruction de la classe
- **`OrFilter`**
  méthode `filter` : retourne ses représentants
  envoi de messages pour la destruction des sous-filtres volatiles
- **`VolatileOrFilter`** :
  méthode `filter` : retourne ses représentants
  envoi de messages pour la destruction des sous-filtres
  auto-destruction
- **`OrSubFilter`** :
  méthode `filter` : sélection des instances appartenant à toutes les
  superclasses directes sauf à la classe filtre disjonctif englobant
  recueil de celles satisfaisant le sous-filtre
- **`VolatileOrSubFilter`** :
  méthode `filter` : (super as OrSubFilter)
- **`MetaFilter`** :
  méthode `new` : création d'une classe de filtre
  envoi du message `filter` à la classe créée
  méthode `new*` : génération d'un nom temporaire
  application de la méthode `new` avec le nom généré
- **`MetaOrFilter`** :
  méthode `new` : création d'une classe de filtre disjonctif
  création de ses sous-filtres par envois de messages
  envoi de message `filter` à la classe créée.

1. ...qui est une méta-métaclasse.

- **`MetaOrSubFilter`** :
  méthode `new` : création d'une classe de sous-filtre d'un filtre disjonctif
  envoi de message `filter` avec en paramètre la classe du filtre disjonctif à exclure des classes parent à prendre en compte.

La réalisation du filtrage est décrite en détail dans [Ver93]. Les différentes métaclasses et méta-métaclasses de filtres sont représentées schématiquement à la Fig. 60.

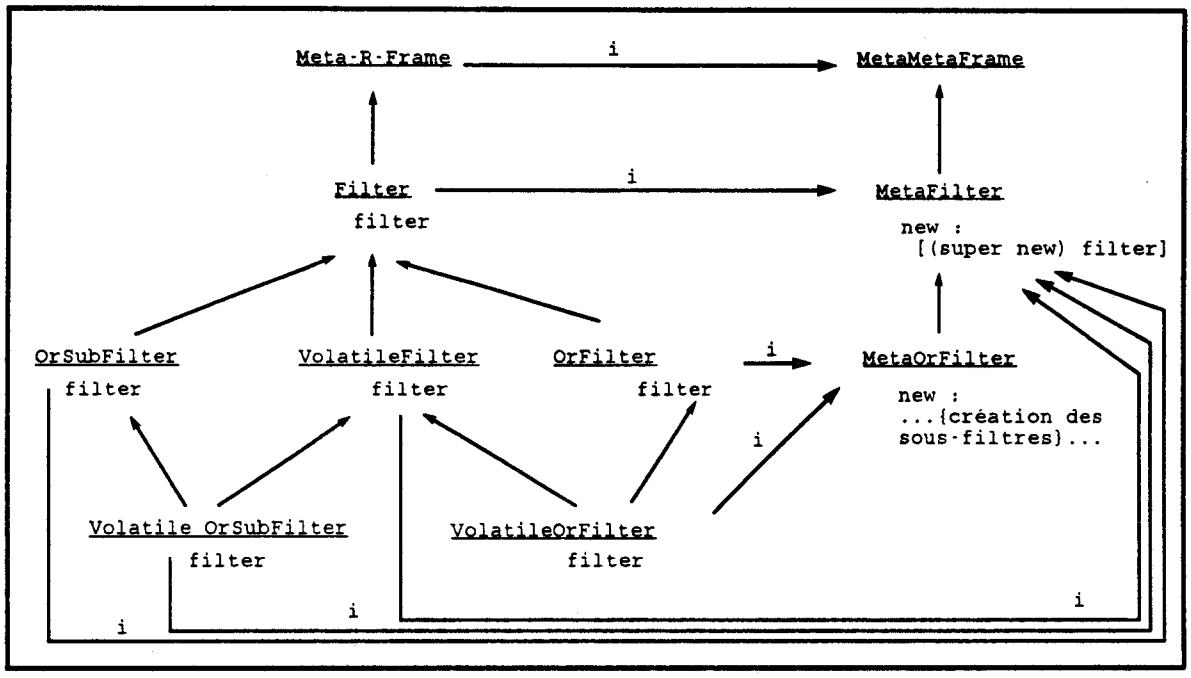


Fig. 60: Les métaclasses de filtres

Exemple :

La requête qui sélectionne les appartements à libérer (cf. 3.5.2.3)

```
(match '(ToBeFreed)
 'not '(Reserved ToBeDemolished)
 'where '(date ($verify (lambda(d)
                [d '< [date 'new* 'day 1 'month 7 'year 1993]] ))))
```

est traduite par le message suivant :

```
[VolatileFilter 'new* 'superclasses '(ToBeFreed)
 'excludes '(Reserved ToBeDemolished)
 'slots '(date ($verify (lambda(d)
                [d '< [date 'new* 'day 1 'month 7 'year 1993]])))) ]
```

La méthode `new*` de `VolatileFilter` crée une classe dont le nom est généré automatiquement, qui représente le filtre volatil et lui envoie le message `filter`.

Le `not` et le `where` sont transposés respectivement en `excludes` et `slots`, hérités de `MetaFrame`, qui définissent les classes exclues et les slots contraints par la classe de filtre.

Le filtrage (méthodes `filter`) procède par tentative de spécialisation d'instances vers la classe de filtre. Cette solution simple exploite le processus de spécialisation existant en Frome

reposant lui-même sur les capacités de représentation multiple et évolutive.

La partie *do* d'une requête est réalisée par l'envoi d'un message "after-filter" à chaque instance filtrée. La méthode after-filter correspondante est définie par la classe filtre. La traduction de cette option de la requête est donc la définition de cette méthode dans la classe filtre. Par exemple (cf. 3.5.2.4) :

```
(match '(Old)
 'where '(age ($interval (30 50))
              state ($domain ("bad" "very bad")))
 'do '({r+ ToBeDemolished}))
```

est traduite en :

```
[VolatileFilter 'new* 'superclasses '(Old))
 'slots '(age ($interval (30 50))
              state ($domain ("bad" "very bad")))
 'methods '(after-filter (lambda(){r+ ToBeDemolished}))]
```

4.4.3 La classification d'instances

Nous décrivons les principes d'implantation des différentes approches primitives de classification évoquées en 3.7 : la classification par attachement de réflexes, l'identification de slots discriminants et l'utilisation et de filtres.

4.4.3.1 Classification par réflexes

Nous avons distingué deux approches pour programmer la classification des instances à l'aide de réflexes : la programmation par rattachement explicite de réflexes et la programmation générique en identifiant des slots discriminants.

Réflexes explicites

La classification par programmation explicite consiste à attacher des réflexes aux attributs considérés déterminants pour la décision d'appartenance à une (sous-)classe. Dans notre implantation, ces réflexes (*$after-put*) sont complétés par un "réflexe sur évolution", la méthode *after-r+*[1], qui permet d'itérer l'évolution d'une instance.

Nous illustrons cette approche par la redéfinition de certaines classes de logements. Elles expriment l'évolution des logements contrôlée par les valeurs de certains slots, l'appartenance simultanée à certaines classes, ou une combinaison de conditions. Ainsi, un logement occupé et à démolir est classifié automatiquement dans la classe des logements à remplacer (cf. aussi 3.3.4.2 et la Fig. 36). Dans le point de vue `StatePov`, le logement est automatiquement sous-classifié dans une des sous-classes selon la valeur du slot `age`. Un vieux logement est amené à la destruction ou à la rénovation (*$after-put* dans `Old`). Les réflexes *$before-put* sont utilisés ici pour assurer la cohérence et rejeter l'appartenance de logements à des classes auxquelles ils ne peuvent plus appartenir (`Recent`, `NotTooOld`).

```
[meta-r-frame 'new 'occupied 'supercl 'occupationpov
 'excl '(vacant)
```

1. Cette méthode est appelée après évolution, par la redéfinition de la méthode *r+g* de la classe rframe : 'methods '(r+g (lambda (c) (super) (applymeth 'after-r+)))

```
 'slots '(tenant ($type person )
          contact ($type string))
 'methods '(after-r+ (lambda()
                (when (and (applymeth 'repg? 'tobedemolished)
                     (not (applymeth 'repg? 'tobefreedfordemolition)))
                  (applymeth 'r+g 'tobefreedfordemolition))
                self))]

[meta-r-frame 'new 'statepov 'supercl 'accommodation
 'slots '(age ( $type integer $default 0
                $interval (0 +inf)
                $after-put (lambda (a)
                  (cond
                   ((< a 5) [(applymeth '? '$frame) 'r+ 'recent])
                   ((< a 15) [(applymeth '? '$frame) 'r+ 'nottooold])
                   (t [(applymeth '? '$frame) 'r+ 'old])))))]

[meta-r-frame 'new 'recent 'supercl 'statepov
 'slots '(age ($before-put (lambda (v)
                (if (>= v 5) [(applymeth '? '$frame) 'r- 'recent] ()))))]

[meta-r-frame 'new 'nottooold 'supercl 'statepov
 'slots '(age ($before-put (lambda (v)
                (if (>= v 15)[(applymeth '? '$frame) 'r- 'nottooold] ()))))]

[meta-r-frame 'new 'old 'supercl 'statepov
 'slots '(age ($after-put (lambda (a)
                (when (> a 25)
                       (print "This old accommodation should be
                             demolished or renovated! ")
                       (print "- demolish ? (y/n): ")
                       (let ((rep (read)))
                            (cond
                               ((equal rep 'y)
                                [(applymeth '? '$frame) 'r+ 'tobedemolished])
                                (t (print "-renovate ? (y/n): ")
                                   (let ((rep (read)))
                                        (cond ((equal rep 'y)
                                         [(applymeth '? '$frame) 'r+ 'renovated])
                                          (t (applymeth '$after-put a))))))))))))]

[meta-r-frame 'new 'tobedemolished 'supercl 'old
 'slots '(date ($type date))
 'methods '(after-r+ (lambda ()
                (print "La date de demolition: ")
                (applymeth '<<- 'date
                  [[date 'new (genobj 'date)] 'init])
                (when (applymeth 'repg? 'occupied)
                  (applymeth 'r+g 'tobereplaced)) )) ]

[meta-r-frame 'new 'tobereplaced 'supercl '(occupied tobedemolished)
 'slots '(contact ($before-get (lambda ()
                (print "warning: a proposal has to be made before: "
                  [%date 'pretty] " to this number: "))))
 'methods '(after-r+ (lambda () (super 'as 'occupied)))]
```

Dans cette première approche, tous les réflexes doivent être placés explicitement sur les slots concernés. Leur cohérence, ainsi que celle des réflexes sur évolution doit être assurée par le programmeur de la base.

Slots discriminants

La deuxième approche, plus déclarative, consiste à indiquer les slots discriminants, permettant de choisir une sous-classe d'appartenance selon leur valeur. L'association de réflexes appropriés est prise en charge automatiquement. La classification à l'aide de slots discriminants est implantée avec des slots particuliers, définis par la classe abstraite DiscriminantSlot. Il s'agit, une fois de plus, d'une extension du langage par la définition d'une classe de slot particulière. Cette classe est définie avec la méthode *$after-put* appropriée, qui envoie un message à toutes les sous-classes de la classe de frame qui affinent le slot discriminant en question pour spécialiser le frame qui possède ce slot. La tentative de spécialisation est à comparer avec celle mise en œuvre dans le filtrage, avec cette différence qu'aucun slot n'est obligatoirement valué (sauf le slot discriminant, qui l'est de fait, puisque l'on exécute un *$after-put*) pour que l'appariement réussisse.

On utilise un slot discriminant pour indiquer que, selon la valeur qu'un frame possède pour ce slot, on peut le classifier dans une (ou plusieurs) des sous-classes, c'est à dire dans celles qui l'accepteront en fonction des domaines restreints qu'elles définissent pour ce slot.

A titre d'exemple simple, la classe `TypePoV` (Logement du point de vue du type) avec ses deux sous-classes `House` et `Flat` est redéfinie comme suit :

```
[… TypePoV 'superclasses 'Accommodation
 'slots '(type ($supercl DiscriminantSlot $type string
                $domain ("flat" "house")) …)]

[… House 'superclasses 'TypePoV
 'slots '(type ($domain ("house")) …)]

[… Flat 'superclasses 'TypePoV
 'slots '(type ($domain ("flat"))
          level ($supercl DiscriminantSlot $type string
                 $domain ("ground" "middle" "upper")) …)]
```

Dès l'affectation du slot `type` d'un logement représentant de `TypePoV`, celui-ci évolue vers une des sous-classes (`House` ou `Flat`), en fonction de la compatibilité de la valeur affectée avec les domaines affinés. Pour assurer la continuité, on peut définir un "réflexe sur création" (*after-new*) sur les slots discriminants (dans la classe `DiscriminantSlot`). Toute évolution de frame entraînant la création d'objets pour ses nouveaux slots, la méthode *after-new* demande aussitôt à l'utilisateur une valeur pour un nouveau slot discriminant (comme `level` dans `Flat`) et peut ainsi continuer le processus. Notons que différents slots discriminants dans une classe peuvent induire des sous-classifications d'instances dans des sous-ensembles de sous-classes différents. Un slot est discriminant pour toutes les sous-classes qui l'affinent.

Il est clair que cette approche par slots discriminants convient particulièrement aux hiérarchies construites selon ce procédé, que sont par exemple les clés d'identification (souvent dichotomiques) pratiquées dans de nombreux domaines scientifiques (mycologie, botanique et aussi médecine, en enseignement de diagnostic). Les critères de choix peuvent

éventuellement être plus complexes que la valeur d'une seule propriété traduisible par un slot.

La classification est néanmoins bloquée dès qu'une classe ne contient pas de slot discriminant pour ses sous-classes.

4.4.3.2 Classification par filtrage

Nous avons vu que la clause *do* dans les filtres autorise une forme de classification explicite, le filtre lui-même spécifiant les conditions, un peu à la manière d'une règle de production (cf. l'exemple de la page 130). Les filtres peuvent de cette manière servir de "règles de classification". Quelques exemples :

```
(match '(Accommodation)
 'do '({r+ *tous les points de vue*})
 'install)

(match '(StatePoV)
 'where '(age ($verify
                (lambda(a) (< a 5))))
 'do '({r+ Recent})
 'install)

(match '(StatePoV)
 'where '(age ($interval (5 14)))
 'do '({r+ NotTooOld})
 'install)

(match '(StatePoV)
 'where '(age ($interval (15 +inf)))
 'do '({r+ Old})
 'install)

(match '(Old)
 'where '(state
           ($domain ("very bad")))
 'do '({r+ ToBeDemolished})
 'install)
```

```
(match '(Old)
 'where '(state ($domain ("bad")))
 'do '({r+ ToBeRenovated})
 'install)

(match '(Old ToBeRenovated)
 'where '(state ($domain ("ok")))
 'do '( {r- ToBeRenovated}
        {r+ Renovated})
 'install)

(match '(ToBeDemolished Occupied)
 'do '({r+ ToBeFreedForDemolition})
 'install)

(match '(ToBeReplaced)
 'do '({r+ ToBeFreedForDemolition})
 'install)
```

L'installation d'un filtre consiste à mettre à jour l'attribut *filters* des classes candidates (les super-classes directes) du filtre. Les classes possèdent une méthode *modified(o)*, qui est appelée à chaque modification de leurs représentants, c. à d. lors de l'arrivée d'un nouveau représentant de la classe (dans *after-r+*) et lors de l'affectation d'un slot (dans *$after-put* de chaque slot). La méthode *modified(o)* consiste à envoyer un message *filter(o)* à toutes les classes filtres contenues dans l'attribut *filters*. C'est une tentative de filtrage ciblé sur l'objet concerné (o).

```
after-r+ (c)    ; pour les frames
  (mapc (lambda (rc) [rc 'modified self]){? 'rclasses})

$after-put (v) ; pour leurs slots
  (mapc (lambda (rc)
          [rc 'modified {? '$frame}])
  [{? '$frame} '? 'rclasses])

modified (o)    ; pour les classes de frames
  (mapc (lambda (f) [f 'filter o]){? 'filtres})

install ()      ; pour les classes filtres
  (mapc (sc) [sc 'filters+ self]){supercld})
```

4.5 Conclusion

L'implantation orientée objet du langage FROME par extension de ROME repose entièrement sur la définition de nouvelles classes et méta-(méta-)classes. Nous nous sommes laissés guider par le principe de la dualité des "mondes" : celui des frames d'un côté et celui des slots de l'autre. Nous avons montré comment ce principe fournit un fil directeur pour la répartition des fonctionnalités des objets, selon leurs responsabilités. Ce principe peut être appliqué pour toute extension similaire de langage méta-circulaire.

L'avantage de cette méthode pour obtenir de nouveaux systèmes est l'héritage des notions du langage sous-jacent. Nous avons hérité des notions de ROME liées à l'héritage multiple par points de vue pour FROME et nous avons obtenu la représentation multiple et évolutive des frames et slots par simple spécialisation, grâce à l'héritage multiple même.

L'extensibilité de FROME a été montrée tant au niveau des nouveaux types de slots, qu'au niveau des nouveaux comportements de classes de frames, pour le filtrage. Ainsi, l'extension peut nécessiter la définition de nouveaux objets au niveau classe, méta-classe, méta-méta-classe... Le nombre de niveaux méta n'est en effet pas limité [Coi&87], grâce à la méta-circularité de ROME[1].

1. Je n'ai jamais vu encore de niveau métamétaméta, ni eu besoin de le définir.

5
Conclusion de la partie I

La représentation par objets demande une grande puissance de description. Nous avons opté pour une représentation à base de frames en raison de leur structuration en attributs et facettes. La distinction classe/instance nous semble importante pour le niveau d'abstraction qu'elle introduit. Il ne s'agit pourtant pas de la rigidité de l'approche classe/instance des langages de PPO : la Représentation Multiple et Evolutive héritée de ROME donne la souplesse supplémentaire requise pour un langage de représentation.

L'homonymie de caractéristiques découle naturellement du polymorphisme induit par la modularité de toute description d'objets. Il est donc indispensable de fournir des moyens de la gérer. La notion de point de vue de ROME s'applique à la sélection de caractéristiques homonymes ("conflits de noms") et s'étend en FROME à la sélection de spécialisations de slots ("conflits de définition").

L'implantation du langage comme une extension d'un noyau méta-circulaire lui donne toute la puissance d'un langage extensible. Nous n'avons pas intégré toutes les fonctionnalités souhaitables dans FROME, mais nous avons montré que de nouvelles fonctionnalités peuvent être obtenues sans difficulté par la définition de nouvelles classes et méta-classes. Ainsi, le nombre de facettes prédéfinies est restreint, mais peut être étendu par la définition de nouvelles classes de slots. Le fait que les classes elles-mêmes sont des objets, avec des attributs et des méthodes (ses dernières sont empruntées à la PPO), leur confère la possibilité d'avoir un comportement, qui peut être affiné pour des fonctionnalités spécifiques du système.

Nous avons implanté une forme de filtrage d'instances grâce à ces capacités du système. Un autre type de raisonnement capable d'exploiter une hiérarchie de classes est le raisonnement par classification. Nous nous proposons, dans la deuxième partie, d'étudier les conditions et les mécanismes qui régissent la classification et plus particulièrement la classification d'instances dans le cadre de la représentation par objets multi-points de vue de FROME.

Nous reviendrons plus en détail sur le statut des classes en RPO en tant que description ou définition de leurs instances. Nous étudierons également les modalités de description des objets à classifier et les éléments à comparer dans un appariement. Le traitement de l'homonymie pose de nouveaux problèmes dans le cadre de la classification et forme un point important de notre étude.

Partie II

Le Raisonnement par Classification

1
Introduction

Dans le cadre de la représentation par objets, deux types de raisonnement sont couramment exploités : le filtrage et le raisonnement par classification [Hat&91]. Nous avons étudié le filtrage dans la partie I de ce mémoire. Nous avons montré comment les moyens offerts par FROME (évolution, réflexes, méthodes) rendent possible la classification des instances de manière primitive. Une étude plus approfondie des critères de décision de l'appartenance à une classe, des moyens de description des objets à classifier et des mécanismes mis en œuvre est nécessaire.

Dans le chapitre 2 nous étudions la classification dans divers systèmes existants, selon trois axes : la description des objets à classifier, les différents types de comparaison et leurs résultats associés, et le contrôle du processus de classification lui-même.

Dans le chapitre 3 nous discutons les solutions choisies pour une classification d'instances en FROME, ce qui nous amène à combiner différentes interprétations de la notion de classe, chacune avec son mode d'appariement. Dans ce cadre, nous étudions également la prise en compte de l'homonymie des attributs, qui est une caractéristique importante du modèle FROME, induite par la représentation multi-points de vue héritée de ROME. La remise en cause de l'instanciation comme unique moyen de description des objets nous mène à une forme de description libre de l'objet, qui de ce fait n'est plus nécessairement partielle.

La conception orientée objet du système de classification pour FROME est décrite à la fin du chapitre 3. Un outil particulier est introduit pour gérer au niveau méta la description de l'objet à classifier, sous la forme d'un "méta-objet". Nous montrons que l'aspect orienté objet du langage permet d'avoir facilement accès aux différents éléments de la description, en particulier à la description des slots. Par ailleurs, le processus de classification et l'appariement qui le sous-tend sont implantés de façon répartie comme comportements des objets classes.

2
Problèmes liés à la classification d'instances

L'objectif de la classification d'objets dans un graphe de classes est de trouver sa (ou ses) classe(s) d'appartenance les plus spécifiques. Dans ce chapitre, j'examine les différentes solutions apportées par les systèmes existants aux problèmes suivants :

(1) la comparaison de l'objet à classifier aux classes d'appartenance candidates
(2) la description de l'objet à classifier
(3) le contrôle du processus de classification

L'appartenance de l'objet à une classe est jugée en fonction de l'appariement de l'objet avec cette classe. L'appariement prend en compte la définition des caractéristiques donnée par la classe (attributs et domaines de valeurs) et les caractéristiques de l'objet (valeurs d'attributs). L'appariement peut réussir, échouer, ou être indécidable. Certains systèmes proposent alors une partition des classes en classes sûres, impossibles et possibles (SHIRKA, TROPES)[1] ; d'autres donnent une mesure (numérique) de la compatibilité de chaque classe avec l'objet (SHOOD, CLASSIC), fondée sur un calcul faisant intervenir des coefficients (de cohérence, de complétude, etc.). Certains systèmes distinguent les attributs obligatoires des attributs facultatifs (SHOOD). Les classifieurs de la famille KL-ONE qui traitent les concepts individuels, par abstraction de ceux-ci et classification des concepts génériques ainsi obtenus (Classic), comparent les concepts génériques (classes) entre eux. L'appariement de deux concepts est alors décidé par le test de subsomption. Le problème qui intervient dans ces systèmes est celui des concepts primitifs, qui ne peuvent être traités par le classificateur.

En ce qui concerne la description de l'objet à classifier, on peut distinguer deux approches. La première consiste à donner une description "a priori complète" de l'objet. Le problème se pose alors de savoir comment décrire un objet complètement avant sa classification, i.e. indépendamment de son appartenance aux classes le décrivant. La seconde approche part d'une description minimale de l'objet, enrichie au fur et à mesure lors de la classification. Cette approche pose le problème de l'obtention des informations manquantes.

Quant au contrôle du processus de classification, il peut être centralisé, comme dans la plupart des systèmes, ou distribué (CSRL, YCHEM). Quelques moteurs de classification sont paramétrables (TROPES, SHOOD, CLASSIC).

2.1 L'appariement

2.1.1 Classes sûres, possibles et impossibles — La famille SHIRKA

SHIRKA

En **SHIRKA** [Rec&90], la classification se fait en spécifiant un objet existant et une classe

1. Cf. objets certains, possibles, impossibles pour l'appariement à un modèle lors du filtrage en YAFOOL (I 2.4.2.2)

(par défaut sa classe d'instanciation) qui sera le point de départ de la classification. Le système essaie de classifier l'instance dans le sous-graphe qui a pour racine cette classe, en effectuant un parcours en largeur d'abord. Le résultat est une mention attribuée à chaque classe examinée : possible, sûre, impossible.

Une classe sûre est une classe dont l'objet à classifier value tous les attributs - l'objet est dit complet pour la classe - et dont les valeurs satisfont les contraintes imposées par la classe. Ces contraintes sont exprimées à travers les facettes `$un` ou `$liste-de` qui indiquent le type de la valeur, `$intervalle`, `$domaine`, `$sauf`, `$valeur`, qui expriment une restriction du domaine, `$card-min` et `$card-max` qui déterminent le nombre de valeurs d'un attribut multivalué, et `$a-verifier` qui contient un prédicat à vérifier.
Une classe est impossible si une valeur d'attribut de l'objet à classifier viole une contrainte.
Une classe est possible si aucune contrainte n'est violée par les attributs valués, mais qu'il reste des attributs qui ne sont pas valués par l'objet[1]. Toutes les sousclasses d'une classe impossible sont impossibles. Les sousclasses d'une classe possible sont possibles ou impossibles.

L'instance classifiée n'est jamais rattachée directement aux classes résultantes du processus de classification. Elle peut être attachée à une (seule) des classes qualifiées comme sûres ou possibles par une commande explicite. Sans ce rattachement explicite, l'instance reste attachée à sa classe initiale, mais elle peut être visualisée avec une option spéciale "vu-comme" une classe. Cela permet de prendre l'hypothèse de l'appartenance de l'instance à la classe et de voir les conséquences de son rattachement en termes d'attributs, valeurs d'attributs inférées etc. sans que l'instance ne les possède réellement. Cette option "vu-comme" ne peut être utilisée que sur les classes reconnues sûres ou possibles au préalable.

TROPES

Le système TROPES ([Mar89], [Mar&90], [Mar91]) est un système dédié à la classification multi-points de vue. Les points de vue sont clairement identifiés et forment la base de structuration des classes. Interdisant l'héritage multiple entre concepts, TROPES organise chaque concept en une structure de classes indépendante (une "famille"), cette structure étant elle-même organisée en hiérarchies représentant des points de vue différents. Chaque point de vue regroupe un sous-ensemble des attributs du concept, considérés comme pertinents sous ce point de vue. Ces sous-ensembles ne sont pas nécessairement disjoints : un même attribut peut être pertinent dans plusieurs points de vue. L'ensemble des points de vue donne une vision complète du concept. De plus, la racine d'un point de vue d'un concept décrit l'ensemble complet des éléments du concept. Chaque instance du concept est donc au moins instance de la racine de chaque point de vue.

Chaque point de vue partitionne le concept suivant un attribut discriminant de façon à former un arbre de classes disjointes. Il y a donc représentation multiple au niveau du concept et représentation simple au niveau de chaque point de vue.

Comme en SHIRKA, la représentation d'une classe est postulée complète[2], c'est à dire que la satisfaction des contraintes des attributs de la classe est une condition nécessaire et

1. Une option de la commande de classification permet de ne prendre en compte qu'un sous-ensemble des attributs, identifiés par leur type (classe d'attribut). Ces attributs doivent alors être distingués comme tels dans la classe. Voir aussi page 160 §2.2

suffisante pour appartenir à la classe.[1]

L'originalité de TROPES est que l'on peut relier certaines classes, situées sous des points de vue différents, par des passerelles inter-points de vue. Ces passerelles expriment des "liens logiques" [Mar89] entre ces classes, du style :

$$C/PV_i \Rightarrow C'/PV_j$$

ou : "toutes les instances de C appartiennent à C′ "

(où C et C′ appartiennent à des points de vue différents du même concept)

Exemple [Mar91] :

"Tout véhicule de transport de marchandises est économique".

Cette implication est traduite par une passerelle entre ces deux classes d'automobiles appartenant respectivement aux points de vue "utilisation" et "physique" (cf. Fig. 61).

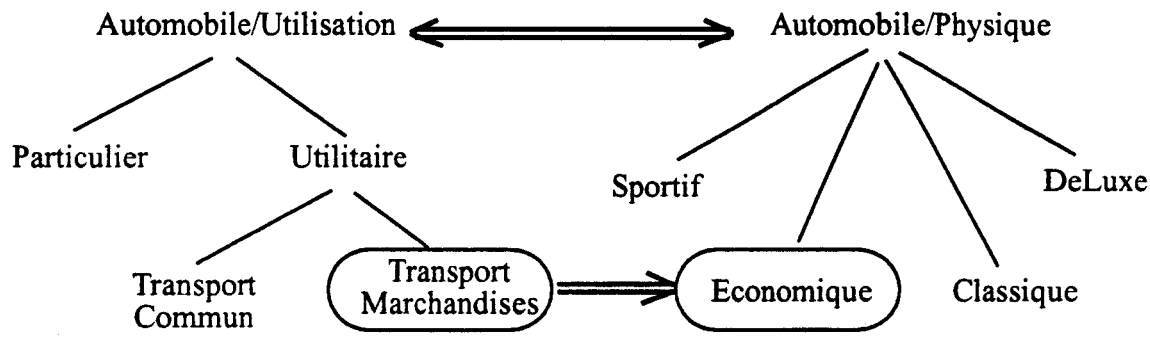


Fig. 61 TROPES : passerelles entre points de vue

S'il s'agit d'un lien dans les deux sens, comme entre les racines de tous les points de vue d'un concept, nous avons affaire à la même classe (même extension) représentée sous des points de vue différents (intensions différentes, i.e ensembles d'attributs pertinents différents).

Cette notion de passerelle (uni-ou bi-directionnelle) joue un rôle primordial dans le processus de classification. En effet, les labels "possible", "impossible", et "sûr" attribués aux classes lors de la classification d'une instance (cf. SHIRKA), sont propagés via ces passerelles d'un point de vue à un autre, ce qui permet de prendre des raccourcis par rapport au parcours classique. Puisque la représentation d'une classe est considérée comme complète, la satisfaction des contraintes dans une représentation de la classe (c. à d. la description de cette classe sous un point de vue) implique l'appartenance à cette classe, indépendamment de tout point de vue. Partant de cette hypothèse, la propagation des labels est parfaitement justifiée.

De plus, les règles connues de SHIRKA concernant la propagation des labels sur les sousclasses, permettent au système d'inférer plusieurs labels sur les classes du point de vue d'arrivée d'une passerelle. En particulier, toutes les surclasses d'une classe sûre deviennent

2. C. à d. (dans la terminologie de la communauté KL-ONE) qu'une classe est considérée comme un concept défini et non pas comme un concept primitif.

1. Récemment, [Euz93] a proposé une autre interprétation, où les classes sont *descriptives* (i.e. primitives).

sûres.

Quand on simule l'appartenance multi-points de vue des classes passerelles par des liens de sous-classement (lien classique pour exprimer l'inclusion des ensembles représentés), on obtient de l'héritage multiple entre points de vue (Fig. 62).

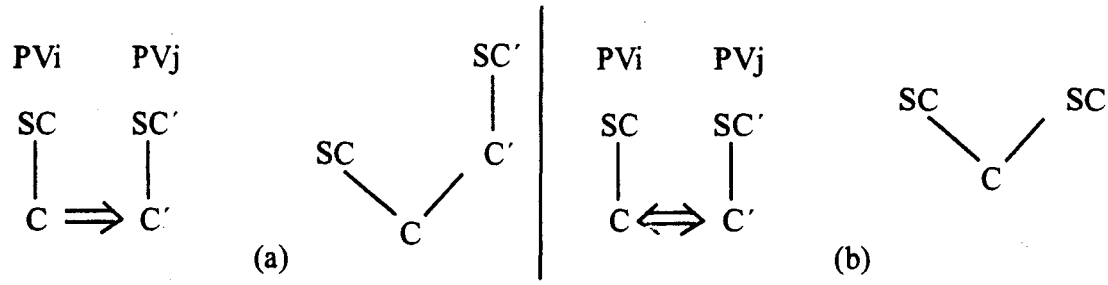


Fig. 62 Simulation de passerelle unidirectionnelle (a) et bi-directionnelle (b) par héritage multiple.

Classiquement, pour qu'une instance puisse appartenir à une classe, une condition nécessaire est qu'elle appartienne à toutes les superclasses de cette classe. Selon cette approche, il n'est pas possible de déduire d'abord l'appartenance à la classe C sans prendre en compte une des superclasses, C′ et d'en déduire ensuite l'appartenance à cette superclasse.

L'exemple suivant tiré de [Nap92] illustre la même idée qui doit justifier les passerelles de TROPES : "si C est un concept décrit par la disjonction $C_1 \cup C_2 \cup ... \cup C_n$ alors un objet O est un représentant de C s'il satisfait une des descriptions C_i; réciproquement, si O satisfait une description C_i, il les satisfait toutes".

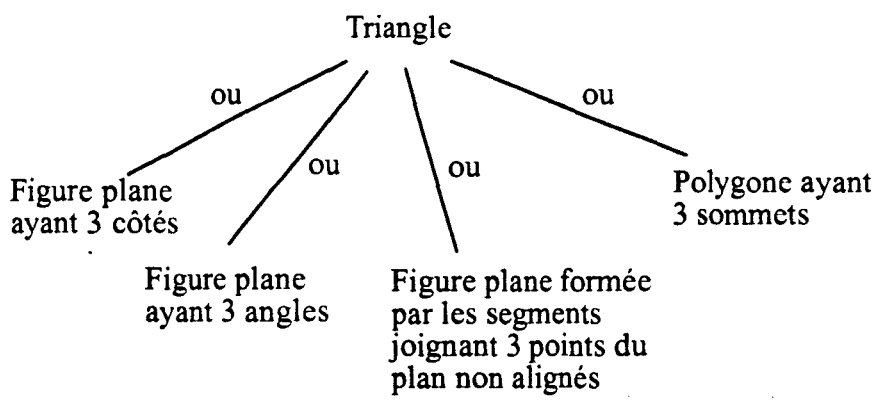


Fig. 63 : Descriptions d'un triangle selon différents points de vue

Le concept de Triangle est représenté par plusieurs descriptions qui expriment des points de vue différents mais équivalents (Fig. 63).

Cela marche très bien quand il s'agit de concepts parfaitement définis, comme les concepts mathématiques, mais dès que l'on parle de concepts moins bien définis comme les automobiles de transport de marchandises, les équivalences ou implications de descriptions sont beaucoup plus difficiles à trouver. C'est là tout le problème de l'identification des passerelles, qui incombe nécessairement à l'utilisateur.

2.1.2 La subsomption

Dans les langages de description de concepts, un classificateur place chaque nouvelle description dans une hiérarchie ou taxonomie, selon un algorithme de classification qui fait intervenir des tests d'appariement fondé sur la *subsomption* entre ces descriptions. La définition de la subsomption a été donnée de différentes manières équivalentes dans la littérature [Nap92]. [Sch&83] donne une définition fondée sur l'interprétation ensembliste des concepts, qui est celle de l'inclusion des ensembles : le concept A subsume le concept B ssi l'ensemble dénoté par A contient nécessairement l'ensemble dénoté par B.

Comme l'inclusion des ensembles, la subsomption est une relation transitive, réflexive et anti-symétrique et induit un ordre partiel sur les concepts, représenté par un graphe, qui matérialise également les liens d'héritage entre concepts : un concept hérite les propriétés de tous ceux qui le subsument.

2.1.2.1 Subsomption de termes

Chaque concept est décrit à l'aide de concepts déjà présents dans la taxonomie. Un concept peut être vu comme un terme logique complexe, construit à partir de termes élémentaires. La taxonomie des concepts est également appelée *terminologie*. Elle constitue le vocabulaire de description des concepts : la TBOX. Nebel montre dans [Neb90] que la subsomption dans une terminologie de concepts peut être réduite à la subsomption de termes. Chaque concept peut en effet être remplacé par sa définition (et récursivement). Il est alors constitué d'un ensemble de termes atomiques. Un terme réduit de cette façon est appelé *p-terme*. La subsomption de termes fait complètement abstraction de la taxonomie, ce qui peut être utile pour l'analyse formelle des propriétés de la subsomption, mais n'a pas beaucoup d'intérêt pour la représentation des connaissances.

2.1.2.2 Algorithme de subsomption de concepts

[Sch&83] décrit l'algorithme qui implante le test de subsomption, tel qu'il a été défini pour le classificateur de KL-ONE. Le test compare les concepts élément par élément :

SubsumesP(A,B) est vrai ssi

- tous les concepts primitifs[1] qui subsument A subsument B
- pour chaque rôle de A, il existe un rôle de B qui dénote la même relation, et pour chacune de ces paires de rôles :
  - la cardinalité associée au rôle de A inclut celle associée au rôle de B
  - la description de valeur du rôle de A subsume celle du rôle de B

Fig. 64 Algorithme de subsomption

1. voir 2.1.2.3

2.1.2.3 Concepts primitifs, concepts définis

La description des concepts génériques ("classes") dans les systèmes à subsomption est fondée sur la distinction entre concepts primitifs et concepts définis. Les concepts primitifs sont ceux qui représentent une description incomplète ou un concept indéfinissable ; la description renferme des **conditions nécessaires** (CN), mais non suffisantes d'appartenance au concept. Par contre, la description des concepts définis précise toutes les **conditions nécessaires et suffisantes** (CNS) d'appartenance. Les concepts sont ordonnés entre eux par le classificateur. Le critère de placement d'un concept par rapport à un autre est le test de subsomption. Toutefois, ce test ne peut comparer des concepts primitifs entre eux. Un concept défini ne peut être classé sous un concept primitif, à moins de renfermer explicitement celui-ci dans sa définition [Doy&91]. De même, la relation de subsomption entre un concept primitif et un concept individuel (l'"appartenance" du concept individuel au concept générique primitif) doit être déclarée par l'utilisateur et ne peut pas être déduite par le classificateur.

Pour faire fonctionner le classificateur de manière optimale, le concepteur de la base doit donc prendre soin de minimiser le nombre de concepts primitifs. Ceci peut être un sérieux handicap, car dans certains domaines d'application (peut-être même la plupart) les concepts primitifs tendent à être plus nombreux que les concepts définis [Fin86].

Fikes et Kehler remarquent dans [Fik&85] que les langages de frames en général considèrent que la classe (le frame) ne peut être plus qu'une description (concept primitif) : elle renferme des conditions nécessaires, mais non suffisantes d'appartenance, et on peut donc tout au plus inférer qu'une instance ne peut y appartenir. Les conditions suffisantes manquantes peuvent être spécifiées pour chaque classe par des règles de production locales qui spécifient les conditions pour que les représentants des superclasses directes y appartiennent, permettant un algorithme de classification par affinements successifs.

Si, en revanche, la description d'une classe ne peut même pas être considérée comme un ensemble de conditions nécessaires, alors aucune inférence n'est possible et on peut tout décrire, autrement dit : on ne peut plus rien décrire vraiment. C'est le cas des systèmes à base de prototypes, ou chaque description doit être interprétée comme une **description par défaut**, car sujette à exceptions. Brachman a dénoncé cette approche dans [Bra85], où il plaide pour une composante définitionelle dans les systèmes de représentation de connaissances. On doit au moins pouvoir définir des concepts par *composition* et utiliser cette facilité pour pouvoir inférer un minimum de choses. Exemple : si on définit le concept Eléphant-à-trois-pattes, on voudrait bien pouvoir inférer d'une part, qu'un individu instance d'Eléphant et ayant trois pattes appartienne à ce concept, et d'autre part, qu'une instance du concept Eléphant-à-trois-pattes possède bien trois pattes. Or, paradoxalement, les langages qui défendent l'exception ("cancellation"[1]) pour pouvoir représenter les éléphants roses et les éléphants à trois pattes, ne peuvent même pas reconnaître les individus qui correspondent à ces descriptions.

2.1.2.4 Composantes primitives

Afin d'uniformiser la sémantique des concepts, Nebel a introduit la notion de "composante primitive" d'un concept [Neb90]. La composante primitive d'un concept primitif est précisément la partie de la définition qui manque pour en faire un concept défini. En utilisant

1. par exemple [Fah&81].

les composantes primitives, chaque concept peut être entièrement défini. Notons que les concepts définis subsumés par un concept primitif comportent également la composante primitive de ce dernier, par héritage. La composante primitive pour le concept Personne est notée $\overline{\text{Personne}}$. Personne est maintenant défini :

```
Personne = (and Anything Personne)
```

De la même façon on peut définir complètement les concepts Homme et Femme :

```
Homme = (and Personne Homme)

Femme = (and Personne Femme)
```

Dans BACK [Hop&93], les composantes primitives sont générées et manipulées par le système pour décrire complètement les définitions de concepts en fonction de leurs termes atomiques.

2.1.2.5 Règles

Pour compléter les inférences opérées par le classificateur et contourner les difficultés soulevées par la présence de concepts primitifs, Classic [Bra&91] utilise des règles de chaînage avant (cf. aussi page 62). Ces règles sont assez particulières, car leur prémisse est un concept générique existant dans la base, et leur conclusion est une expression de concept générique quelconque. La sémantique de ces règles est la suivante : tout concept individuel appartenant au concept de la prémisse (i.e. subsumé par celui-ci) est affirmé[1] appartenir au concept de la conclusion. D'une part cette pratique fournit un moyen de déduire des informations de l'appartenance de l'individu à un concept particulier, sans en faire des conditions d'appartenance à ce concept. D'autre part, elle permet de déduire l'appartenance à des concepts primitifs sous certaines conditions.

En effet, [Bra&91] distingue les propriétés que possèdent tous les représentants d'un concept (propriétés accidentelles[2]) des propriétés qui permettent de décider si un individu appartient au concept (propriétés essentielles). Ces dernières sont des conditions nécessaires et forment la définition du concept, alors que les premières sont des conséquences de l'appartenance au concept, qui ne doivent pas être placées dans sa définition.

Exemple (tiré de [Bra&91]) : si on sait que tous les vins rouges de Bordeaux sont secs (sic), on peut exprimer cela sous la forme d'une règle Classic :

```
assert-rule[RED-BORDEAUX-WINE, DRY-WINE]
```

Cette règle est applicable à tout concept individuel subsumé par le concept `RED-BORDEAUX-WINE`, et a pour conséquence l'assertion de son appartenance au concept `DRY-WINE`. La propriété d'être sec ne fait cependant pas partie de la définition des vins rouges de Bordeaux, qui sont définis par le concept :

```
define-concept[RED-BORDEAUX-WINE,
               (AND WINE
                (FILLS color Red)
                (FILLS region Bordeaux))]
```

1. "asserté"
2. *incidental* properties.

Ce concept est classé sous celui de WINE, et non pas sous celui de DRY-WINE, même si tous les vins rouges de Bordeaux sont des vins secs, comme il est exprimé dans la règle. Les règles interviennent seulement dans la classification des concepts individuels, pas dans celle des concepts génériques.

La distinction entre propriétés essentielles et accidentelles semble pourtant difficile à faire. Pour identifier un vin comme vin rouge de Bordeaux, on est peut-être plus aidé par la connaissance de la règle (car on peut goûter le vin) que par la définition donnée par le concept, qui impose de connaître la région. En particulier, si le vin n'est pas sec, l'appariement peut être considéré comme échoué. Ces informations ne sont pas prises en compte par le classificateur de Classic.

L'utilisation des règles pour la classification des concepts individuels par rapport aux concepts primitifs est illustrée par l'exemple suivant :

Le concept de PERSONNE est un concept primitif, car on ne peut pas énumérer toutes les propriétés nécessaires et suffisantes pour décider si un individu appartient à ce concept. Par contre, on peut donner des sous-ensembles de propriétés considérées comme suffisantes pour déterminer l'appartenance, par exemple BIPEDE-SANS-PLUMES[1] ou ENFANT-D'UNE-PERSONNE, qui seraient des concepts définis. Il suffit ensuite de définir des règles avec comme conditions chacun de ces concepts définis et en conclusion le concept PERSONNE, pour pouvoir classer les individus correspondants à ces définitions. Cette approche rappelle celle des passerelles de TROPES qui relient des définitions équivalentes d'un même concept. [Woo91] introduit le terme d'*abstraction conceptuelle* pour l'opération de description d'un concept par une disjonction de définitions équivalentes. Quand une des définitions est satisfaite, toutes les autres le sont. Cette équivalence correspond aux passerelles bi-directionnelles de TROPES, tandis que les règles de CLASSIC traduisent seulement des implications (passerelles uni-directionnelles).

L'idée est attrayante, car elle donne un moyen de définir un concept de façons multiples, là où une définition générale fait défaut. Le problème de décider de l'équivalence des définitions reste ouvert. De plus, la nuance entre l'implication de concepts (exprimée dans la règle) et la relation de subsomption entre ces concepts est difficile à saisir. Elle semble être surtout d'ordre opérationnel : "on n'a pas besoin de s'assurer que l'instance appartienne au concept en conclusion pour décider de son appartenance au concept de la condition".

Les systèmes LOOM et MESON permettent également la spécification de relations d'implication entre concepts [McG91][2]. Selon Mac Gregor, ces règles sont utilisées pour spécifier des conditions nécessaires supplémentaires qui augmentent la définition d'un concept, sous la forme de nouvelles contraintes sur ce concept, ce qui semble être en contradiction avec l'utilisation qui est faite des règles en Classic (déduction en chaînage avant : expression d'une *conséquence* et non de conditions nécessaires d'appartenance au concept). On peut dire néanmoins qu'il s'agit de conditions nécessaires dans le sens où elles

1. Cette définition (Platon, "Le politique," in *Sophiste, Politique, Philèbe, Timée, Critias*, Trad. E. Chambry, Garnier Flammarion, 1969, Paris) était déjà quelque peu controversée dans l'antiquité : "Platon ayant défini l'homme un animal à deux pieds sans plumes, et l'auditoire l'ayant approuvé, Diogène apporta dans son école un coq plumé, et dit : «Voilà l'homme selon Platon.» Aussi Platon ajouta-t-il à sa définition : « et qui a des ongles plats et larges»." (Diogène Laërce, "Vie, doctrines et sentences des philosophes illustres II", Trad. R. Genaille, Garnier Flammarion, 1965, Paris)
2. ainsi que BACK [Hop&93].

sont nécessairement vérifiées pour tout individu appartenant au concept.

2.1.3 Mesures de compatibilité

2.1.3.1 SHOOD

[Rie&91a/b] remet en question l'instanciation comme moulage parfait des instances par leur classe. Une instanciation par "moulage souple" doit autoriser un objet à passer par des états incohérents. On associe à un objet son degré de complétude et de cohérence par rapport à chaque classe d'appartenance. Le degré de complétude d'un objet dépend du nombre d'attributs valués. SHOOD distingue les attributs obligatoires des attributs facultatifs. Les attributs obligatoires doivent être valués dès l'instanciation de l'objet dans la classe. Le pourcentage d'incomplétude d'une instance i dans une classe C est alors défini comme le pourcentage d'attributs facultatifs qu'il reste à instancier, donné par la formule :

$$N(C,i) = \frac{Nc(C) - Ns(C,i)}{Nc(C)}$$

avec Nc(C) le nombre d'attributs facultatifs de C, Ns(C,i) le nombre d'attributs facultatifs de C valués par i.

La cohérence d'une instance dans une classe dépend de la vérification des contraintes définies par la classe. SHOOD distingue contraintes fortes et contraintes faibles. Les contraintes fortes ne peuvent pas être violées, contrairement aux contraintes faibles. La violation de contraintes faibles diminue la cohérence de l'instance. Des exemples de contraintes fortes sont les contraintes de type et la valuation des attributs obligatoires. Une contrainte faible peut être l'intervalle des valeurs admises pour un attribut.

Une contrainte est appelée "indécidable" si l'absence d'un (ou de plusieurs) paramètres ne permet pas de la vérifier. Les paramètres étant le plus souvent des attributs, l'indécidabilité d'une contrainte correspond alors à une incomplétude de l'objet, qui ne value pas ces attributs-paramètres. Le pourcentage d'incohérence d'une instance i dans une classe C est alors défini par la formule :

$$L(C,i) = \frac{Lc(C) - Ls(C,i)}{Lc(C)}$$

avec Lc(C) le nombre de contraintes faibles, Ls(C,i) le nombre de contraintes faibles indécidables ou satisfaites par i. Le pourcentage d'incohérence dépend donc du nombre de contraintes faibles violées par l'instance, les contraintes fortes devant obligatoirement être vérifiées. En effet, une classe est impossible pour une instance qui viole une (ou plusieurs) de ses contraintes fortes. Toutes les autres classes sont alors possibles pour l'instance, et elle peut y être rattachée tant qu'aucune contrainte forte n'est violée.

La notion de classe "sûre", pour désigner une classe pour laquelle l'instance est complète et cohérente (cf. SHIRKA) est supprimée, pour faire place à une gradation de classes "plus ou moins" possibles en fonction des pourcentages d'incohérence et d'incomplétude de l'instance

dans celles-ci. Par conséquent, toutes les instances d'une classe ne représentent pas nécessairement cette classe au même degré. Une instance peut être plus complète ou plus cohérente qu'une autre pour la même classe d'appartenance. Les plus complètes et plus cohérentes sont celles qui valuent tous les attributs de la classe et ne violent aucune contrainte (faible). Pour distinguer les degrés de correspondance des différentes instances d'une classe, on construit des structures dites significatives à l'intérieur d'une classe, qui reflètent toutes les combinaisons possibles d'attributs facultatifs valués et de contraintes faibles violées ou satisfaites. Chaque instance est alors rattachée dans sa classe à sa structure significative maximale, c. à d. celle dont elle value tous les attributs et vérifie toutes les contraintes. Toutes les instances d'une classe valuant les mêmes attributs et respectant les mêmes contraintes sont donc rattachées à la même structure. A chaque structure sont associés un pourcentage d'incohérence et un pourcentage d'incomplétude communs à toutes ses instances.[1]

Pour une instance i à classifier, le mécanisme de classification de SHOOD délivre pour chaque classe possible un triplet, appelé critère de classification, composé de la structure significative dans la classe, du pourcentage d'incomplétude, et du pourcentage d'incohérence de l'instance dans la classe. L'utilisateur peut ensuite rattacher l'instance à la classe ou aux classes de son choix.

Il peut aussi fournir une "condition de classification" en paramètre à la commande de classification, qui indique les seuils de retenue des classes en termes de pourcentages de cohérence et de complétude. L'ensemble des classes retenues est donc un sous-ensemble des classes possibles. Comme en SHIRKA, le rattachement effectif d'une instance à une classe possible ("possible" ou "sûre" en SHIRKA, "retenue" en SHOOD), doit être fait explicitement.

N'autorisant que l'instanciation simple, SHIRKA peut se contenter de donner la liste des classes en vrac, même si certaines classes possibles sont en réalité des classes disjointes du fait de l'exclusion des domaines de valeurs d'attributs par exemple. En effet, le rattachement effectif se fait toujours à une seule de ces classes. En TROPES, l'instance ne peut appartenir qu'à une seule classe sous chaque point de vue. SHOOD par contre, permet l'instanciation multiple[2] et ne peut pas se contenter de donner la liste des classes retenues en vrac. Quand il y a des classes disjointes[3] parmi les classes retenues, le résultat de la classification délivre plusieurs graphes de classes retenues, qui correspondent à des alternatives de rattachement. L'objet peut être rattaché à une ou plusieurs classes d'un seul graphe retenu.

La commande de classification se présente comme suit :

```
MIC (i|new) [(dans C1)| (depuis C2 [sauf LCE])]
    [si cond] [valsattributs]
```

La classification se fait :

1. Les structures significatives sont ordonnées par une relation d'ordre. Leur statut n'est pas clair. Elles font penser à une sorte de hiérarchie de classes "dans la classe".
2. On ne sait pas s'il s'agit seulement de l'instanciation multiple des instances terminales, comparable à la représentation multiple de ROME, ou aussi de celle des classes. Quant au mécanisme de classification, [Rie&91a/b] affirme qu'il s'applique aussi bien sur des instances terminales que sur des classes. (Les répercussions de la classification des classes sur leurs instances n'y sont pas traitées.)
3. Un lien de disjonction explicite (comme en FROME) indique les classes disjointes. On ne sait pas si d'autres formes de disjonction (implicite : comme l'exclusion mutuelle de domaines de valeurs) sont prises en compte.

`i|new` : sur une instance existante (i) ou sur une instance qui n'existe pas encore, et qui sera éventuellement créée <u>après</u> cette classification, par la commande INST explicite. En attendant, seul un identificateur est généré.

`(dans C1)` : dans une seule classe (C1)

`(depuis C2 [sauf LCE])` : dans le sous-graphe de racine C2, à l'exclusion des sous-graphes de racines incluses dans LCE

`()` : dans tout le graphe.

`[si cond]` : la condition pour qu'une classe soit retenue, portant sur les critères de classification.

`[valsattributs]` : donne les valuations de tout ou partie des attributs.

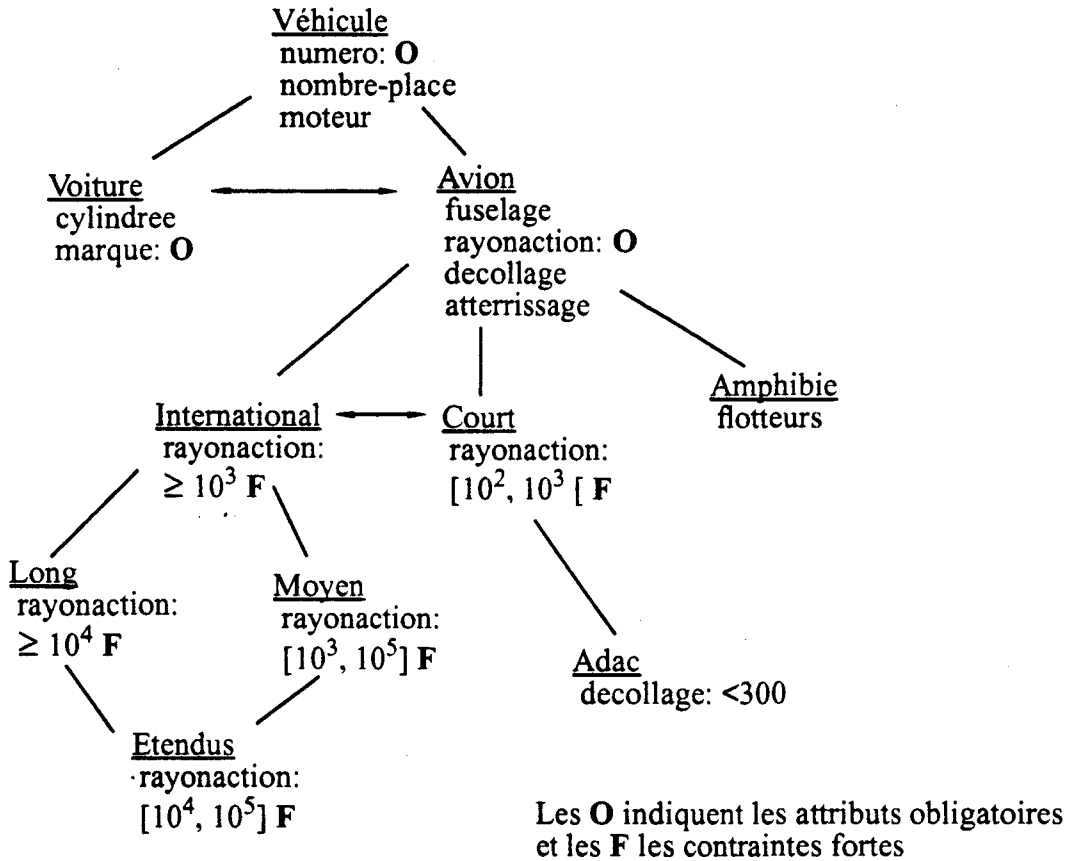


Fig. 65 SHOOD : Graphe de classification d'avions

Exemple (cf. Fig. 65)

```
> MIC new depuis Vehicule [numero = F-GFXT, rayonaction = 120]
```

retourne l'arbre suivant des classes possibles :

```
(Objet Vehicule Avion ((Court Adac) Amphibie))
```

La classe Voiture n'est pas possible car l'attribut obligatoire "marque" n'est pas valué. La classe Court est possible avec un pourcentage d'incomplétude de 100% (aucun des attributs facultatifs n'est valué) et un pourcentage d'incohérence de 0% (aucune contrainte faible n'est

violée).

Exemple avec condition de classification : (supposons que i appartienne à la classe Avion)

```
> MIC i dans Adac si N < N.Court et L ≤ L.Court [decollage = 400]
```

La condition exprime qu'il faut retenir la classe Adac si l'instance est plus complète que pour Court et au moins aussi cohérente. Ceci n'est pas le cas : le critère de classification de i dans Adac est :

SxAdac = {numero, rayonaction, decollage, NOT decollage < 300}
N = 4/5 = 80% (un seul attribut facultatif sur cinq est valué)
L = 100% (la seule contrainte faible est violée).

Adac n'est donc pas retenue car l'instance est plus cohérente dans la classe Court (bien que moins complète).

La distinction entre, d'une part, une partie obligatoire constituée par les attributs obligatoires et les contraintes fortes et, d'autre part, une partie facultative constituée par les attributs facultatifs et les contraintes faibles forme la base d'une qualification de l'appartenance d'une instance à ses classes. La partie obligatoire garantit la cohérence et la complétude minimum des instances de la classe, et la partie facultative permet d'établir une gradation d'instances correspondant plus ou moins bien au moule.

2.1.3.2 CLASSIC

CLASSIC est un générateur de systèmes experts de classification, qui a été commercialisé par ILOG [Gra88][1]. Le système fonctionne avec un arbre de classes, appelées prototypes, un objet à classer, une base de règles et un "moteur".

Les classes définissent les champs, de type "intervalle" pour les valeurs numériques, "ensemble" pour les valeurs symboliques, ou "prototype" pour les parties.

La Fig. 66 montre un exemple de hiérarchie de prototypes, avec définition de champs et de leurs domaines.

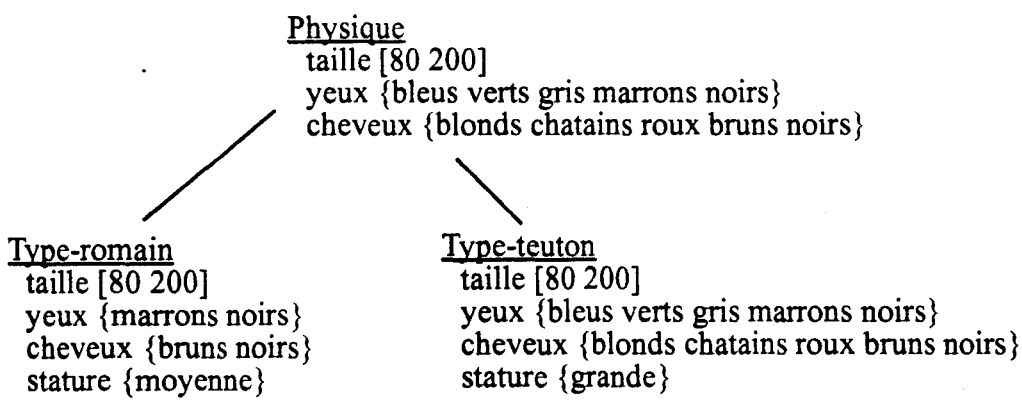


Fig. 66 CLASSIC : hiérarchie de types de figurants

Pour déclencher une classification, l'utilisateur décrit l'objet à classer à l'aide d'un éditeur

1. A ne pas confondre avec le système de même nom descendant de KL-ONE [Bor&89].

d'objets. Dans cet éditeur, il peut faire apparaître soit la liste des champs demandables de toute l'arborescence des classes, soit la liste des champs déductibles, soit la totalité des champs. Il peut alors valuer certains de ces champs et laisser les autres à "inconnu".

Lors de la classification, les classes sont examinées suivant un ordre de priorité. Par défaut c'est le parcours suivant l'ordre des sous-classes de la classe courante (en partant de la racine), qui correspond à un parcours en profondeur d'abord. Des seuils de compatibilité et d'incompatibilité sont associés à chaque classe. On peut spécifier si la comparaison doit prendre en compte les champs hérités ou seulement les champs locaux. Enfin, on peut spécifier des règles "si échec" et "si succès". Ces paramètres sont modifiables localement (pour une classe) ou globalement.

Une étape de classification standard se déroule comme suit :

- classification des parties
- comparaison objet-classe
- activation des règles "si échec" ou "si succès" et en cas de succès, activation des règles d'inférence attachées à la classe.
- établissement de la liste des classes à examiner
- traitement de la liste établie dans l'ordre

La comparaison entre l'objet et la classe se base sur des calculs de coefficients de compatibilité et d'incompatibilité pour les champs de la classe (calculés suivant le nombre de valeurs d'attributs de l'objet respectant les domaines définis dans la classe). Les coefficients de compatibilité et d'incompatibilité des champs sont combinés pour calculer ceux de la classe, en faisant intervenir le coefficient d'importance de chaque champ. Ensuite, la classe est acceptée ou rejetée en comparant les coefficients et les seuils de compatibilité et d'incompatibilité associés à la classe. Le coefficient de compatibilité mesure le degré de ressemblance entre l'objet et la classe (l'objet présente-t-il toutes les caractéristiques de la classe ?) tandis que le coefficient d'incompatibilité mesure leur degré de différenciation (l'objet présente-t-il au moins un trait non caractéristique de la classe ?). En cas d'acceptation, les règles d'inférence sont déclenchées.

Comme SHOOD, CLASSIC donne une mesure de la compatibilité de l'objet avec les différentes classes. Pourtant, il parait difficile d'attribuer un sens à tous les calculs numériques faisant intervenir toutes sortes de coefficients.

2.1.3.3 La famille MDX

Dans le cadre du projet MDX autour du diagnostic médical, l'équipe de Chandrasekaran à l'Université d'Etat d'Ohio a identifié la classification hiérarchique comme une tâche générique intervenant dans des tâches plus complexes comme le diagnostic et la planification. [Cha83][Cha85][Dek92]. Selon Chandrasekaran, un processus de raisonnement complexe peut être décomposé en un certain nombre de tâches. Chaque tâche requiert son propre mode de raisonnement et son propre type de connaissance. Les mêmes tâches peuvent surgir dans des processus différents de résolution de problèmes. C'est en ce sens qu'elles sont qualifiées de "génériques".

Ainsi, le diagnostic englobe des tâches comme la classification hiérarchique, l'inférence de données, la vérification d'hypothèses (*matching*) et la construction d'hypothèses explicatives complexes par raisonnement abductif. Si les premiers systèmes de diagnostics développés

dans le projet intègrent encore plusieurs tâches en un seul système (MDX, CSRL), par la suite, l'équipe s'efforcera de développer des outils propres à chaque tâche et de combiner ces "building blocks" pour réaliser des tâches plus complexes (*design*, planification...).

MDX

Dans MDX, système dédié au diagnostic des maladies du foie, la classification hiérarchique et l'appariement d'hypothèses sont confondus en une seule "tâche diagnostique". Deux systèmes auxiliaires jouent le rôle d'assistants : ils organisent et fournissent les données des patients (PATREC) et les analyses radiologiques (RADEX), nécessaires au raisonnement diagnostique.

Le diagnostic consiste à identifier une description de cas (de maladie) dans une hiérarchie de diagnostics préétablie [Cha83]. Chaque nœud (concept) dans la hiérarchie représente une hypothèse de diagnostic, les plus générales étant placées plus haut que les plus spécifiques. A chaque concept sont associées des règles qui permettent de déduire si l'hypothèse représentée doit être rejetée ou retenue. Chaque nœud est alors un "spécialiste" de l'hypothèse qu'il représente.

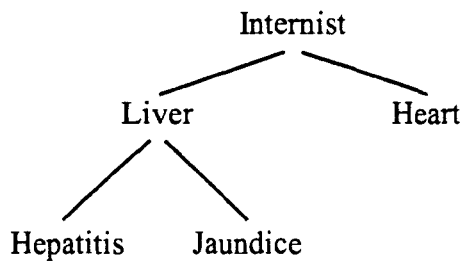


Fig. 67 Hiérarchie de spécialistes : hypothèses de maladies

Le processus de classification consiste à confronter la description d'un cas avec la hiérarchie des concepts en parcourant cette dernière à partir du concept le plus général vers les plus spécifiques. Chaque concept visité applique les règles locales et se rejette ou s'établit. En cas de rejet, les concepts fils ne seront pas examinés. En cas d'acceptation, le processus se propage sur les fils.

Les règles de production s'appliquent sur les données du patient et produisent des coefficients de vraisemblance qui sont combinés pour déterminer le coefficient de vraisemblance de l'hypothèse représentée par le concept auquel elles sont attachées. C'est en fonction de ce coefficient que l'hypothèse est acceptée ou rejetée.

Une application de la méthode MDX est la réimplantation de la partie diagnostic de Mycin : MDX-Mycin [Sti&85].

CSRL

Le successeur de MDX, CSRL [Byl&85, 86], est l'outil dédié à la classification, qui a servi à réaliser plusieurs systèmes de diagnostic, dont :

- Auto-Mech : un système de diagnostic de problèmes de carburant de voitures

- RED : identification d'anticorps dans le sang
- WELDEX : détection de soudures défectueuses

Les premières versions de CSRL font aussi bien de la classification que de l'appariement d'hypothèses (comme MDX). Cette dernière fonctionnalité est ensuite identifiée comme une tâche à part entière [Byl&86] et aura son propre outil spécialisé (HYPER [Byl&89]).

Le langage CSRL (= Conceptual Structures Representation Language), écrit en LOOPS et INTERLISP-D [Byl&85], permet de :

- créer une hiérarchie de spécialistes
- coder la connaissance d'appariement (*pattern matching*) pour chaque hypothèse dans un spécialiste
- contrôler le processus "establish-refine" qui consiste à essayer d'établir une hypothèse et si elle est établie, affiner celle-ci en appliquant récursivement le processus sur les sous-spécialistes (hypothèses plus spécifiques).

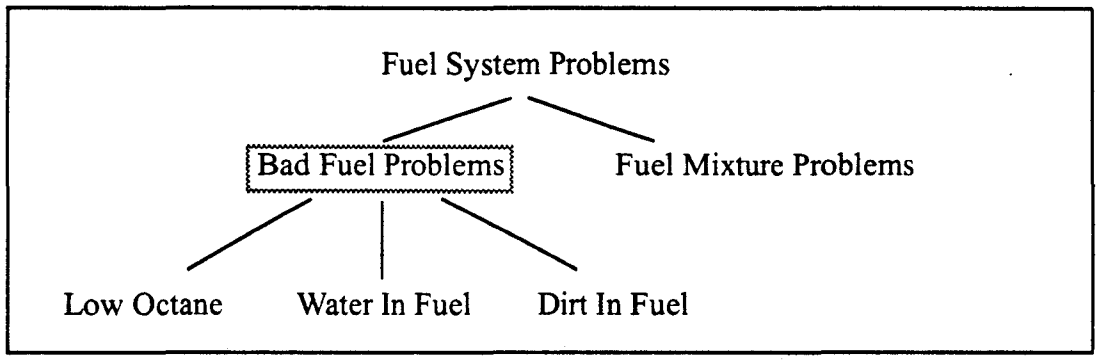


Fig. 68 CSRL : Hiérarchie de spécialistes

Chaque spécialiste est défini comme une classe LOOPS qui est instanciée pour chaque cas traité. Le spécialiste chargé de l'hypothèse "Bad Fuel" de la hiérarchie ci-dessus est défini comme suit:

```
(Specialist BadFuel
  (declare (superspecialist FuelSystem)
           (subspecialists LowOctane WaterInFuel DirtInFuel)
  (kgs ...) ;knowledge groups
  (messages ...)) ;establish, refine ou autres
```

Les "knowledge groups" regroupent la connaissance décisionnelle concernant l'hypothèse sous forme de règles. Les règles permettent d'obtenir un coefficient de vraisemblance pour le groupe auquel elles appartiennent. La combinaison des résultats des différents groupes détermine le coefficient de l'hypothèse. Les messages "establish", "refine", correspondent à des méthodes pour établir le coefficient de vraisemblance de l'hypothèse (comment combiner les résultats des groupes de règles) et pour propager le processus sur les fils.

Les règles sont représentées sous forme d'une expression d'appariement ("match <liste d'expressions> with <liste d'expressions>") qui traduit une table de vérité. Chaque ligne de la table représente une règle et chaque colonne les valeurs possibles pour les expressions évaluées en entrée. Le groupe `relevant` du spécialiste `BadFuel` est défini comme suit :

```
(relevant Table
 (match(AskYNU? 'Is the car slow to respond')
       (AskYNU? 'Does the car start hard')
       (And (AskYNU? 'Do you hear knocking or pinging sounds')
         (AskYNU? 'Does the problem occur while accelerating'))
 with  (if T ? ? then -3
       elseif ? T ? then -3
       elseif ? ? T then 3
       else 1)))
```

La table de vérité correspondante (cf. Fig. 69) doit être lue de haut en bas. La première ligne, qui correspond à la situation décrite (les valeurs des expressions du "match", celles des colonnes 1, 2 , 3), fournit la valeur du groupe (colonne résultat). La connaissance diagnostique représentée est [Byl&85]:

> "If the car is slow to respond or if the car starts hard, then BadFuel is not relevant in this case. Otherwise, if there are knocking or pinging sounds and if the problem occurs while accelerating, then BadFuel is highly relevant. In all other cases BadFuel is only mildly relevant."

Relevant Knowledge Group

Expressions de la Table:
1: (AskYNU? 'Is the car slow to respond')
2: (AskYNU? 'Does the car start hard')
3: (And (AskYNU? 'Do you hear knocking or pinging sounds')
(AskYNU? 'Does the problem occur while accelerating'))

1	2	3	résultat
T	?	?	-3
?	T	?	-3
?	?	T	3
?	?	?	1

'?' signifie "valeur indifférente"

Fig. 69 : Table de vérité pour le groupe "Relevant"

Les groupes peuvent être hiérarchisés (appariement structuré ou "structured matching"), c à d. que les résultats des uns servent à déterminer les autres, comme dans le groupe `summary`, qui combine `relevant` et `gas` (ce dernier prend en compte la relation temporelle entre le dernier achat de carburant et le début du problème) :

```
(Summary Table
  (match relevant gas)
 with   (if 3 (GE 0)
        then 3
        elseif 1 (GE 0)
```

```
        then 2
        elseif ? (LT 0)
        then -3)))
```

Finalement, c'est la méthode "establish" qui renferme la conclusion à tirer des différents résultats par le spécialiste `BadFuel` :

```
(Establish (if (GE relevant 0)
               then (SetConfidence self summary)
               else (SetConfidence self relevant)))
```

où `summary` et `relevant` sont des knowledge groups.

Bilan

Toute la connaissance d'appariement invoquée lors de la classification est exprimée sous forme de règles. La hiérarchie des concepts ne sert pas à représenter des cas de maladies (ou de pannes) particuliers au sens de l'instanciation. S'il est question d'instanciation, c'est l'instanciation des concepts en tant que spécialistes capables de raisonner sur l'hypothèse qu'ils représentent. Une instance de spécialiste est alors capable de répondre aux messages *establish* et *refine* définis dans le concept. La hiérarchie des concepts est donc plutôt une structure d'organisation des connaissances diagnostiques (regroupement et hiérarchisation des règles), qui sert de canevas à l'affinement successif d'hypothèses, qu'une structure d'accueil pour la description de la maladie particulière d'un patient, qui sera enrichie au fur et à mesure de sa classification. Ce n'est pas la description du cas traité qui bénéficie de l'héritage et des inférences liés à la hiérarchie, mais c'est l'instance de spécialiste qui hérite des méthodes de raisonnement, et témoigne par son activation de l'état d'avancement du diagnostic.

Les conditions d'adéquation d'une hypothèse par rapport au cas traité ne sont plus posées en termes de valeurs et domaines d'attributs, mais uniquement en termes de règles associées à l'hypothèse.

2.2 La description de l'objet à classifier

Nous distinguons deux types de descriptions de l'objet à classifier :

- la description incomplète
- la description complète.

2.2.1 Description incomplète

La description incomplète repose sur la création de l'objet comme instance d'une classe appartenant à la hiérarchie. La description est alors donnée par cette classe. L'objet possède les attributs — valués ou non — définis ou hérités par sa classe d'instanciation. Le processus de classification est en fait un processus de *spécialisation* de l'instance incomplète qui tente de rattacher l'objet à des classes plus spécifiques que sa classe d'instanciation. Pour déterminer si l'objet peut appartenir à des classes plus spécifiques, d'autres informations — sous la forme de valeurs d'attributs — sont requises. Comment procéder pour obtenir ces informations, autrement dit comment peut-on compléter la description incomplète de l'objet afin de le classer correctement ? Il existe différentes approches, dont l'approche interactive (questions posées à l'utilisateur) est la plus simple. D'autres approches font intervenir des moyens de calcul ou d'inférence. Lorsqu'il s'agit de moyens de calcul définis dans une classe candidate, on peut s'interroger sur la légitimité de leur utilisation. Examinons ces problèmes dans le cadre des systèmes de la famille SHIRKA et de celle de MDX.

2.2.1.1 Le problème de l'obtention des valeurs manquantes — La famille SHIRKA

En **SHIRKA** [Rec&90], la classification se fait en spécifiant un objet existant et une classe (par défaut sa classe d'instanciation) qui sera le point de départ de la classification. Le système essaie de classifier l'instance dans le sous-graphe qui a pour racine cette classe. On part donc de la description de l'objet rendue possible dans le cadre de son instanciation. Cette description ne peut prendre en compte que les attributs hérités ou définis par la classe d'instanciation de l'objet. Nous avons vu que l'attribution du label "sûr" à une classe ne peut se faire que si la description de l'objet est complète pour cette classe, c. à d. que tous les attributs doivent être valués. Il y a alors plusieurs manières d'obtenir les valeurs d'attributs manquantes.

Le processus de classification est interactif. Il y a deux modes de fonctionnement du mécanisme de classification : avec inférence (mode par défaut) et sans inférence. Le procédé sans inférence demande systématiquement les valeurs des attributs inconnus à l'utilisateur. Le procédé avec inférence essaie de déduire d'abord les valeurs des attributs inconnus en utilisant les moyens d'obtention de valeurs définis dans la classe, dans les facettes `$sib-filtre`, `$sib-exec` (cf. I 2.4.2.3), et en dernier recours `$defaut`. En cas d'échec, l'utilisateur est sollicité. A la demande de l'utilisateur, le système fournit des informations sur l'attribut en question, comme le type requis, les domaines possibles,... en fonction des restrictions spécifiées dans les différentes sous-classes. Il s'agit de la réunion de tous les domaines (sous forme de listes de valeurs des différentes facettes `$domaine`, `$intervalle` etc.).

Ni l'un ni l'autre de ces procédés ne paraissent entièrement satisfaisants. Celui avec inférence utilise les moyens de calcul définis dans la classe (lorsqu'ils existent), donc

applicables aux seuls représentants de cette classe. Or, on les applique à un objet pour obtenir des valeurs qui serviront à déterminer si l'objet satisfait les conditions d'appartenance à la classe. Cette méthode est donc forcée de postuler l'appartenance de l'objet à la classe afin de l'établir ("pétition de principe").

Quant au procédé qui consiste à demander systématiquement la valeur à l'utilisateur, il s'agit du procédé le plus simple pour obtenir des valeurs manquantes, lequel a l'inconvénient d'être complètement interactif. Ce procédé est peu satisfaisant du fait de l'intervention maximale demandée à l'utilisateur. Le mérite de cette approche est qu'elle est correcte : elle ne commet pas la pétition de principe. Cependant, aucune distinction n'est faite entre la non-valuation d'un attribut et l'absence de cet attribut pour l'objet. Par exemple, l'absence de l'attribut "employeur" pour un enfant ou un chômeur n'est pas distinguée de l'absence de valeur pour cet attribut. Il serait intéressant de pouvoir indiquer que l'objet ne possède pas tel attribut, plutôt que de laisser sa valeur indéterminée. L'absence d'un attribut permettrait de qualifier une classe (et toutes ses sousclasses) d'impossible, alors que le manque de valeur la qualifie de possible.

[Rec91] reconnaît bien le problème du calcul des valeurs par les moyens spécifiés dans la classe :

> "It is therefore necessary to conclude that, during classification, either the value of a slot must be known at instantiation time or must be provided by the user. In no case can it be inferred in the knowledge base through classical procedural attachment."

Il propose alors comme solutions possibles d'une part une façon alternative de définir les méthodes de calcul et d'autre part une approche par point de vue qui est un premier pas vers TROPES (cf. infra).

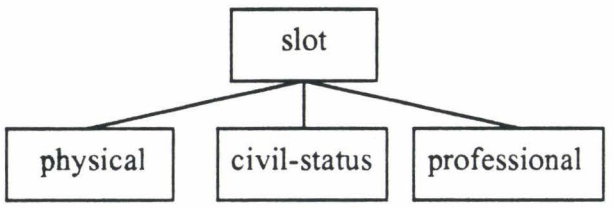
2.2.1.2 Une classification des méthodes de calcul de valeurs

La première idée, concernant les attachements procéduraux, est de ne plus les attacher directement aux slots dans les classes, mais de les décrire par des classes de méthodes, dans une hiérarchie séparée. Une telle hiérarchie est alors globalement associée à un slot dans un concept "père de famille", la famille étant la hiérarchie des classes où l'attribut est défini. Quand on a besoin de calculer la valeur d'un slot, on crée une instance de la racine de la hiérarchie des méthodes associée. Le contexte d'appel fournit alors les valeurs des slots correspondant aux paramètres d'entrée de la méthode. L'instance de méthode est ensuite classifiée dans sa hiérarchie en fonction des valeurs et de la disponibilité de ces paramètres. Le but est de rendre indépendants le choix de la méthode à exécuter et l'appartenance de l'instance à une classe donnée. En dissociant les méthodes des classes, une méthode ne pourra plus être appliquée à tort à un objet parce que l'on aurait postulé son appartenance à une classe. Par contre, le problème qui se pose, avant de calculer la valeur d'un attribut a, défini par une classe C, est de savoir si l'objet en question possède a, sans quoi il ne pourra appartenir à C. Ce problème-là n'est résolu ni en laissant les méthodes dans les classes, ni en les mettant en dehors. On peut se demander aussi si la définition d'une méthode ne dépend pas intrinsèquement de l'objet sur lequel on l'applique, donc de son type. Est-ce que l'on ne retrouve pas le même problème de l'appartenance de l'instance à une classe du côté des classes de méthodes ? On ne l'aura pas résolu pour autant, mais simplement déplacé. Un autre point obscur dans cette approche est le passage des paramètres d'entrée.

2.2.1.3 Vers une approche multi-points de vue

Les moyens d'inférence de valeurs d'attributs définis dans la classe ne peuvent pas être utilisés tant que l'appartenance de l'objet à cette classe n'est pas établie (classe "sûre"). Pour établir cette appartenance, [Rec91] propose d'utiliser la notion de point de vue. L'idée est de classifier une instance de façon partielle en se limitant à un point de vue correspondant à un sous-ensemble des attributs. On se ramène ainsi à une description complète de l'objet sous ce point de vue. L'appartenance à la classe étant ainsi établie (en respect des règles d'appariement, voir page 142) selon un point de vue, les valeurs des attributs regroupés sous d'autres points de vue peuvent être inférées en utilisant les moyens d'obtention définis dans la classe. Pour regrouper les attributs dans des points de vue différents, [Rec91] propose d'abord de définir pour chaque point de vue un type d'attribut (matérialisé par une sous-classe de slots) cf. Fig. 70. La classification pourrait alors ne prendre en compte que les attributs d'un certain type afin d'obtenir une classification sûre sous un point de vue. La fonctionnalité de restriction du processus de classification à un sous-ensemble d'attributs selon leur type est déjà offerte dans Shirka. (Elle n'est pas présentée comme une approche par points de vue dans le manuel [Rec&90]).

Dans cette solution, seule la hiérarchie des classes de slots reflète la structuration des objets en points de vue. Les classes de slots semblent d'ailleurs n'être rien de plus que des regroupements d'attributs, sans définition de caractéristiques propres.



Les attributs **weight** et **height** de person#51 sont instances de la classe d'attributs **physical**, qui détermine le point de vue physical sur person#51.

person#51	
/civil-status	
name	
age	25
sex	male
/physical	
weight	**70**
height	**165**
/professional	
salary	12000
employer	

Fig. 70 : Types d'attributs réflétant des points de vue

[Rec91] propose ensuite d'introduire une notion de points de vue reflétés dans la hiérarchie des classes elle-même, à la TROPES. TROPES sépare clairement les points de vue : il n'y a plus d'héritage multiple entre points de vue, ce qui force une organisation plus structurée de la hiérarchie des classes.

TROPES.

La notion de passerelle (uni-ou bi-directionnelle) constitue une solution originale au problème d'obtention des valeurs manquantes. D'une part, on diminue le nombre de valeurs d'attributs à inférer, car on prend des raccourcis entre points de vue qui permettent d'établir directement l'appartenance à une classe, et d'autre part, chaque nouvelle appartenance établie grâce à une passerelle autorise l'utilisation des moyens d'inférence définis dans la classe établie et permettent donc de déduire de nouvelles informations.

L'approche de TROPES allège la tâche de l'utilisateur : il n'a pas besoin de suivre un processus complet de classification partant de la racine et parcourant toutes les classes (cf. SHIRKA), en fournissant les valeurs de tous les attributs inconnus rencontrés. Au contraire, l'utilisateur indique lui-même le ou les points de vue qu'il connait le mieux, afin de n'avoir à fournir que les valeurs des attributs sous ces points de vue.

Sur l'exemple des automobiles [Mar91] (cf. Fig. 61 (page 143)), la classification d'une automobile avec une charge utile de 4 tonnes permet d'abord de la classer dans la classe Véhicule de Transport de Marchandises, et ensuite, grâce à la passerelle, dans la classe Economique, où elle hérite automatiquement de 4 roues, d'une carrosserie normale, et par la même occasion de 4 portes à poignée et 4 sièges simples. Tout cela est déduit de la seule information de la charge utile.

Cette méthode propose donc d'importantes améliorations par rapport à l'algorithme de SHIRKA :
- réduction de l'ensemble de valeurs d'attributs demandées à l'utilisateur
- l'utilisation des moyens d'obtention de valeurs définis dans les classes est devenue possible.

Cette méthode n'est valable que sous l'hypothèse de départ de l'équivalence des descriptions sous les différents points de vue et de la validité des passerelles.

2.2.1.4 Obtention des valeurs manquantes — la famille MDX

MDX

Dans MDX, les données sur lesquelles opèrent les règles sont fournies par le module de base de données PATREC (= Patient Record), en réponse à des requêtes de MDX. Ces données sont organisées dans une hiérarchie de frames représentant des concepts médicaux.

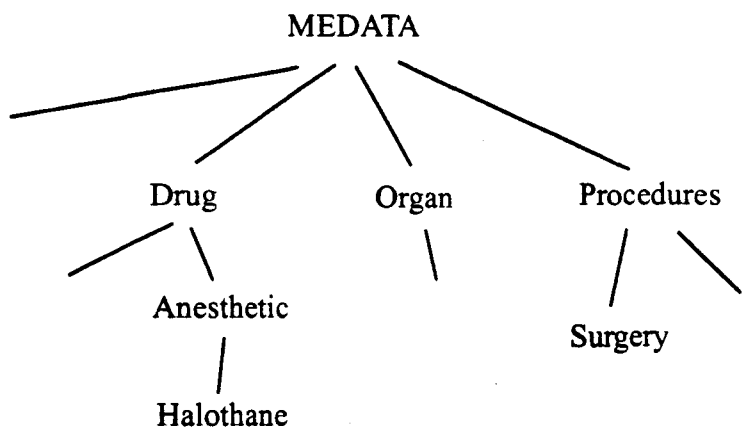


Fig. 71 PATREC : base de données patients

Des inférences de données sont effectuées à l'aide d'attachements procéduraux et d'héritage de valeurs par défaut. Elles permettent, par exemple, de déduire que des anesthésiants ont été administrés à partir de la donnée d'une opération importante. Ces inférences sont indépendantes du processus de classification. Un concept, par exemple "Foie", dans la structure de classification a d'autres caractéristiques que le concept du même nom dans la base de données. La structure de classification est censée refléter la connaissance du

diagnostiqueur, alors que celle de la base de données des patients reflète la connaissance (d'une expertise plus élémentaire) de l'infirmière. Les inférences locales de la base de données intelligente correspondent à une tâche générique : l'inférence de données (ou "knowledge-directed information passing").

CSRL

En CSRL, les données requises par les prémisses des règles peuvent soit être obtenues directement en posant des questions à l'utilisateur (cf. `Relevant` knowledge group page 156), soit par des envois de requêtes à une base de données, comme dans [By&89] (cf. Fig. 72), ou encore dépendre des résultats obtenus par d'autres "knowledge groups".

PhysicalEvidence Knowledge Group

Prémisses :
1: (Invoke ($PATREC) Ask (QUOTE cholangitis)(QUOTE present?))
2: (Invoke ($PATREC) Ask (QUOTE abdomen) (QUOTE (pain quality)))
3: (Invoke ($PATREC) Ask (QUOTE vomiting) (QUOTE present?))

1	2	3	value
(eq yes)	(eq colicky)	?	HighlyLikely
?	(eq colicky)	(eq T)	Likely
?	(eq colicky)	T	SlightlyLikely
(eq yes)	?	?	SlightlyLikely

Fig. 72 Requêtes à la base PATREC pour obtenir des informations manquantes

Nous avons vu qu'il ne s'agit pas tant de classifier une instance de maladie (ou de panne), mais plutôt d'instancier des spécialistes correspondant à des hypothèses, chargés de s'établir ou de se rejeter. Le problème des valeurs manquantes ne se pose donc pas du tout de la même façon que pour les systèmes étudiés précédemment. Les conditions d'adéquation d'une hypothèse par rapport au cas traité ne sont plus posées en termes de valeurs et domaines d'attributs (que ce soient des attributs obligatoires ou facultatifs), mais uniquement en termes de règles associées à l'hypothèse. Ces règles peuvent faire appel à toutes sortes d'informations dans leurs prémisses. Ce sont alors les données intervenant dans les prémisses des règles qui doivent être rendues disponibles à travers des requêtes, adressées soit à l'utilisateur, soit à une base de données. Les données sont donc soit éparpillées dans une base de données, soit dans la tête de l'utilisateur, mais à aucun moment on ne dispose d'une description synthétique de "l'objet" en classement sous la forme d'une instance avec des attributs, appartenant à une ou plusieurs classes.

2.2.2 Description complète

Les descriptions incomplètes sont fondées sur la contrainte de l'instanciation, qui est la condition de description d'un objet dans un système à base de classes. Seuls les systèmes de la famille KL-ONE sont affranchis de cette contrainte, car un concept individuel est décrit par les assertions faites à son propos. Le caractère complet de la description des concepts n'est d'ailleurs pas toujours considéré comme un avantage. Dans [Fin86], une critique est faite de la description complète d'un nouveau concept requise avant sa classification. L'auteur propose une approche interactive de la classification.

D'autres systèmes contournent l'instanciation comme base minimale de la description de l'objet à classer et offrent un moyen de fournir une description "complète". SHOOD, par exemple, adopte une instanciation "a posteriori", selon les résultats de la classification.

2.2.2.1 Les systèmes à subsomption sont des systèmes de description

Les systèmes à subsomption sont affranchis de la contrainte d'instanciation, car les concepts ne sont pas instanciés mais décrits par des assertions. Ces assertions concernent leurs relations avec d'autres concepts individuels (comme valeurs de ses roles) ou l'appartenance à des concepts génériques. Les assertions concernent les *faits* et appartiennent à la *ABOX*[1], contrairement aux définitions des concepts génériques qui forment la connaisance *terminologique*, exprimée par les primitives de la *TBOX*. Les concepts génériques sont ordonnés entre eux par le classificateur selon la relation de subsomption. Les concepts individuels — "instances" de concepts génériques — sont subsumés par les concepts génériques auxquels ils appartiennent, mais ne subsument aucun concept. Le classificateur peut alors trouver leur place dans la hiérarchie des concepts suivant cette relation de subsomption. Les concepts individuels seront toujours placés en feuilles. Il ne s'agit donc pas vraiment de classification d'instances, mais plutôt d'un cas particulier de la classification de concepts — prise en charge par le classificateur —, qui est la classification de concepts *individuels*. Il suffit de trouver les subsumants les plus spécifiques [Woo91], les concepts individuels n'ayant pas de subsumé. Ces concepts individuels ne sont pas *instances* de concepts génériques (au sens de l'instanciation en PPO), mais en sont des spécialisations, au sens de la subsomption. Le seul critère de placement est donc la relation de subsomption, et le problème des valeurs manquantes est hors propos.

Le placement des concepts individuels est ajusté par le "recognizer" ou "realizer" [McG88] dès que le besoin s'en fait sentir, c. à d. dès qu'une nouvelle définition de concept ou une assertion vient changer les relations de subsomption. De tels ajusteurs fonctionnant par chaînage avant existent en KL-TWO, BACK, Classic et LOOM [McG91].

L'approche la plus simple est de créer une abstraction à partir du concept individuel [Vil85] et de classifier celle-ci parmi les concepts génériques :

> " "to find those concepts which are matched by an individual *I*, we form an abstraction *A* of *I*, and then classify *A*. *I* necessarily matches all concepts which subsume *A*." " [McG88].

Exemple de classification de concept individuel suivant ce procédé en KL-TWO :

1. La distinction entre définitions et faits en TBOX et ABOX a été introduite pour la première fois dans KRYPTON [Bra&83].

Les assertions suivantes sont ajoutées à la base PENNI[1] :

```
(BIG-ANIMAL a1)
(ANIMAL a2)
(EATS a1 a2)
```

L'ajusteur forme un concept abstrait `Ca1` pour le concept individuel `a1`, qui constitue sa Généralisation la Plus Spécifique (GPS) :

```
(CMeet BIG-ANIMAL (CMin (VRDiff EATS ANIMAL) 1))
```

dont la sémantique est définie par

```
λ x. (BIG-ANIMAL x) & ∃ y (EATS x y) & (ANIMAL y)
```

L'application de cette λ-expression au concept individuel `a1` donne la proposition suivante :

```
(BIG-ANIMAL a1) & ∃ y (EATS a1 y) & (ANIMAL y)
```

La classification de `Ca1` (prise en charge par le classificateur de NIKL) entraîne l'identification de sa forme canonique :

```
(CMeet Carnivore BIG-ANIMAL)
```

qui est justement la définition du concept `BIG-CARNIVORE` (présent dans la base NIKL). L'ajusteur termine donc par l'ajout de l'assertion suivante à la base :

```
((BIG-ANIMAL a1) & (EATS a1 a2) & (ANIMAL a2)) ⇒ (BIG-CARNIVORE a1)
```

Celle-ci permet d'inférer `(BIG-CARNIVORE a1)`.

On peut se poser la question de l'utilité d'une telle abstraction intermédiaire, surtout si celle-ci n'est pas suffisamment complète pour trouver tous les concepts génériques subsumant *I* ([McG88]). Au lieu de travailler avec une abstraction (la plus) complète (possible), l'ajusteur de LOOM construit une abstraction partielle de façon incrémentale et interroge le concept individuel d'origine pour obtenir les détails nécessaires à sa classification. Cette abstraction intermédiaire est enrichie en fonction des informations contenues dans trois listes attachées à l'objet : HITS, MISSES et TYPE. Elles contiennent respectivement les expressions vérifiées pour l'objet, les expressions non vérifiées et son type courant, constitué de la liste des concepts qu'il vérifie. Si à l'issue d'un changement dans la base, certaines expressions de la liste des HITS deviennent non vérifiées pour l'objet, ou certaines expressions dans les MISSES deviennent vraies, le type est recalculé. Si la modification n'a aucune conséquence sur les HITS et MISSES, le type de l'objet reste inchangé. Cette technique est surtout présentée comme une optimisation de l'algorithme d'ajustement/classification utilisé par KL-TWO et BACK [McG91], mais on peut y voir aussi un moyen d'attacher à l'instance des informations intermédiaires entre concepts de la hiérarchie. L'objet appartenant au concept C et ne possédant pas encore toutes les propriétés nécessaires et suffisantes pour appartenir au concept C' peut "garder de côté" celles qu'il possède déjà en attendant d'obtenir les autres.

1. PENNI constitue la base de concepts individuels (la *ABOX*) de KL-TWO. La *TBOX* est celle de NIKL.

2.2.2.2 SHOOD

L'algorithme de classification de SHOOD travaille avec une description "complète" de l'objet, dans le sens où aucune tentative n'est entreprise pour enrichir l'objet au fur et à mesure du processus de classification, ni en utilisant des moyens d'inférence, ni en posant des questions à l'utilisateur[1]. Tous les éléments d'information utilisés sont disponibles dans la commande de classification (MIC). Ensuite, il s'agit simplement de rejeter les classes impossibles et de calculer les degrés de cohérence et de complétude pour les autres, en fonction des paramètres fournis.

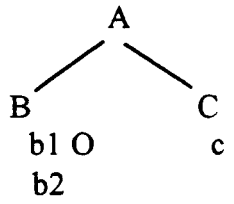
Le problème de l'obtention des valeurs manquantes ne se pose donc plus.

Pourtant, il y a quelques inconvénients à décrire ainsi l'objet à classer. L'utilisateur doit fournir directement les valeurs des attributs à prendre en compte dans la classification. Il faut donc qu'il connaisse ces attributs, leur syntaxe et leur domaine de valeurs. C'est une contrainte d'autant plus forte qu'en oubliant de valuer un attribut défini obligatoire à quelqu'endroit dans la hiérarchie, toute une partie du graphe risque d'être qualifiée d'impossible.

Avant de lancer une commande de classification, l'utilisateur doit donc bien connaître la hiérarchie avec ses attributs obligatoires et ses contraintes fortes.

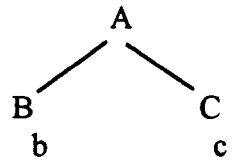
Le problème des attributs "inexpliqués"

Un autre problème qui peut surgir est celui des attributs "inexpliqués" : que fait-on des attributs valués dans la commande de classification, qui s'avèrent n'appartenir qu'à des classes non-retenues ? Comment l'instance pourra-t-elle garder un attribut qu'aucune de ses classes ne définit ?



Exemple : Mic new depuis A [b2 = v1, c = v2]
résulte en un rejet de la classe B, car b1, attribut obligatoire n'est pas valué. L'objet à classer est décrit avec les deux attributs b2 et c valués. La classe possible C n'en définit qu'un. Où est exprimée l'incohérence de *la classe* par rapport à l'objet?

Analogue est le problème suivant :



Mic new depuis A [b = v1, c = v2].
Les deux classes B et C sont retenues. Seulement, seule l'instanciation multiple de l'objet dans B *et* C pourra garantir la possession des deux attributs b et c. Ceci n'est pas exprimé.

1. Dans [Ngu&92] il est question d'inférences attachées aux attributs pour calculer leurs valeurs. Une inférence "user" permettrait de demander une valeur à l'utilisateur. Il n'est cependant pas clair de savoir à quel moment ces inférences peuvent être déclenchées, ni si elles jouent un rôle dans la classification.

2.2.2.3 CLASSIC

En CLASSIC [Gra88], la description de l'objet à classer n'est pas contrainte par l'instanciation. L'utilisateur décrit librement son objet à l'aide d'un éditeur et value les attributs qu'il veut, indépendamment des classes qui les définissent.

Pour les champs dont la valeur est inconnue lors du processus de classification, il existe différents moyens d'obtention des valeurs.

Il y a trois facettes pour les champs numériques et symboliques : Valeur à demander (oui non), Valeur à calculer (expression) et Valeur par défaut.

Les valeurs de certains champs peuvent être déduites par des règles d'inférence.

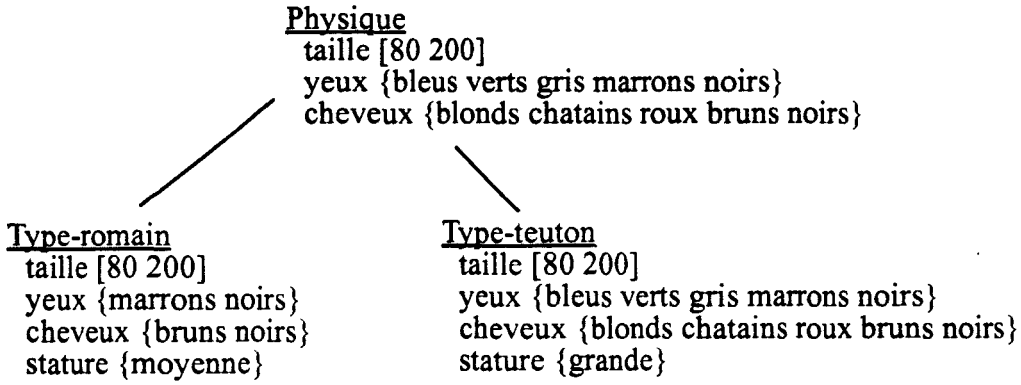


Fig. 73 CLASSIC : hiérarchie de types de figurants

Dans l'exemple de la Fig. 73, les champs `taille`, `yeux`, `cheveux` sont déclarés demandables. La valeur du champ `stature` est déduite de la taille à l'aide de règles d'inférence. Ces règles sont attachées à la superclasse des classes pour lesquelles le champ est discriminant (ici `Physique`).

Exemples:

```
Règle moyenne
...
classe : Physique
si : taille ~< [2] 175 [1]
alors: stature = moyenne [1]
...
Règle grande
...
classe : Physique
si : taille ~> [2] 173 [1]
alors : stature = grande [1]
...
```

Dans la partie prémisse de la règle, on peut utiliser des prédicats numériques flous, qui autorisent une certaine tolérance dans la comparaison de la valeur d'un champ et la valeur seuil. Ce sont les prédicats ~=, ~<, ~>, près-de, loin-de. Ils sont utilisés suivant le modèle :

valeur-numérique prédicat-flou [tolérance] seuil

ou: valeur-numérique prédicat-flou [pourcentage %] seuil.

Un calcul élaboré faisant intervenir des coefficients d'importance de chaque prémisse de la règle (entre [] à la fin de chaque prémisse) et des coefficients de confiance de chaque valeur d'attribut (entre [] dans la partie "alors" d'une règle), permet de décider de déclencher ou non une règle et d'attribuer un coefficient de confiance aux valeurs des champs modifiés dans la conclusion.

Quand un champ n'est pas valué, les facettes sont examinées pour trouver une valeur, dans l'ordre : Valeur à demander, Valeur à calculer et Valeur par défaut.

En ce qui concerne la description de l'objet au départ, elle a l'avantage de ne pas être limitée par l'instanciation. Tous les attributs sont mis à plat et peuvent être valués indépendamment de l'appartenance à une classe. Contrairement à SHOOD, où le processus de classification travaille avec les seules informations données au départ, CLASSIC complète la description de l'objet durant la classification. Malheureusement, les moyens d'obtention des valeurs manquantes n'assurent pas toujours la cohérence. Les facettes "Valeur à calculer" et "Valeur par défaut" attachées à un attribut dans une classe sont utilisées pour les objets avant de savoir si l'objet représente cette classe. L'emploi des règles d'inférence paraît plus correct, car elles sont déclenchées une fois la classe acceptée. On ne sait pas comment est réglé le problème des "attributs inexpliqués" (cf. SHOOD). Le degré d'incompatibilité d'une classe pour l'objet ne concerne pas les attributs inexpliqués de l'objet, mais les valeurs d'attribut qui ne correspondent pas aux contraintes de la classe. La référence est toujours la classe. Comme pour SHIRKA, SHOOD et TROPES, il n'y a pas de distinction entre le fait de ne pas avoir un attribut et le fait que la valeur de cet attribut soit inconnue. Les seuls cas pris en compte sont que l'attribut a une valeur ou n'en a pas (encore). S'il n'en a pas, la classe est seulement possible (SHIRKA, TROPES), ou incomplète (SHOOD) ou un peu moins compatible (CLASSIC). Si, par contre, il existait un moyen d'indiquer l'absence de l'attribut (cf. page 159), la classe serait impossible en SHIRKA, TROPES et SHOOD en raison d'une violation de contrainte et incompatible à 100% en CLASSIC[1].

Il semble que la description de l'objet est gérée indépendamment de son appartenance à une classe. Il s'agit plutôt d'identifier l'objet à travers la classification que de trouver une classe appropriée à laquelle il pourra être rattachée par instanciation.

Le problème des attributs "inexpliqués" — RED-2

Le problème des "attributs inexpliqués", évoqué pour SHOOD (page 165) et CLASSIC, peut se traduire dans le contexte de CSRL en un problème de symptômes restés inexpliqués. En effet, les spécialistes de la hiérarchie arrivent à obtenir les informations nécessaires pour l'établissement ou le rejet de leurs hypothèses, mais il y a peut-être des informations supplémentaires disponibles qui ne sont prises en compte par aucun spécialiste (soit parce que non prévu par l'ensemble des spécialistes de la hiérarchie, soit parce que le rejet d'un spécialiste a entraîné le rejet d'un sous-spécialiste qui aurait pu rendre compte de ces informations). L'établissement d'un diagnostic ne donne aucune information sur d'éventuels symptômes restés inexpliqués.

Une approche qui prend peut-être en compte ces problèmes est celle de la tâche d'"abduction d'hypothèses composites", appliquée dans le système RED-2 [Jos&87]. Le but

1. Ou moins, si les coefficients d'importance interviennent dans le calcul.

de cette tâche est de construire une hypothèse qui explique au mieux un ensemble d'observations données au départ. L'algorithme proposé est le suivant :

- Initialiser l'ensemble des hypothèses retenues à vide
- Répéter jusqu'à ce qu'il ne reste plus rien à expliquer, ou que l'on ne puisse plus rien expliquer:
  - Choisir une observation inexpliquée (initialement elles le sont toutes)
  - Retenir l'hypothèse la plus plausible qui explique cette observation. Si aucune hypothèse plausible n'est trouvée, noter l'observation comme inexplicable.
  - Inclure l'hypothèse dans l'ensemble retenue
  - Calculer ce que l'ensemble des hypothèses retenues sait expliquer maintenant et déterminer ce qui reste à expliquer.
- Retourner l'ensemble des hypothèses retenues.

Il s'agit en fait de produire une hypothèse "composite", qui combine l'ensemble des hypothèses en éliminant les parties superflues. Les différentes hypothèses partielles sont trouvées par classification.

L'idée d'identifier les observations restées inexpliqués peut être intéressante, transposée dans le cadre d'un système de classification d'un objet (complètement) décrit au départ. Le résultat de sa classification pourra distinguer clairement entre la partie de l'objet "expliquée" (décrite) par la ou les classe(s) retenue(s) et les attributs restés orphelins.

2.3 Le contrôle : centralisé ou distribué

Le contrôle du processus de classification peut être pris en compte de façon centralisée par un algorithme global de classification, qui opère un parcours du graphe des classes, ou au contraire, être conçu de manière orientée objet. La conception orientée objet résulte en une répartition de l'algorithme sur les objets intervenants, qui contribuent selon leurs compétences (méthodes locales) au résultat global.

2.3.1 Algorithme global de classification

La majorité des systèmes passés en revue définissent un algorithme de classification fixe (abstraction faite du paramétrage) et global. En effet, les systèmes de la famille SHIRKA et les langages terminologiques sont avant tout des systèmes de représentation des connaissances, et non pas des langages de programmation. SHIRKA et TROPES séparent résolument la connaissance représentée et le raisonnement qui porte sur elle. L'algorithme de classification est donc défini complètement en dehors des schémas (classes et instances) qui représentent la connaissance.

L'algorithme de base [Lip82] des langages terminologiques est fondé sur la subsomption. Il s'agit avant tout de la classification de concepts génériques. Il est composé en général de deux étapes :

(1) la recherche des subsumants les plus spécifiques
(2) la recherche des subsumés les plus généraux

La première étape est menée en profondeur d'abord en commençant par le concept le plus général et compare successivement le concept à insérer aux concepts déjà présents, à l'aide du test de subsomption. La deuxième étape examine successivement les descendants des subsumants les plus spécifiques trouvés dans la première étape. Différentes variantes existent sur cet algorithme de base : selon le parcours du graphe effectué, certains tests de subsomption peuvent être économisés [Fin86] [McG88] [Baa&92]. La brique de base de l'algorithme de classification est en effet le test de subsomption entre deux concepts, qui est lui-même composé de tests de subsomption entre rôles (2.1.2.2). Comme la définition d'un concept n'est rien d'autre que la composition d'autres concepts [Bra85], la comparaison de deux concepts passe par la comparaison de leurs parties respectives. Woods a évoqué dans [Woo91] la diversité des règles qui gouvernent la subsomption entre relations (rôles) de types différents. Pour chaque type de relation, on peut définir la relation de subsomption, qui n'est rien d'autre qu'une relation d'ordre partiel.

Dans une approche orientée objet, cet état de fait peut directement être exploité pour distribuer le processus de classification sur les objets intervenants, sous la forme de méthodes propres à chaque type d'objet.

2.3.2 Conception orientée objet de la classification

L'appariement entre concepts (qu'il s'agisse de l'appariement de concepts génériques ou de l'appariement entre un concept générique et un concept individuel) peut facilement se concevoir comme un test local aux concepts concernés, qui contiennent tous les éléments nécessaires à leur comparaison. Nous avons appliqué cette approche dans le filtrage

d'instances (cf. I 4.4.2), où l'appariement avec une classe filtre est opérée par cette dernière, qui possède une méthode appropriée.

L'appariement local est également effectué dans les systèmes de la famille MDX (cf. 2.1.3.3), dont l'implantation est facilitée par l'utilisation d'un langage hybride disposant de méthodes (LOOPS). La classification est définie de manière distribuée par des méthodes d'appariement et de propagation (*establish/refine*) associées aux "spécialistes" organisés en hiérarchie.

La subsomption des langages de description de concepts a été reprise par Napoli dans le cadre d'un langage de programmation objet basé sur l'approche par prototype (YAFOOL). Il définit une forme de subsomption pour objets, appelée la O-subsomption [Nap&92], comme un ordre partiel sur les objets, en fonction de la relation d'ordre sur leurs propriétés. La subsomption sur les propriétés dépend du type de celles-ci et compare les facettes. Napoli montre que chaque relation d'ordre peut induire un raisonnement du même type que la subsomption sur la relation de généralisation/spécialisation de concepts. Il distingue en particulier la relation de composition structurelle, qui permet d'organiser composants et composés en un graphe (ordre partiel). La relation de subsomption associée est appelée "co-subsomption". L'application de ces deux types de subsomption dans un système d'aide à la synthèse de molécules organiques, appelé YCHEM, est décrite dans [Nap92]. Les objets (molécules, liaisons) sont organisés selon les différentes relations de subsomption, qui peuvent être vues comme autant de "dimensions" de représentation et de classification. Les tests de (co-)subsomption sont implantés de manière répartie comme des méthodes spécialisées pour les objets, leurs attributs et leurs facettes.

L'avantage majeur de l'approche orientée objet est de pouvoir définir localement les règles de la subsomption, comme il est fait en ProObj (cf. page 24), où l'utilisateur peut spécifier de nouvelles facettes d'attribut, en y associant une méthode de subsomption (prédicat Prolog) adaptée. Le système peut ainsi être étendu à volonté et l'expressivité peut en être augmentée tout en prenant en compte les nouvelles caractéristiques dans la classification. ProObj prévoit de plus d'exclure certains types d'attributs ou de facettes de l'appariement.

L'implantation orientée objet d'un langage de description de concepts avec classifieur a été expérimentée par Yelland en Smalltalk [Yel92]. Rathke a implanté un protocole de subsomption à la KL-ONE distribué sur les objets en CLOS [Rat93].

2.4 Conclusion

2.4.1 Appariement

La comparaison des objets aux classes pour décider de l'appartenance des premières aux secondes est liée à l'interprétation de la notion de classe. Si la classe est interprétée comme une définition, la comparaison donne un résultat sûr, à condition de disposer de tous les éléments de la description. C'est le cas des classes "sûres" en SHIRKA et TROPES, et des concepts définis des langages terminologiques. Pour ces derniers l'appariement est fondé sur la subsomption entre concepts. Si les classes ne sont pas interprétées comme des définitions, l'appariement donne une mesure de la correspondance entre la classe et l'objet (SHOOD, CLASSIC).

2.4.2 Description

La description de l'objet à classifier se fait dans le cadre de l'instanciation dans la majorité des langages à base de classes et instances (SHIRKA, TROPES). Après l'instanciation minimale dans la classe de départ, cette description est complétée au fur et à mesure des besoins de la classification. Les langages terminologiques sont des langages de description, où les objets sont décrits librement, avant d'être classifiés dans la hiérarchie. Ces descriptions sont complètes, car il n'y a pas de complétion lors du processus. L'approche par prototype est comparable à celle des langages terminologiques, car le prototype n'est pas considéré comme un moule stricte des prototypes qui le spécialisent. En Yafool, les prototypes "instances" peuvent posséder des caractéristiques que n'ont pas les prototypes "classes" dont ils dérivent. La définition de la O-subsomption est donc semblable à celle de la subsomption des langages terminologiques. Le seul langage à base de classes et instances à retenir la description libre est Shood. Le système dédié à la classification CLASSIC (ILOG) autorise également la description libre, combinée avec la complétion en cours de classification.

2.4.3 Classification

La majorité des systèmes étudiés appliquent un processus de classification global et fixe. La seule approche répartie et orientée objet de la classification est décrite par Napoli. Il existe pourtant des tentatives de définition répartie (redéfinissable) du test de subsomption (ProObj, FrameTalk [Rat93]), qui sont un premier pas à la programmation orientée objet de la classification.

2.4.4 Homonymie des attributs

La possibilité d'homonymie des attributs définis dans des classes différentes est une caractéristique des systèmes de représentation "centrée objet", dans le sens où un attribut est défini dans le contexte de la classe et a un sens pour les instances de cette classe. Nous constatons que l'homonymie n'est pas gérée dans le cadre de la classification par les systèmes que nous avons examinés.

Dans la première version de KL-ONE, les rôles étaient définis dans les concepts et avaient un sens par rapport à ceux-ci. Goodwin évoque dans [Goo79] les problèmes de définition autour de rôles homonymes. Pour lui, deux rôles de même nom hérités de deux concepts

différents doivent être considérés comme distincts. S'il s'agit de la même propriété, celle-ci doit alors être définie dans un ancêtre commun. Les définitions (affinements) doivent alors être fusionnées et ne pas être en contradiction[1]. L'algorithme de subsomption de Lipkis [Lip82] prévoit l'appariement de rôles **issus d'un même rôle** (par *diffs* ou *mods*, c. à d. par décomposition en sous-rôles ou restriction de valeur ou de cardinalité). C'est la descendance d'un rôle commun qui détermine qu'un rôle a le même **sens** qu'un autre rôle, et non pas l'ensemble des facettes[2], ni même le nom du rôle, qui ne porte pas d'information sémantique (toujours selon [Lip82]). L'homonymie des attributs semble donc avoir été prévue à la base, mais on ne retrouve pas ses traces dans les systèmes plus récents.

Les descendants de KL-ONE définissent les rôles de manière globale comme des relations (cf. I 2.1.1.3). Les relations étant sorties des concepts, leur sens n'est plus déterminé par le concept, mais il est devenu global. L'homonymie est assimilée à la synonymie dans ces systèmes. Les rôles sont définis (*introduits*) par des opérateurs d'introduction de rôles, avant de pouvoir être utilisés dans la description d'un concept. L'introduction d'un rôle consiste à associer une définition (terme) à un symbole (identificateur, nom). Dans plusieurs systèmes (NIKL, LOOM, BACK), la définition du rôle est l'association d'un domaine et d'un co-domaine à un nom de rôle (introduction de rôle primitif)[3]. Ce nom peut être utilisé dans la description de concepts (subsumés par le domaine spécifié), où l'on peut affiner le rôle, en conformité avec le co-domaine d'origine. Tout affinement d'un rôle défini globalement est donc bien une spécialisation d'un seul et même rôle et toute homonymie se résume à la synonymie. Un concept héritant de plusieurs affinements d'un même rôle fusionne ces affinements[4]. La gestion de l'homonymie de rôles comme propriétés distinctes quand elles sont associées à des domaines différents nécessiterait la possibilité de manipuler à la fois le nom et le domaine d'un rôle pour pouvoir désigner des rôles homonymes de manière non-ambiguë. Le système BACK autorise l'affinement des domaines de rôles [Hop&93], mais l'association d'une définition à un identificateur de rôle ne pouvant être faite que de manière unique, l'introduction de rôles homonymes ne semble pas être prévue. Contrairement à [Lip82], le nom de rôle est devenu porteur de sémantique et détermine à lui seul l'identité d'une propriété.

Dans SHIRKA et TROPES, tous les attributs homonymes ont le même sens à l'intérieur d'une "famille" (concept), ce qui revient à assimiler l'homonymie à la synonymie et à imposer la même sémantique (unification sémantique au sens d'OBJLOG [Dug88]). Chaque instance ne pouvant appartenir qu'à une seule famille, il est impossible pour une instance d'avoir des attributs homonymes distincts.

SHOOD prône l'utilisation des noms complets, combinant le nom de l'attribut avec le nom de la classe d'origine de sa définition, pour distinguer les slots homonymes [Esc&90], mais l'homonymie n'est pas évoquée dans le cadre de la classification. En particulier, lors de la valuation d'attributs dans la description d'une instance à classer, la prise en compte de

1. Goodwin préfère ne pas utiliser l'affinement multiple pour éviter des problèmes de conflit d'héritage multiple.
2. Lipkis donne l'exemple du rôle mère de Personne dont on ne peut dire qu'il est en relation avec le rôle épouse du concept Homme, juste parce que les deux rôles doivent être de type Femme.
3. Dans CLASSIC, il n'y a pas d'association de domaine ni de co-domaine lors de l'introduction d'un rôle [Res&93].
4. S'il existe des contradictions, le concept est "impossible" : il n'aura pas de subsumés et son extension est vide.

l'homonymie nécessiterait soit l'utilisation des noms complets pour une désignation non-ambiguë des attributs homonymes, soit la gestion de l'ambiguïté par le processus même de classification. Dans le premier cas, la donnée d'un nom complet supposerait de connaître la classe avant d'avoir classifié l'instance. Dans le deuxième cas, si les attributs homonymes sont de types différents, la valeur spécifiée permettrait éventuellement de réduire les possibilités. Dans le cas d'attributs homonymes de même type, la classification pourrait retenir plusieurs classes comme "possibles".

En YAFOOL, chaque attribut (défini globalement comme un objet sous son nom d'attribut) possède une facette `liste-frame`, qui contient l'ensemble des objets qui fixent cet attribut, c. à d. qui contiennent une définition de lui. Celle-ci est utilisée dans le filtrage (cf. page 64) et également dans la classification [Nap&91], pour délimiter l'espace de recherche. La définition des attributs au niveau global a pour conséquence l'absence d'homonymie. Tous les conflits liés à l'héritage multiple sont des conflits de *valeur* (cf. I 2.3.4.1).

Dans le chapitre suivant, nous proposons la classification en FROME, fondée sur la description libre des objets, l'appariement en fonction de la structure et des valeurs d'attributs, et la distinction des classes définies et descriptives. Nous traitons également l'homonymie des attributs dans le cadre de la description libre des objets, ainsi que sa prise en compte dans la classification.

3
La classification en FROME

3.1 Introduction

Dans ce chapitre, nous proposons un système de classification des instances pour et en FROME, guidés par les problèmes soulevés dans le chapitre précédent, autour des points suivants :

(1) la comparaison de l'objet aux classes
(2) la description de l'objet
(3) l'homonymie des attributs

Le contrôle (distribué et métaprogrammé) du processus de classification est abordé en 3.8.

La classification que nous proposons pour FROME comporte des éléments d'inspiration provenant d'une part des **langages de frames à base de classes** (que nous appelons l'approche objet dans la suite) et d'autre part des **langages terminologiques** (approche terminologique).

Premièrement, nous mettons l'accent sur une **classification fondée autant sur la structure que sur les valeurs**. Les outils de classification de l'approche objet étudiés dans le chapitre 2 classifient les instances selon les valeurs de leurs attributs. La structure même de l'objet, c.-à-d. le fait d'avoir ou non l'attribut indépendamment de sa valeur, n'intervient pas. Dans les approches interactives, on est alors amené à demander la valeur des attributs sans se poser la question de savoir si ceux-ci ont un sens pour l'objet. L'approche terminologique prend en compte tous les descripteurs de rôles (type, restriction de type, valeur...) définis dans le langage (cf. par exemple le tableau page 24).
Nous étendons la classification dans l'approche objet pour prendre en compte également la structure, en termes d'informations sur la présence ou l'absence de certains attributs. Ces informations peuvent être utiles aussi bien pour établir l'appartenance d'un objet à une classe que pour la réfuter. La structure joue donc un rôle important à la fois dans la description de l'objet et dans l'appariement.

L'appariement

En ce qui concerne la comparaison de l'objet aux classes ou appariement, nous avons vu qu'il existe une multitude de modes de décision (cf. 2.1). L'appariement peut être qualifié de possible, sûr ou impossible, être mesuré de façon numérique, en fonction de la vérification des contraintes sur les attributs et éventuellement en faisant intervenir des qualifications d'attributs (obligatoires, facultatifs). Nous pouvons constater que ces différents modes sont en fait fortement liés à l'interprétation de la notion de classe. Ainsi, l'interprétation de la classe comme une **description** (approche objet), exprimant des conditions nécessaires d'appartenance, ne peut mener à un appariement certain. Dans un environnement descriptif, on peut déduire de façon automatique que l'appartenance d'un objet à une classe est possible ou impossible. L'appartenance sûre est établie par un acte volontaire. A l'inverse,

l'interprétation de la classe comme une **définition** (approche terminologique) permet de déduire l'appartenance de façon certaine, puisque la définition comporte à la fois les conditions nécessaires et suffisantes.

Les conditions d'appartenance sont exprimées dans la description des attributs de chaque classe. En général, il s'agit des descriptions de types et de domaines des valeurs de ces attributs et des contraintes que ces valeurs doivent respecter. Nous introduisons différents types d'attributs, selon l'importance qu'ils ont pour l'identification d'une instance de la classe. Certains attributs devront ainsi nécessairement recevoir une valeur, alors que pour d'autres il suffira de déclarer qu'ils ont un sens pour l'objet. D'autres encore n'auront pas besoin d'être mentionnés pour pouvoir décider de l'appartenance. L'appariement pourra se fonder ainsi autant sur la structure (pertinence des attributs) que sur les valeurs.

Nous étudions la classification selon la structure et les valeurs successivement dans un environnement descriptif (3.3), dans un environnement définitoire (3.4), pour ensuite intégrer les deux approches dans un système unique (3.5). Dans cette approche mixte, nous obtenons des résultats d'appariement tri-valués : sûr, possible, impossible.

La description

Dans les systèmes étudiés au chapitre 2, la description de l'objet à classifier est libre (potentiellement complète) ou liée à l'instanciation (incomplète). La description incomplète pose le problème de l'obtention des informations manquantes, nécessaires aux procédures d'appariement (cf. 2.2.1).

Nous retenons de l'approche terminologique **la description libre** des objets en termes de structure et de valeurs d'attributs.

Homonymie

FROME offre la possibilité d'exprimer et de manipuler des caractéristiques homonymes, comme nous l'avons vu dans I 3.3.2. On appelle **attributs homonymes** des attributs distincts possédant le même nom, mais des définitions différentes dans des classes indépendantes[1]. En principe il n'y a aucun lien entre attributs homonymes, leurs noms étant accidentellement identiques. Chacun de ces attributs a un sens par rapport à la classe qui le décrit. L'homonymie des attributs constitue un paramètre important lors de la classification d'instances. En effet, la description libre d'un objet à classifier se fait en termes d'attributs, référencés par leurs noms. La description de l'objet est donc potentiellement ambiguë, due à l'homonymie de certains attributs. Deux approches sont possibles pour traiter l'homonymie des attributs :

(1) Lors de la description, la spécification d'un attribut doit être non-ambiguë. Il faut alors un moyen adéquat de désignation non-ambiguë, telle que la sélection de point de vue (spécification d'une classe). Il n'y a pas d'ambiguïté à gérer par la classification dans ce cas.

(2) On autorise une description ambiguë de l'objet à classifier. La classification doit alors rendre compte des différentes interprétations possibles d'un attribut. Ces différentes interprétations peuvent mener à des alternatives de classification pour les objets classifiés, qui pourront être choisis selon l'approche (1).

1. On appelera attribut univoque un attribut qui ne possède qu'une seule définition, au sens d'une seule classe l'introduisant.

Nous laissons la liberté des deux approches. La prise en compte de l'homonymie est décrite en 3.7, où l'on donne les moyens de désignation non-ambiguë et de traitement de l'homonymie par la classification. Dans un premier temps, nous allons étudier la classification dans le système de base, sous l'hypothèse de l'absence d'homonymie. Le cas de la description non-ambiguë se ramène à ce système de base.

3.2 Description libre de l'objet

Structure et valeurs

Dans les langages à objets fondés sur l'instanciation, la structure d'un objet est celle définie par sa classe. Il possède les attributs de la classe, qui peuvent être valués dans le respect des contraintes imposées par celle-ci. La seule description possible est donc celle qui consiste à instancier l'objet dans une classe et valuer les attributs contenus dans sa structure. La classification est possible en discriminant sur ces valeurs (cf. les solutions primitives proposées en I 3.7). Ce procédé montre rapidement ses limites, lorsque l'on tente de classifier l'objet dans une classe qui définit des attributs qui ne font pas partie de sa structure initiale. Le processus d'appariement a besoin d'informations sur ces attributs pour pouvoir décider de l'appartenance de l'objet à la classe.

Nous distinguons deux types d'informations concernant ces attributs :

(1) la donnée d'une valeur
(2) la déclaration de la pertinence de l'attribut

La pertinence de l'attribut concerne se traduit soit par l'absence explicite de l'attribut chez l'objet que l'on classifie (l'attribut n'a pas de sens pour lui), soit par sa présence explicite (l'objet possède cet attribut, mais la valeur peut être inconnue). Les informations concernant la pertinence d'un attribut rendent possible l'exclusion de certaines classes, ou l'utilisation de moyens de calcul, dès lors que la présence d'un attribut est affirmée.

La plupart des outils de l'approche objet ne fondent l'appariement que sur la valeur (éventuellement indéterminée) des attributs. Selon la description par instanciation, les valeurs manquantes sont calculées ou demandées à l'utilisateur. La description initiale, incomplète, est complétée au fur et à mesure. Cette approche, interactive, est contraignante pour l'utilisateur, dans la mesure où celui-ci est guidé selon un parcours préétabli hiérarchique. Il est parfois interrogé sur la valeur d'attributs qui ne sont pas pertinents, sans pouvoir toujours donner toutes les informations dont il dispose.

Le procédé alternatif que nous proposons est la description libre de l'objet, qui consiste à donner une ou des classes d'appartenance initiales, et à énoncer des informations supplémentaires, sous la forme de valeurs d'attributs et de déclarations de présence ou absence d'attributs, du type "l'objet ne possède pas l'attribut x" ou "l'objet possède l'attribut x".

La classification fondée à la fois sur la structure et sur les valeurs passe nécessairement par la remise en cause de l'instanciation comme seul moyen de description des objets. La description libre s'impose pour pouvoir exprimer toutes les informations sur l'objet que l'on veut classifier, et qui, par essence, ne peuvent être prises en compte par les classes mêmes auxquelles on cherche à établir son appartenance.

L'expression et la manipulation de ces informations sur la structure relèvent d'un niveau méta. En effet, les attributs pouvant faire partie d'une description d'objet — le "vocabulaire" disponible — sont définis dans les classes de la hiérarchie. Nous allons exploiter les moyens d'ordre méta adéquats pour pouvoir obtenir et manipuler ce vocabulaire. Pour la suite de notre propos, nous considérons que la description de l'objet est complète et disponible. La réalisation de la description libre en termes de structure et de valeurs, qui sert d'hypothèse de base à la classification en FROME, est exposée en 3.8.

Description d'objet

Soit C l'ensemble de toutes les classes et S l'ensemble de tous les slots décrits dans les classes de C ; $S = \cup_{c \in C} S_c$ avec S_c l'ensemble des slots décrits[1] dans la classe c. S constitue "le vocabulaire" de description des objets. Rappelons que nous considérons ici le cas où tous les slots sont univoques : leur seul nom constitue leur identité. C'est aussi le cas dans tous les systèmes qui ne gèrent pas l'homonymie, comme notamment ceux de l'approche terminologique, qui définissent les rôles globalement en dehors des concepts. Cette approche n'exclut pas les affinements (éventuellement multiples) d'un même slot, à la manière décrite en I 3.3.3.

Appelons O l'ensemble des objets (instances terminales). Nous donnons la description d'un objet $o \in O$ en termes de quatre ensembles :

- $\mathcal{REP}_o$, l'ensemble de ses classes d'appartenance[2]
- $\mathcal{E}_o$, l'ensemble des slots qu'il possède, ou slots *existants* (pertinents) pour o
- $\mathcal{A}_o$, l'ensemble des slots qu'il ne possède pas, ou slots *absents* (non-pertinents) pour o
- $\mathcal{V}_o$, l'ensemble des slots *valués* pour o.

Nous avons :

- $\forall o \in O$:
  - $\mathcal{REP}_o \subseteq C$
  - $\mathcal{V}_o \subseteq \mathcal{E}_o \subseteq S$
  - $\mathcal{A}_o \subseteq S$
  - $\mathcal{A}_o \cap \mathcal{E}_o = \varnothing$
  - $\forall c \in \mathcal{REP}_o : S_c \subseteq \mathcal{E}_o$.

Nous définissons également :

- $\mathcal{U}_o = S \setminus (\mathcal{E}_o \cup \mathcal{A}_o)$, l'ensemble des slots non évoqués dans la description de o.

1. La description d'un slot est composée de l'ensemble de ses facettes, introduites dans la classe où héritées. L'ensemble des slots décrits dans une classe contient les slots décrits dans ses superclasses.
2. ou classes de représentation [Car89].

3.3 La classe comme description de concept

L'interprétation des classes comme des descriptions de concepts considère les attributs comme des conditions nécessaires (CN) mais non suffisantes d'appartenance. Cette interprétation consiste à considérer les classes comme des concepts *primitifs*, au sens des langages terminologiques (cf. 2.1.2.3). C'est aussi l'interprétation habituelle mais implicite des classes en RPO. [Fik&85] soulève la différence entre les concepts *définis* (conditions nécessaires et suffisantes d'appartenance) de KL-ONE et KRYPTON et les classes (prototypes) des langages comme KEE, qui ne contiennent que des conditions nécessaires. Dans l'approche terminologique, la distinction entre concepts définis et concepts primitifs est fondamentale. Les concepts définis rendent possible la classification automatique de concepts, alors que les primitifs la bloquent.

Les deux acceptions sont parfois confondues et des classes qui ne sont que des descriptions sont utilisées comme des définitions de concepts, afin de pouvoir classifier des instances. L'ensemble de leurs CN est abusivement pris pour des CNS, et l'appariement jugé "sûr" dès lors que celles-ci sont satisfaites.
L'appariement des classes interprétées comme descriptions porte sur les CN. En cas d'échec de l'appariement, l'instance est incompatible avec la classe et ne peut y appartenir. En cas de succès, l'instance est compatible avec la description de la classe, mais le rattachement effectif dépend d'un acte volontaire (de l'utilisateur). Nous retrouvons ici la distinction entre classes **possibles** (compatibles) et classes **impossibles** (incompatibles).

Fonction d'extension $\mathcal{E}$

Nous définissons une fonction d'extension sur l'ensemble des classes :

- $\mathcal{E} : C \rightarrow 2^O$
- $\mathcal{E}[c] = \{o \in O \mid c \in \mathcal{RTP}_o\}$ (c est une classe sûre pour o).

Fonction d'extension possible $\mathcal{P}$

Nous définissons également une fonction d'extension possible sur l'ensemble des classes :

- $\mathcal{P} : C \rightarrow 2^O$
- $\mathcal{P}[c] = \{o \in O \mid c$ est une classe possible pour $o\}$.

Fonction d'extension impossible I

Nous définissons également une fonction d'extension impossible sur l'ensemble des classes :

- $I : C \rightarrow 2^O$
- $I[c] = \{o \in O \mid c$ est une classe impossible pour $o\}$.

Nous avons les relations suivantes :

$\forall c \in C : I[c] \cap \mathcal{P}[c] = \varnothing$ et $\mathcal{E}[c] \subseteq \mathcal{P}[c]$

L'appariement

La classe n'étant pas une définition, l'appariement ne peut donner que le résultat "possible" ou "impossible", en fonction du respect ou de la violation des contraintes (conditions nécessaires).

Soit $\mathcal{X}[c]$ l'ensemble des classes en exclusion mutuelle avec c.

Soit v_o la fonction de valuation de slots, qui associe à un slot valué de $o \in O$ sa valeur dans l'ensemble des valeurs de slots $\mathcal{VAL}$; $v_o : \mathcal{V}_o \rightarrow \mathcal{VAL}$.

Alors l'appariement pour une classe descriptive est spécifié comme suit.

Soient c une classe descriptive et o une instance
Si $\forall \chi \in \mathcal{X}[c] : o \notin \mathcal{I}[\chi]$
et $\mathcal{S}_c \cap \mathcal{A}_o = \emptyset$
et $\forall s \in (\mathcal{V}_o \cap \mathcal{S}_c) : v_o(s)$ est valide pour c
alors $o \in \mathcal{P}[c]$
sinon $o \in \mathcal{I}[c]$

Fig. 74 Appariement avec une classe descriptive

L'objet ne doit pas être représentant d'une classe exclue par la classe examinée. Aucun slot de la classe ne doit être déclaré non-pertinent pour l'objet. Tous les slots valués doivent avoir des valeurs valides pour la classe. La valeur d'un slot est valide pour une classe si elle respecte toutes les contraintes spécifiées dans la description du slot dans cette classe. Si l'appariement réussit, la classe est possible pour l'objet, sinon elle est impossible.

Classification

La classification est définie comme l'appariement de l'objet avec une classe, propagée en cas de succès vers les sous-classes (voir algorithme Fig. 75).

La classification sur des classes descriptives est **non-déterministe**, en ce sens que les différentes classes possibles peuvent constituer des alternatives : elles peuvent éventuellement s'exclure mutuellement. Elle n'est pas opportuniste : l'appartenance sûre n'est pas établie et le succès de l'appariement n'autorise donc pas le rattachement automatique aux classes, puisqu'elles ne représentent pas des définitions de concepts.

```
classifier (C, O)
        Si O ∈ ℰ[C] ou O ∈ 𝒫[C] alors succès
        sinon
           apparier O avec C
           si l'appariement réussit alors
              succès (O ∈ 𝒫[C])
           (sinon O ∈ ℐ[C])
           fsi
        fsi
        si succès alors
           pour toutes les sousclasses C' de C
              classifier (C', O)
        fsi
```

Fig. 75 Classification non-déterministe

On peut optimiser cet algorithme en testant si toutes les super-classes sont des classes d'appartenance ou des classes possibles pour O avant d'effectuer l'appariement. Cette optimisation évite de faire l'appariement sur les sous-classes des classes impossibles (qui sont également impossibles).

Exemple

Suivons le déroulement d'une classification pour l'identification d'un "machin" (cf. Fig. 76). Toutes les classes sont des classes descriptives.

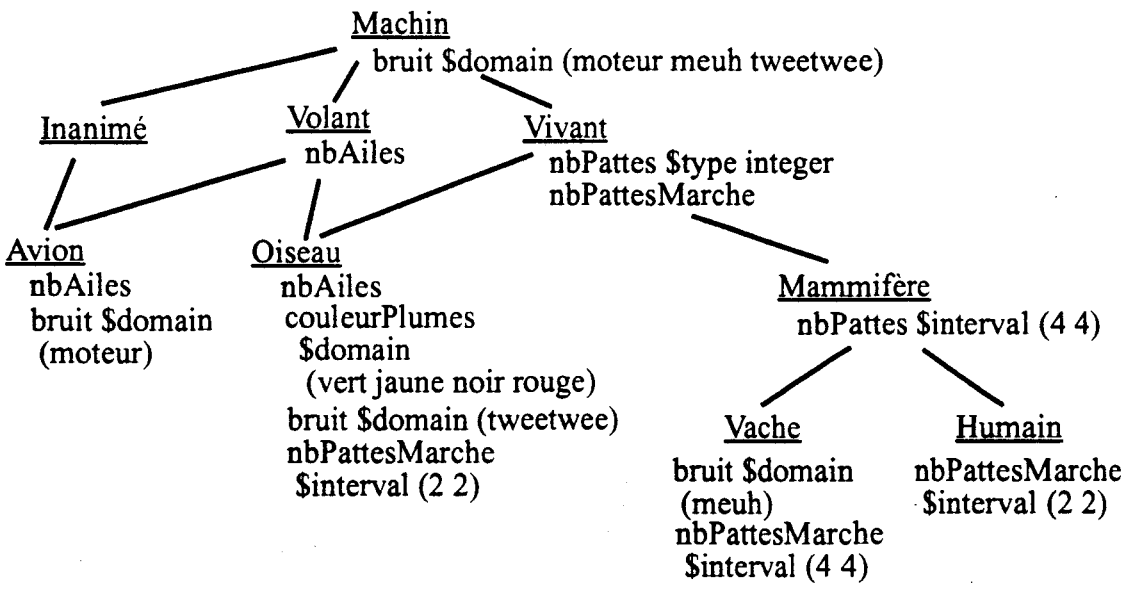


Fig. 76 Hiérarchie de machins

Description

La description d'un objet à classifier est fondée sur le choix d'une classe minimale de départ, à laquelle se conforme la structure initiale de l'objet. La description est complétée en spécifiant des valeurs pour des slots (ce qui indique implicitement la présence de ces attributs dans l'objet), ou en indiquant explicitement la présence ou l'absence de certains slots. Les slots pouvant intervenir dans cette description sont tous ceux qui sont définis (hérités, redéfinis) dans les sous-classes de la classe de départ.

Décrivons l'objet que nous voulons identifier :

- O est un Machin --- spécification de la classe initiale : Machin $\in \mathcal{REP}_O$
- O ne possède pas couleurPlumes --- non-pertinence du slot couleurPlumes : couleurPlumes $\in \mathcal{A}_O$
- O fait un bruit de moteur --- affectation du slot bruit : bruit $\in \mathcal{V}_O$

Classification

La classification est opérée successivement sur toutes les sous-classes de la classe Machin (selon l'algorithme de la Fig. 75, version non-optimisée). L'appariement de O avec la classe Inanimé ne pose pas de problème : elle est jugée possible. Sa seule sous-classe Avion est possible, car le seul slot valué est le slot `bruit`, qui respecte les contraintes. Volant est également possible. Ensuite :

- Oiseau : impossible avec slot `bruit` non valide et slot `couleurPlumes` déclaré non-pertinent
- Vivant : possible
- Mammifère : possible
- Vache : impossible avec slot `bruit` non valide
- Humain : possible (on sait bien que l'humain imite tous les bruits)

Résultat et interprétation

La description relativement imprécise de notre objet a donné lieu à plusieurs classifications possibles. Ainsi, il peut s'agir d'un Avion aussi bien que d'un Humain (faisant un bruit de moteur).

Si on étend la description initiale pour avoir plus de précision à l'issue de la classification, par exemple en explicitant que :

- O ne possède pas nbPattes --- nbPattes $\in \mathcal{A}_O$
- O possède nbAiles --- nbAiles $\in \mathcal{E}_O$

la classe Avion restera la seule classe (en feuille) possible, car la classe Vivant — et avec elle toutes ses sous-classes — deviennent impossibles.

3.4 La classe comme définition de concept

La classe joue le rôle d'une définition si la description qu'elle représente contient les conditions nécessaires et suffisantes pour appartenir à la classe. Pour une classe définie, l'appariement est fondé sur les CNS, et le succès de l'appariement implique l'appartenance certaine de l'objet à la classe. Cette propriété des classes définies est très intéressante pour la classification automatique, car elle rend possible le rattachement effectif de l'objet à la classe et elle fournit un résultat déterministe.

Dans les langages terminologiques, les conditions nécessaires et suffisantes d'un concept défini sont constituées de l'*ensemble* des formules le décrivant. Tous les descripteurs dans un concept défini sont assimilés à des contraintes. Toutes les contraintes (introductions/ restrictions de rôles ou d'attributs) sont nécessaires, et seule la satisfaction de l'ensemble des contraintes suffit pour établir l'appartenance. Si un concept possède des propriétés qui n'interviennent pas dans la décision de l'appartenance (appelées "incidental properties" dans [Bra&91]) celles-ci n'ont pas leur place dans la définition. Elles ne peuvent être exprimées qu'à l'aide de règles de chaînage avant (cf. 2.1.2).

Nous allons distinguer les attributs qui constituent des conditions nécessaires et suffisantes d'appartenance (CNS), et ceux qui peuvent être considérés comme informations déduites de l'appartenance (conditions nécessaires déduites ou CND) : un objet appartenant au concept possèdera nécessairement ces attributs. Les CND constituent un élément descriptif dans les classes définies, car elles ne sont pas déterminantes pour l'appariement. Nous exprimons les CND comme les autres attributs, au sein de la classe, car nous pensons que toutes les propriétés d'un concept doivent être regroupées dans sa description, qu'il s'agisse de propriétés déterminantes pour l'appartenance au concept ou de propriétés qui ne remplissent qu'un rôle descriptif. En particulier, tous les attributs font naturellement partie de la description des sous-classes, grâce à l'héritage, ce qui n'est pas le cas pour les propriétés accidentelles déduites à l'aide de règles, à la CLASSIC.

Les types de CNS qui interviennent dans les classes définies portent non seulement sur les valeurs de slots, mais aussi sur leur existence (présence ou absence explicitement déclarées pour un objet dans sa description). Nous distinguons les CNS respectivement selon la présence et selon la valeur :

- slots nécessairement existants ou **SNE**
- slots nécessairement valués ou **SNV**

Les CND sont constituées des :

- slots déduits ou **SD**

Pour chaque classe $c \in C$, nous avons :

- $S_c = \mathcal{SNV}_c \cup \mathcal{SNE}_c \cup \mathcal{SD}_c$
- $\mathcal{SNV}_c \subseteq \mathcal{SNE}_c \subseteq \mathcal{SD}_c$

Définissons également :

- $\mathcal{SUPD}c$: l'ensemble des super-classes directes de c
- $\mathcal{SUPG}_c$: l'ensemble des super-classes (fermeture transitive de $\mathcal{SUPD}_c$)

Appariement

L'appartenance à la classe est établie (par définition) si les CNS sont vérifiées, c. à d. :

Soient c une classe définie et o une instance
Si $\forall \chi \in \mathcal{X}[c] : o \notin \mathcal{E}[\chi]$
et $\mathcal{SNE}_c \subseteq \mathcal{E}_o$
et $\mathcal{SNV}_c \subseteq \mathcal{V}_o$
et $\mathcal{SD}_c \cap \mathcal{A}_o = \emptyset$
et $\forall s \in (\mathcal{V}_o \cap \mathcal{S}_c) : v_o(s)$ est valide pour c
alors $o \in \mathcal{E}[c]$

Fig. 77 Appariement avec une classe définie

Les CND ne doivent pas être violées, car il s'agit de conditions nécessaires déduites. Leur violation (déclaration de non-pertinence de slot ou valeur non valide) introduirait des contradictions dès l'établissement de l'appartenance.

Etant donné que tous les slots font partie de la description des sous-classes par héritage, quand nous parlons des slots SNE (resp. SNV, SD) d'une classe, il s'agit de la réunion des slots SNE (SNV, SD) hérités et de ceux définis ou redéfinis localement dans la classe. Les sous-classes héritent toutes les contraintes spécifiées dans leurs super-classes.

Comme c représente une définition de concept, quand o appartient à c ($o \in \mathcal{E}[c]$) , nous avons également :

- $\forall \chi \in \mathcal{X}[c] : o \notin \mathcal{E}[\chi]$
- $\forall sc \in \mathcal{SUPG}_c : o \in \mathcal{E}[sc]$
- $\mathcal{SNE}_c \subseteq \mathcal{E}_o$
- $\mathcal{SNV}_c \subseteq \mathcal{V}_o$
- $\mathcal{SD}_c \subseteq \mathcal{E}_o$
- $\forall s \in (\mathcal{V}_o \cap \mathcal{S}_c) : v_o(s)$ est valide pour c

Ceci implique que quand on rattache une instance à une classe définie, soit lors de sa création, soit ultérieurement, les SNV doivent soit recevoir une valeur (initialisation obligatoire), soit la valeur doit être calculable. Par ailleurs, si o appartient à c, tout moyen de calcul disponible dans la classe c (défini ou hérité) pour calculer des valeurs pour $\mathcal{SNE}_c$ ou $\mathcal{SD}_c$ peut être utilisé.

Comme la définition exprime une équivalence (CNS), le non-respect d'une seule des conditions exprimées dans l'encadré ci-dessus entraîne que $o \notin \mathcal{E}[c]$.

Par héritage des CNS et CND, si $o \notin \mathcal{E}[c]$, alors o ne peut appartenir à aucune sous-classe de c.

Classification

Comme nous avons affaire à des classes interprétées comme des définitions, pour chacune d'entre elles, la procédure d'appariement décrite ci-dessus est décisive pour l'appartenance. Un appariement réussi peut donc être couronné d'un rattachement opportuniste de l'instance à la classe. De même, l'échec d'un appariement interdit le rattachement à la classe.

La classification se définit récursivement en fonction de l'appariement d'une classe avec l'objet à classifier. En cas de succès, l'objet appartient à la classe et le processus est itéré sur les sous-classes.

```
classifier (C, O)
        Si O ∈ ℰ[C] alors succès
        sinon
           apparier O avec C
           si l'appariement réussit alors
              succès
              O ∈ ℰ[C]
           fsi
        fsi
        si succès alors
           pour toutes les sousclasses C' de C
              classifier(C', O)
        fsi
```

Fig. 78 Classification opportuniste

L'algorithme est **déterministe**, parce que opportuniste : le succès de l'appariement vaut appartenance *sûre* à la classe, et non appartenance *possible*, par opposition à l'algorithme de classification dans un environnement descriptif, où les alternatives d'appartenance possibles introduisent le non-déterminisme. Notons que le test d'appariement ne comporte pas de condition sur l'appartenance aux super-classes. En effet, comme nous travaillons avec une description complète de l'objet, et que chaque classe hérite la totalité des slots de ses superclasses, chaque classe peut tester l'appariement indépendamment des autres classes : en cas de succès, les super-classes sont également valides, car elles sont plus générales. En cas d'échec, on ne peut rien déduire sur les super-classes, mais les sous-classes sont invalidées, car elles sont plus spécifiques. La classification n'est pas propagée sur les sous-classes en cas d'échec.

La stratégie de l'algorithme adopté (Fig. 78) est celle d'un parcours en profondeur d'abord à partir de la classe racine. L'ordre des classes n'est cependant pas imposé, et toutes les optimisations connues fondées sur des changements de l'ordre de parcours (décrites pour la classification de concepts dans [Baa&92]) peuvent être appliquées. Une variante de l'algorithme, qui effectue l'appariement d'abord sur toutes les super-classes d'une classe est donnée Fig. 79.

```
classifier (C, O)
    Si O ∈ ℰ[C] alors succès
    sinon
      si ∀C' ∈SUPD_C : O ∈ ℰ[C'] alors
        apparier O avec C
        si l'appariement réussit alors
          succès
          O ∈ ℰ[C]
        fsi
      fsi
    fsi
    si succès alors
      pour toutes les sous-classes C" de C
        classifier(C", O)
    fsi
```

Fig. 79 Classification opportuniste : variante

Exemple

Prenons un exemple de classification fonctionnant sur une hiérarchie de classes interprétées comme des définitions. Nous définissons l'univers des pâtes, en conditions nécessaires et suffisantes (Fig. 80).

Description

La description de l'objet à classifier se fait de la même façon que dans l'approche fondée sur des classes descriptives.

Décrire une pâte à classer peut se faire de la façon suivante :

- O est une Pâte --- spécification de la classe initiale
  Pâte ∈ $\mathcal{REP}_O$
- O a un trou au milieu --- affectation du slot nbTrous
  nbTrous ∈ $\mathcal{V}_O$
- O a une forme cylindrique --- affectation du slot forme
  forme ∈ $\mathcal{V}_O$
- O a un diamètre de 13 mm --- affectation du slot diamètre
  diamètre ∈ $\mathcal{V}_O$

Classification

La classification selon l'algorithme de la Fig. 78 consiste à apparier successivement O avec les sous-classes de la classe Pâte. Chaque appariement réussi donne lieu à un rattachement de O à la classe, auquel cas le processus est propagé vers les sous-classes de celle-ci. O est rattaché successivement à

- Cylindrique, car le SNV `forme` est valué correctement et le SNE `nbTrous` existe et est

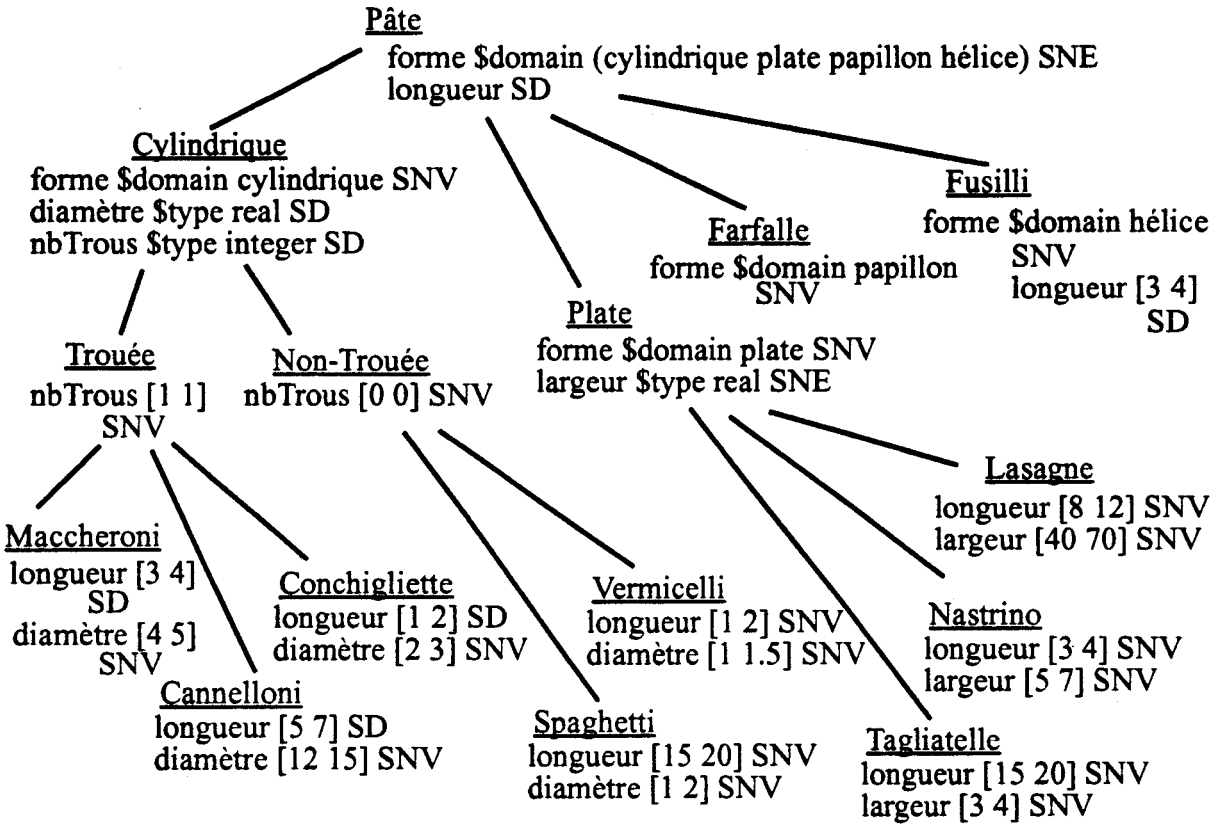


Fig. 80 Classification des pâtes

valué correctement

- Trouée, car `nbTrous` (SNV) est valué correctement
- après échec sur Maccheroni à cause de la valeur invalide pour `diamètre`, rattachement à Cannelloni, car `diamètre` (SNV) est valué correctement

Pour toutes les autres classes, l'appariement échoue. Sur cet exemple, l'optimisation de l'algorithme qui consiste à tester d'abord tous les parents n'a pas d'effet, car la hiérarchie a la forme d'un arbre (un seul parent pour chaque classe).

Notons que, pour identifier les cannellonis, la valuation de tous les slots n'est pas nécessaire. En particulier, la valuation du slot `longueur` ne fait pas partie des conditions nécessaires et suffisantes, car il s'agit d'un slot SD.

3.5 Classes définies et classes descriptives

L'avantage des classes définies est que l'appariement fournit une réponse sûre concernant l'appartenance de l'objet à une classe. Le résultat est simple et sans ambiguïté : l'objet y appartient si les CNS sont satisfaites, il n'y appartient pas sinon. L'inconvénient majeur est la contrainte de la définition au sens fort du terme : la hiérarchie entière[1] est constituée de classes dont il faut expliciter les conditions nécessaires et suffisantes d'appartenance.

Les classes descriptives ne posent pas ce problème de définition : donner des conditions nécessaires d'appartenance est peut-être contraignant — elles interdisent les exceptions — mais beaucoup moins contraignant que de définir des CNS. En contre-partie, l'appariement avec les classes descriptives ne peut donner de résultat aussi tranché que dans le cas des classes définies. Il faut se contenter de l'appartenance possible ou impossible.

Afin de se placer dans un cadre plus réaliste qui est la définition d'une hiérarchie de classes mixte, composée à la fois de classes définies et de classes descriptives, nous allons combiner les deux approches extrêmes, dont nous espérons conserver au maximum les avantages et minimiser les inconvénients. L'appariement sera suivi d'un rattachement opportuniste partout où ce sera possible (satisfaction de CNS), et fournira des classes possibles ou impossibles là où ce rattachement n'est pas permis.

La sémantique de chacune des notions (classe définie et classe descriptive) s'en trouvera légèrement modifiée, car les deux types de classes devront cohabiter dans le même environnement et coopérer à travers un protocole commun. Nous atteignons ici la distinction défini/primitif qui existe au sein des hiérarchies de concepts des langages de description de concepts.

Une première adaptation de la sémantique est celle des classes descriptives. Nous autorisons tous les types de slots introduits pour les classes définies : les SNE, les SNV et les SD. Pour les SNE et les SNV, cela revient à autoriser la définition de conditions nécessaires plus fortes pour l'appartenance à une classe descriptive. Notons que rien n'interdit la prise en compte de ces types de slots dans un environnement descriptif pur. Nous avons choisi de ne pas les introduire lors de notre étude de l'environnement descriptif pur en 3.3, car ces types de slots y représentent relativement peu d'intérêt et auraient inutilement compliqué l'exposé.

La sémantique des classes définies est modifiée en ce que celles-ci peuvent être qualifiées de **possibles** pour un objet, quand elles héritent de classes **descriptives possibles** pour cet objet.

Terminologie mixte

Définissons la terminologie mixte T = ($\mathcal{DEF}$, $\mathcal{DESC}$, O, $\mathcal{S}$), avec $\mathcal{DEF}$ l'ensemble des classes définies et $\mathcal{DESC}$ l'ensemble des classes descriptives, tels que $\mathcal{DEF} \cup \mathcal{DESC} = C$ et O l'ensemble des objets et $\mathcal{S}$ l'ensemble des slots décrits dans les classes de C.

1. Notons que (théoriquement) toutes les classes pourraient être définies entièrement en CNS, jusque la classe Object elle-même, qui peut être considérée comme une définition avec un ensemble de CNS vide : tout objet lui appartient.

Composantes primitives d'une classe définie

Nous avons vu en 2.1.2.4 que les classes descriptives peuvent être interprétées comme des définitions moyennant l'expression d'une composante primitive, qui représente exactement les conditions suffisantes (la partie indéfinie) qui manquent. Dans une terminologie mixte comportant à la fois des classes définies et descriptives, les classes définies héritent ces composantes primitives de leurs superclasses descriptives. Chaque composante primitive héritée constitue une "condition suffisante"[1] non inférable par le système. Ces composantes doivent être affirmées explicitement, par exemple en affirmant l'appartenance au concept descriptif qui les contient. L'héritage des composantes primitives a des conséquences pour l'appariement avec les classes définies.

Nous pouvons observer la validité des règles suivantes :

(1) les sous-classes d'une classe possible ne peuvent être que possibles (ou impossibles), même si elles sont définies
(2) si une classe descriptive est possible pour *o*, alors toutes ses super-classes sont également possibles pour *o*.

En effet, une classe descriptive comporte une composante primitive héritée par toutes ses sous-classes définies. Si *o* appartient à toutes les super-classes descriptives d'une classe définie, l'appariement réussi avec cette dernière entraîne l'appartenance. Par contre, si une des super-classes descriptives d'une classe définie est seulement possible pour *o* (1), l'appariement réussi avec la classe définie ne peut établir l'appartenance à celle-ci, car les conditions suffisantes pour la super-classe manquent.

Grâce à l'héritage des slots et l'héritage de l'exclusion entre classes, si aucune contrainte n'est violée par *o*, alors les contraintes des classes dont elle hérite ne sont pas violées non plus, puisqu'elles sont incluses dans celle de la sous-classe (2).

Nous observons que la qualification des classes comme possibles ou impossibles se révèle être avantageuse pour les classes définies. Elle fournit plus d'informations que le résultat catégorique (succès ou échec) obtenu dans l'approche avec classes définies seules, qui est également celle des systèmes à subsomption.

L'appariement des classes descriptives et des classes définies

Un seul appariement en deux phases est maintenant défini pour toutes les classes, aussi bien les classes définies que les classes descriptives, avec un résultat "possible" ou "impossible" à la première phase, "sûr" à la deuxième phase. La première phase est la même que l'appariement défini pour un environnement purement descriptif.

Elle reflète l'utilisation des informations "négatives" : la violation de contrainte sur valeur ou la non-pertinence d'un attribut rendent une classe impossible. S'il n'y a pas d'information négative, la classe est possible. Pour être sûre, par contre, il faut des informations "positives" : les SNE doivent tous être affirmés pertinents et les SNV doivent être valués. De plus, les éventuelles composantes primitives doivent être affirmées.

1. il s'agit bien sûr d'une condition nécessaire, mais faisant partie de l'ensemble des conditions suffisantes ; il faudrait dire *l'ensemble suffisant des conditions nécessaires*.

Soient $c \in C$ et $o \in O$
Si $\forall \chi \in \mathcal{X}[c] : o \notin \mathcal{E}[\chi]$
et $S_c \cap \mathcal{A}_o = \emptyset$
et $\forall s \in (\mathcal{V}_o \cap S_c) : v_o(s)$ est valide pour c
alors $o \in \mathcal{P}[c]$
sinon $o \in \mathcal{I}[c]$

Fig. 81 Appariement unique : première phase

Composantes primitives

Définissons pour chaque $c \in C$ l'ensemble $\mathcal{CP}_c$: l'ensemble de ses composantes primitives. La fonction d'extension $\mathcal{A}f$ sur l'ensemble des composantes primitives associe à une composante primitive l'ensemble des objets pour lesquels cette composante est affirmée.

De façon générale, nous avons maintenant :

Soient $c \in C$ et $o \in O$
Si $o \in \mathcal{P}[c]$
et $\mathcal{SNE}_c \subseteq \mathcal{E}_o$
et $\mathcal{SNV}_c \subseteq \mathcal{V}_o$
et $\forall cp \in \mathcal{CP}_c : o \in \mathcal{A}f[cp]$
alors $o \in \mathcal{E}[c]$

Fig. 82 Appariement unique : deuxième phase.
Passage de possible à sûre

Pour une classe définie sans super-classe descriptive, l'ensemble des composantes primitives est vide. L'ensemble des composantes primitives d'une classe descriptive contient au moins sa propre composante primitive.

Dans la deuxième phase, il n'y a aucune condition sur les slots SD. En effet, les SD valués doivent être valides, ce qui est déjà vérifié dans la phase I. Tous les slots valués d'une classe possible sont nécessairement valides. Puisqu'une classe ne peut être sûre sans être possible, la validité des slots valués (SD, SNE et SNV) est assurée si la classe est possible. Par leur définition même (et contrairement aux SNV), les SNE et SD non valués ne forment pas d'obstacle pour l'appartenance sûre, de même que les slots SD qui appartiendraient à $\mathcal{U}_o$ (slots non mentionnés dans la description de o).

Rappelons que l'existence des slots SD non mentionnés dans la description de o est déduite de l'appartenance à la classe. Si une classe est sûre pour o, o possède donc tous les SD. Une fois établie l'appartenance sûre, les valeurs des SD (et des SNE) pourront être obtenues par les moyens de calcul (`$if-needed`, `$default`,...) définis dans la classe.

Automate d'appariement

Nous pouvons représenter (de façon schématique) les conditions des différentes étapes de l'appariement sous forme d'un automate à trois états.

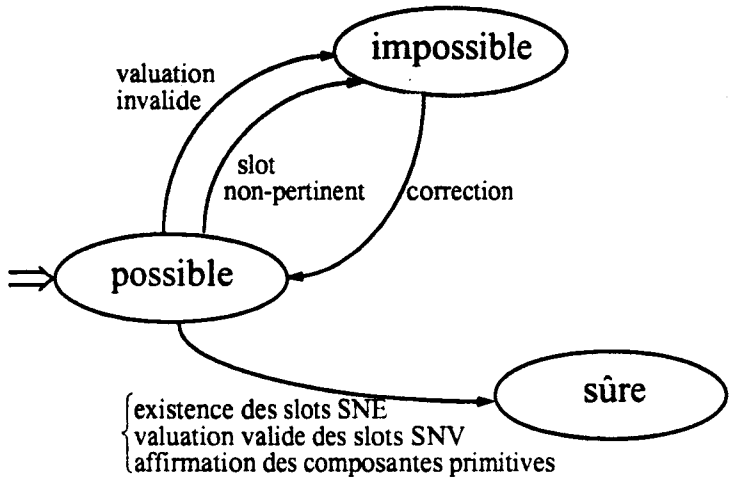


Fig. 83 Automate d'appariement : terminologie mixte

L'absence d'informations négatives mène à l'état "possible". Des informations négatives comme la valuation invalide ou la déclaration de non-pertinence de slots mènent à l'état "impossible", dont on peut sortir par correction des informations. Le passage de "possible" à "sûre" est conditionné par les informations positives obligatoires : déclarations d'existence de slots SNE, valuation de slots SNV et affirmation des composantes primitives.

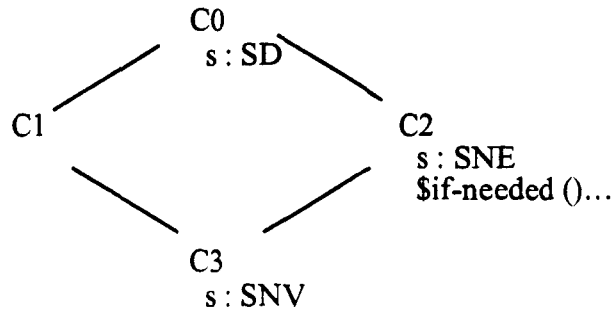
Classification

La classification (unique pour les deux types de classes) se définit maintenant en fonction des deux étapes d'appariement (cf. Fig. 84).

```
classifier (C, O)
    Si O.∈ ℰ[C] alors succès
    sinon apparier O avec C : première phase
       si l'appariement réussit alors
          succès (O ∈ 𝒫[C])
          apparier O avec C : deuxième phase
             (si l'appariement réussit alors O ∈ ℰ[C])
       (sinon O ∈ ℐ[C])
       fsi
    fsi
    si succès alors
       pour toutes les sous-classes C' de C
          classifier(C', O)
    fsi
```

Fig. 84 Classification unifiée pour classes définies et classes descriptives

Notons que suivant cet algorithme, il peut arriver que l'on teste plusieurs fois l'appariement sur des classes qui ont déjà été jugées possibles. C'est le cas quand ces classes ont plusieurs super-classes. Puisque le parcours est effectué en profondeur d'abord (grace à la récursivité sur les sous-classes), une classe peut être rencontrée plusieurs fois. Si elle est devenue une classe sûre, l'appariement n'est plus effectué (cf. premier test de l'algorithme). Par contre, une classe possible est apparié de nouveau avec l'objet. Ceci est correct, car une superclasse non encore testée peut apporter de nouvelles informations lors de son appariement qui peuvent influer sur le résultat d'une sous-classe possible et non seulement rendre cette dernière sûre, mais éventuellement impossible. Illustrons ceci avec un exemple simple :



C0, C1, C2 et C3 sont des classes définies. L'objet O, décrit comme instance de C0, possède de ce fait le slot s. L'appariement avec C1 rend cette classe sûre, l'appariement avec C3 rend celle-ci possible seulement, car s n'est pas valué. L'appariement avec C2 rend C2 sûre. Comme C2 est sûre, les moyens d'obtention définis dans C2 peuvent être invoqués. Le nouvel appariement avec C3 donne maintenant un résultat sûr si la valeur calculée pour s (à l'aide du réflexe `$if-needed` de C2) respecte les contraintes de C3, et un résultat impossible sinon.

Une variante qui privilégie l'appariement des super-classes avant celui des sous-classes élimine les appariements multiples pour classes possibles. Cette variante rend la phase I de l'appariement définitive, car une classe possible ne peut plus hériter d'informations nouvelles de super-classes devenues sûres ultérieurement. Toutes les informations des super-classes sont prises en compte directement (cf. Fig. 85).

Il s'agit de deux variantes du même processus de classification, qui donnent finalement le même résultat. La seule différence est la redondance de certains tests d'appariement.

```
classifier (C, O)
    Si O ∈ 𝓔[C] ou O ∈ 𝓟[C] alors succès
    sinon si ∀C' ∈ SUPD_C : O ∈ 𝓔[C'] ou O ∈ 𝓟[C] alors
            apparier O avec C : première phase
            si l'appariement réussit alors
               succès (O ∈ 𝓟[C])
               apparier O avec C : deuxième phase
               (si l'appariement réussit alors O ∈ 𝓔[C])
            (sinon O ∈ 𝓘[C])
            fsi
        fsi
    fsi
    si succès alors
       pour toutes les sous-classes C' de C
          classifier(C', O)
    fsi
```

Fig. 85 Classification mixte : variante

3.6 Déroulement d'une classification

Reprenons l'exemple de la classification des pâtes, en identifiant maintenant des classes définies et des classes descriptives. Les classes descriptives sont marquées avec une étoile dans la Fig. 86, comme les concepts primitifs de KL-ONE. Il y a une proportion importante de classes descriptives, ce qui aura des répercussions sur la classification : elles ne pourront être qualifiées de sûres automatiquement.

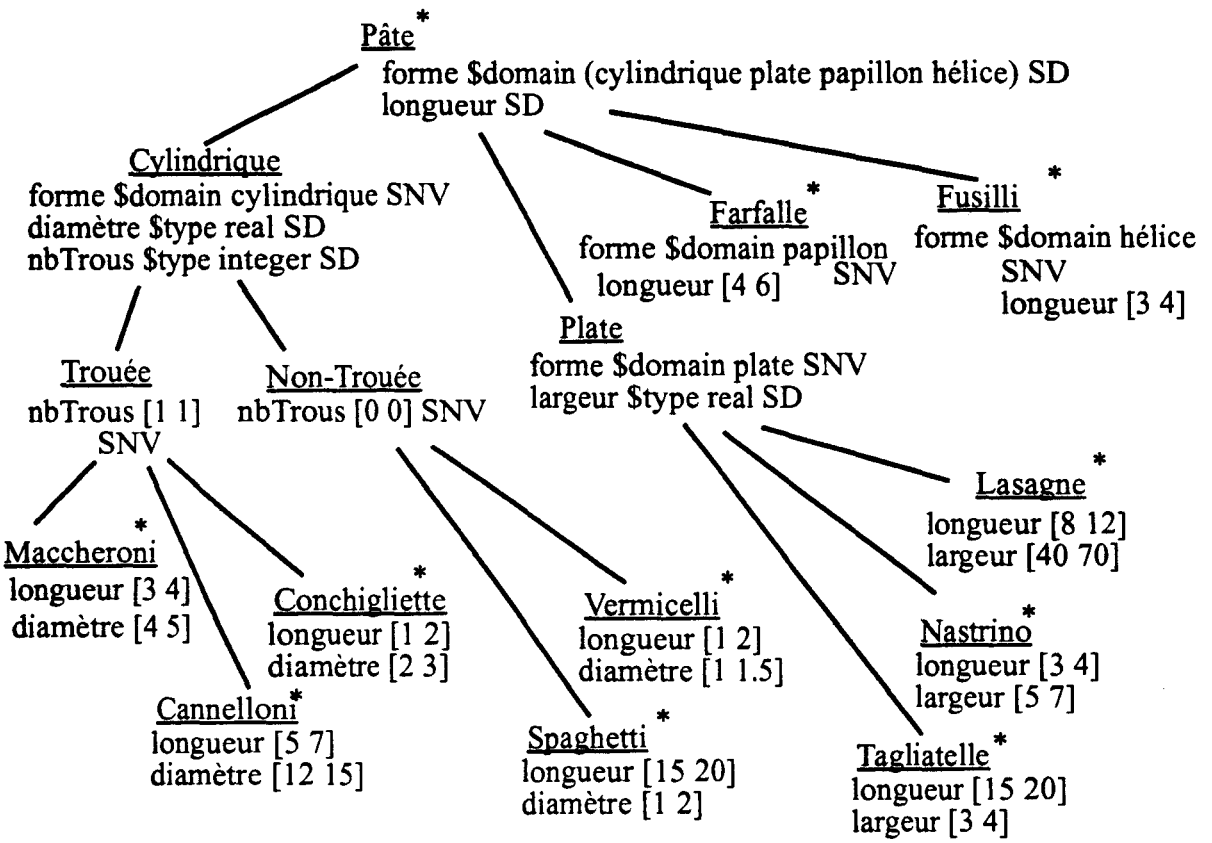


Fig. 86 Pâtes, classes définies et descriptives

Description

Décrivons une pâte à classifier :

- O est une Pâte --- Pâte $\in \mathcal{REP}_O$
- O a une longueur de 15 --- longueur $\in \mathcal{V}_O$

Classification

Après appariement selon l'algorithme (Fig. 84) on trouve :

- Classes sûres : Pâte
- Classes possibles : Cylindrique, Trouée, Non-Trouée, Spaghetti, Plate, Tagliatelle

- Classes impossibles : Maccheroni, Cannelloni, Conchigliette, Vermicelli, Nastrino, Lasagne, Farfalle, Fusilli. Ces classes sont impossibles à cause de la longueur qui est incompatible.

Une extension de la description permet d'inférer plus de choses. Déclarons par exemple :

- diamètre n'est pas pertinent pour O --- diamètre $\in \mathcal{A}_O$

Le résultat de la classification devient :

- Classes sûres : Pâte
- Classes possibles : Plate, Tagliatelle
- Classes impossibles : Cylindrique, Trouée, Maccheroni, Cannelloni, Conchigliette, Non-Trouée, Spaghetti, Vermicelli, Nastrino, Lasagne, Farfalle, Fusilli.

3.7 Homonymie d'attributs

« 'What's the use of their having names,' the Gnat said, 'if they won't answer to them?'
'No use to them,' said Alice; 'but it's useful to the people that name them, I suppose. If not, why do things have names at all?'. »

Lewis Carroll, *Through the Looking Glass.*

Jusqu'ici, nous avons considéré l'approche de base, qui exclut la présence d'attributs homonymes dans la hiérarchie des classes. Dans cette approche, deux classes indépendantes ne peuvent introduire de slots de même nom[1]. Chaque slot a un sens unique dans la hiérarchie. De ce fait, l'identité d'un slot est donnée par son nom seul.

Si on élargit l'approche de base pour prendre en compte la présence de slots homonymes, l'identité d'un slot n'est plus connue par son nom seul. En effet, chaque slot a un sens pour la classe qui l'introduit et pour ses sous-classes. Plusieurs classes peuvent introduire des slots de même nom. L'identité d'un slot est de ce fait constituée de son nom et de la classe qui l'introduit.

Dans la description du système de base, nous avons utilisé une notation abusive pour les slots, fondée sur le nom comme seul élément déterminant l'identité. Pour être plus précis et afin de pouvoir manipuler plus finement l'identité des slots dans le cadre de l'homonymie, nous définissons quelques notions supplémentaires.

Définitions

D'abord, nous distinguons deux sous-ensembles de S :

- S_u : l'ensemble des slots univoques ; un slot est univoque s'il est le seul slot de ce nom, introduit par une seule classe
- S_h : l'ensemble des slots homonymes (ou équivoques) ; un slot est homonyme si plusieurs slots de ce nom sont introduits, par des classes indépendantes

tels que :

- $S = S_u \cup S_h$ et $S_u \cap S_h = \varnothing$.

Ensuite, $\forall o \in O$, nous distinguons les slots décrits pour o (S_o) de la manière suivante :

- $\mathcal{E}_o^i$: les slots existants pour *o complètement identifiés*
- $\mathcal{E}_o^{\bar{i}}$: les slots existants pour *o non complètement identifiés*
- $\mathcal{V}_o^i$: les slots valués pour *o* complètement identifiés
- $\mathcal{V}_o^{\bar{i}}$: les slots valués pour *o* non complètement identifiés

Afin de définir la notion de slot complètement identifié, nous introduisons les fonctions suivantes :

- *nom* : la fonction qui associe à un slot son nom
- *as* : la fonction qui associe à un slot s décrit pour un objet o ($s \in S_o$), la classe désignant le

1. Une classe *introduit* un slot si elle est une classe *maximale fixant* ce slot (au sens de [Duc&92a]).

point de vue sélectionné sur ce slot (cf. expression de point de vue I 3.3.2)

- *ref* : la fonction qui associe à un slot s décrit par une classe c ($s \in S_c$), l'ensemble de référence non-ambiguë à s, définie comme suit :
  soient c1, c2, ..., cn les classes qui introduisent des slots de nom *nom*(s),
  avec $s \in S_{ci}$
  alors $ref(s) = Dep(ci) - \cup_{j \neq i} Dep(cj)$
- *id* : la fonction qui associe à un slot son identité, définie comme suit ($\forall o \in O$):
  $\forall s \in S_u, \forall s' \in S_o^i : id(s') = id(s) \Leftrightarrow nom(s) = nom(s')$
  $\forall s \in S_f, \forall s' \in S_o^i : id(s') = id(s) \Leftrightarrow nom(s) = nom(s') \wedge as(s') \in ref(s)$

L'ensemble de référence non ambiguë d'un slot contient toutes les classes qui peuvent être utilisées efficacement dans une expression de point de vue (as-expression) pour désigner ce slot sans ambiguïté. A l'inverse, toute classe faisant partie de l'intersection des dépendances de classes définissant des slots de même nom, constitue une référence ambiguë aux slots de ce nom.

Notons que la désignation non-ambiguë n'impose pas l'utilisation de la classe introduisant le slot comme seule référence possible dans l'expression de point de vue, ce qui serait contraignant pour l'utilisateur, qui ne connaît pas forcément la hiérarchie des classes en détail. L'ensemble de référence non-ambiguë est constitué de toutes les classes en relation avec celle-ci, sauf celles qui sont également en relation avec d'autres classes introduisant un slot de même nom. Il s'agit donc bien d'exprimer un point de vue au sens de ROME, qui prend en compte une partie du graphe et non une classe unique.[1]

Slots complètement identifiés

> *« 'When I use a word,' Humpty Dumpty said, in rather a scornful tone, 'it means just what I choose it to mean — neither more nor less.'*
> *'The question is,' said Alice, 'whether you can make words mean so many different things.'*
> *'The question is,' said Humpty Dumpty, 'which is to be master — that's all.'»*
>
> Lewis Carroll, *Through the Looking Glass.*

Nous avons pour chaque description d'objet :

- $\forall s \in S_o : (\exists s' \in S_u \mid nom(s) = nom(s')) \Rightarrow s \in S_o^i$
- $\forall s \in S_o : (\exists s' \in S_f \mid nom(s) = nom(s') \wedge as(s) \in ref(s')) \Rightarrow s \in S_o^i$

Autrement dit : tout slot univoque décrit pour un objet par son nom est complètement identifié et tout slot homonyme décrit pour un objet par son nom et par un point de vue non-ambigu est complètement identifié.

Comme nous ne manipulons pas de slots anonymes, tous les slots univoques évoqués dans la description d'un objet sont nécessairement complètement identifiés. Les slots homonymes

1. L'ensemble de référence non-ambiguë pouvant être calculé facilement pour tout slot défini, le système est en mesure de proposer à l'utilisateur un choix de points de vue non-ambigus possibles pour un slot homonyme de nom donné.

désignés sans référence de point de vue ou avec une référence ambiguë sont non complètement identifiés :

- $\forall s \in S_o$:
  $(\exists\, s' \in S_h \mid nom(s) = nom(s')) \wedge (\forall s'' \in S_h \mid nom(s) = nom(s'')\colon as(s) \notin ref(s''))$

  $\Rightarrow s \in \bar{S}_o^{\,i}$

Les ensembles $\mathcal{E}_o^{\,i}$ ($\bar{\mathcal{E}}_o^{\,i}$) et $\mathcal{V}_o^{\,i}$ ($\bar{\mathcal{V}}_o^{\,i}$) se définissent respectivement comme l'intersection de $S_o^{\,i}$ ($\bar{S}_o^{\,i}$) avec $\mathcal{E}_o$ et avec $\mathcal{V}_o$.

Introduisons enfin les deux fonctions suivantes :

- *valeur* : la fonction qui associe à un slot s décrit pour un objet o ($s \in S_o$), sa valeur.
- *domaine* : la fonction qui associe à un slot s décrit par une classe c ($s \in S_c$), l'ensemble des valeurs valides pour s.

Notons que dans cette nouvelle notation, nous distinguons les slots décrits par les classes et ceux décrits pour un objet. La structure de ces descriptions n'est plus la même : nous avons *nom*, *as* et *valeur* pour les descriptions de slots d'objets et *nom*, *ref* et *domaine* pour les descriptions de slots dans les classes. Leur mise en correspondance se fait à travers la fonction *id*. La description des slots d'un objet (S_o) se réfère toujours au vocabulaire disponible des slots définis dans les classes d'une hiérarchie (l'ensemble $S = \cup_{c \in C} S_c$) :

- $\forall s \in S_o : \exists\, s' \in S \mid nom(s) = nom(s')$.

L'appariement en deux phases pour une terminologie mixte, présenté en 3.5, peut s'écrire selon cette notation :

Soient $c \in C$ et $o \in O$

Phase I:
Si $\forall \chi \in \mathcal{X}[c] : o \notin \mathcal{E}[\chi]$
et $\forall s \in S_c, \forall s' \in \mathcal{A}_o : id(s) \neq id(s')$
et $\forall s \in S_c, \forall s' \in \mathcal{V}_o \mid id(s) = id(s') : valeur(s') \in domaine(s)$
alors $o \in \mathcal{P}[c]$
sinon $o \in \mathcal{I}[c]$

Phase II :
Si $o \in \mathcal{P}[c]$
et $\forall s \in \mathcal{SN}\mathcal{E}_c : \exists\, s' \in \mathcal{E}_o \mid id(s) = id(s')$
et $\forall s \in \mathcal{SN}\mathcal{V}_c : \exists\, s' \in \mathcal{V}_o \mid id(s) = id(s')$
et $\forall cp \in \mathcal{CP}_c : o \in \mathcal{A}f[cp]$
alors $o \in \mathcal{E}[c]$

Fig. 87 Appariement approche de base — notation étendue

Appariement avec prise en compte de l'homonymie d'attributs

> *« 'That's a great deal to make one word mean,' Alice said in a thoughtful tone.*
> *'When I make a word do a lot of work like that,' said Humpty Dumpty, 'I always pay it extra.'»*
>
> Lewis Carroll, *Through the Looking Glass.*

Nous allons maintenant étendre l'appariement pour rendre compte de la présence de slots homonymes. De manière générale, l'appariement cherche pour tout slot décrit dans la classe un slot correspondant dans la description de l'objet. Quand la correspondance est établie de manière sûre ($id(s) = id(s')$), la valeur éventuelle du slot devra être valide pour que la classe soit possible. Si elle est invalide, la classe est impossible. Par contre, quand il y a ambiguïté, la classe est possible, aussi bien en cas de valeur invalide qu'en cas de valeur valide pour le slot ambigu. En effet, tant que l'identité du slot est incertaine, on ne peut rien dire sur sa validité.

Ensuite, pour qu'une classe puisse être sûre, ses slots SNE et SNV doivent de plus avoir des correspondants complètement identifiés. De plus, tous les slots SD homonymes avec des valeurs invalides doivent être complètement identifiés comme étant des slots *différents* de ceux décrits dans la classe. Si un tel slot n'est pas complètement identifié, alors la classe n'est que possible. S'il y avait identité entre le slot de la classe et le slot complètement identifié, alors il y aurait violation de contrainte et la classe serait impossible (voir ci-dessus).

De la même façon que dans l'approche de base, l'absence d'information négative rend une classe possible, alors que l'appartenance sûre nécessite des informations positives. L'ambiguïté ne rend donc pas une classe impossible, mais l'identification complète des slots obligatoires (SNE et SNV) est requise pour établir l'appartenance sûre. Pour l'appartenance sûre, les SD peuvent rester ambigus quand ils n'ont pas de valeur ou quand ils ont une valeur valide, car il s'agit de slots déduits. Quand ils ont une valeur non valide, par contre, il faut s'assurer qu'ils représentent des slots différents de ceux de la classe. Dans ce cas, leur identification complète devient également obligatoire.

Nous interprétons la déclaration d'absence de slot comme une information négative forte, qui exprime le fait que l'objet ne possède aucun slot de ce nom. La seule indication du nom (sans identification complète) suffit dans ce cas pour indiquer l'absence de tous les slots qui portent ce nom (cf. phase I de l'appariement Fig. 88). En effet, nous considérons que l'on n'indique l'absence d'un slot que quand on est convaincu de la non-pertinence de ce slot. Dans le cas contraire, l'utilisateur ne ferait plutôt aucune déclaration à propos de ce nom de slot.

Une variante pourrait néanmoins prendre en compte l'identification complète des slots absents. On définirait $\mathcal{A}_o^{\vec{i}}$ et $\mathcal{A}_o^{\vec{i}}$ de manière analogue à $\mathcal{E}_o^{\vec{i}}$ et $\mathcal{E}_o^{\vec{i}}$. La première phase de l'appariement serait alors plus permissive (classe possible pour slots absents ambigus, classe impossible seulement en cas de slots absents complètement identifiés et de même identité), alors que la deuxième phase nécessiterait une identification complète des slots absents, à la manière des slots SD non valides, afin de s'assurer que leur identité soit différente de celle des slots de la classe :

- $\forall s \in (\mathcal{SD}_c \cap \mathcal{S}_h) \mid (\exists\, s' \in \mathcal{A}_o \wedge nom(s) = nom(s')) : s' \in \mathcal{A}_o^{\vec{i}} \wedge id(s) \neq id(s')$.

Ici, nous choisissons la désignation des slots absents par le nom seul. Ce choix entraîne

Soient $c \in C$ et $o \in O$

Phase I:

Si $\forall \chi \in \mathcal{X}[c] : o \notin \mathcal{E}[\chi]$
et $\forall s \in \mathcal{S}_c, \forall s' \in \mathcal{A}_o : nom(s) \neq nom(s')$
et $\forall s \in \mathcal{S}_c, \forall s' \in \mathcal{V}_o^i \mid id(s) = id(s') : valeur(s') \in domaine(s)$
alors $o \in \mathcal{P}[c]$
sinon $o \in \mathcal{I}[c]$

Phase II :

Si $o \in \mathcal{P}[c]$
et $\forall s \in \mathcal{SNE}_c : \exists\, s' \in \mathcal{E}_o^i \mid id(s) = id(s')$
et $\forall s \in \mathcal{SNV}_c : \exists\, s' \in \mathcal{V}_o^i \mid id(s) = id(s')$
et $\forall s \in (\mathcal{SD}_c \cap \mathcal{S}_h) \mid (\exists\, s' \in \mathcal{V}_o \wedge nom(s) = nom(s') \wedge$
$valeur(s') \notin domaine(s)) : s' \in \mathcal{V}_o^i \wedge id(s) \neq id(s')$
et $\forall cp \in \mathcal{CP}_c : o \in \mathcal{A}f[cp]$
alors $o \in \mathcal{E}[c]$

Fig. 88 Appariement dans une terminologie mixte avec homonymie

l'impossibilité de déclarer l'absence d'un slot en même temps que la présence d'un autre slot de même nom.

Avec la prise en compte de l'homonymie et de l'identification complète des slots, l'automate de l'appariement devient celui de la Fig. 89.

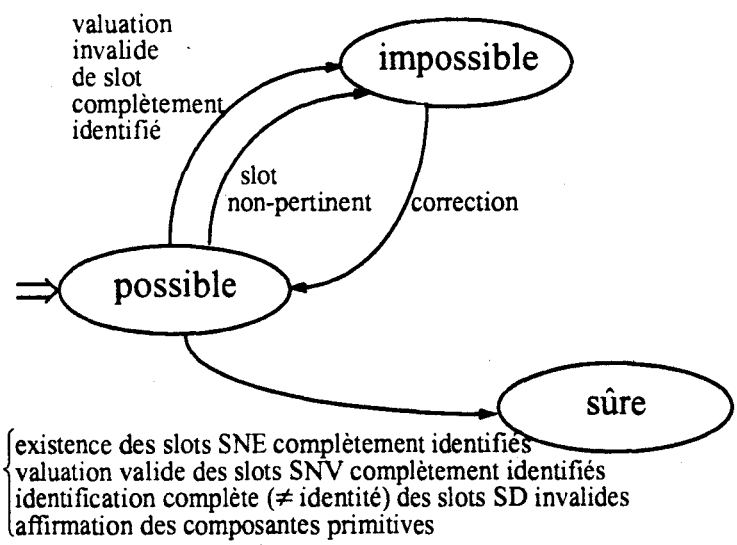


Fig. 89 Automate d'appariement : terminologie mixte avec homonymie d'attributs

Rappelons que les slots univoques sont en fait tous des slots complètement identifiés (cf. page 197). L'approche de base peut donc se résumer à l'approche avec homonymie

moyennant l'identification complète des slots. Le seul cas particulier est celui des SD homonymes valués non valides, dans la phase II de l'appariement. Ce cas ne pouvait se présenter dans l'approche de base (univoque pure), car il s'agit d'un cas où il faut vérifier la non-identité, tout en ayant l'égalité des noms.

Exemple avec homonymie (1)

Pour illustrer la classification avec prise en compte de l'homonymie, dans le contexte de l'univers mixte des classes définies et des classes descriptives, prenons un exemple abstrait :

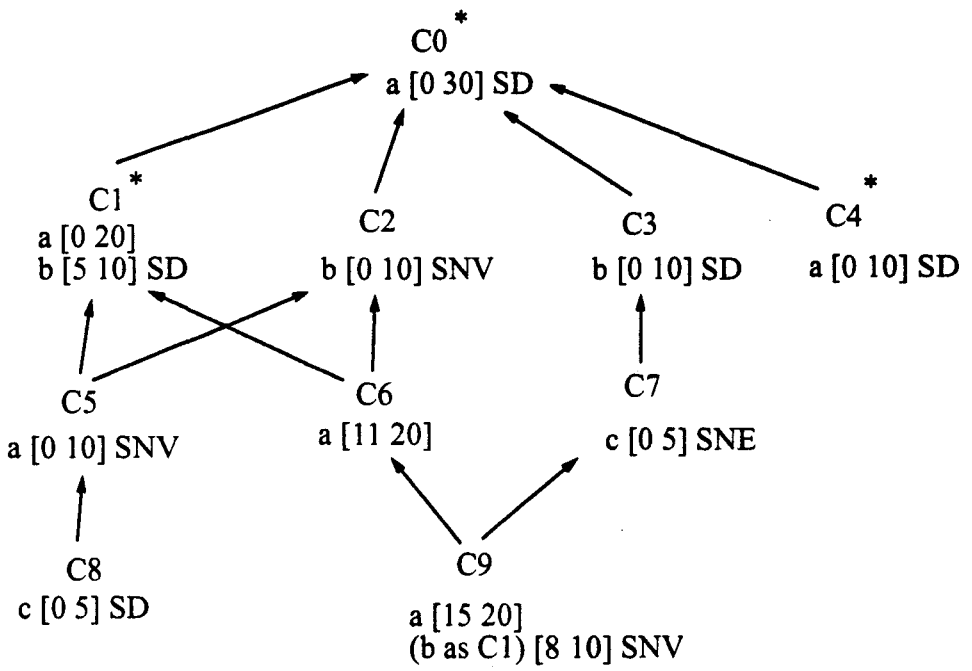


Fig. 90 Hiérarchie avec slots homonymes

Nous avons 3 classes descriptives : C0, C1 et C4. Pour simplifier la présentation, tous les slots sont de type entier, avec des restrictions d'intervalle. Il y a un seul slot univoque a, affiné de façon multiple dans C1, C5, C6, C9 et C4. Les autres slots sont équivoques :

- les homonymes b : introduits par C1, C2, C3
- les homonymes c : introduits par C7, C8

Description

Décrivons l'objet à classifier :

- O est un C0 --- C0 $\in \mathcal{REP}_O$
- O a un b de 6 --- b $\in \mathcal{V}_O^i$
- O a un (c as C9) de 6 --- c $\in \mathcal{V}_O^i$
- O a un a de 15 --- a $\in \mathcal{V}_O^i$

Classification

Après appariement selon l'algorithme (Fig. 88) on trouve :

- Classes sûres : C0, C3 (car b de C3 est un SD)
- Classes possibles : C1, C2, C6
- Classes impossibles : C5, C8 et C4 à cause du slot a incompatible, C9 et C7 à cause du slot c incompatible

Extension de la description :

- O a un (b as C6) de 10 --- $b \in \mathcal{V}_O^i$

Cette extension a pour effet de rendre la classe définie C2 sûre, car son SNV b est bien identifié et valué correctement. Si de plus nous ajoutons :

- O est un C1 --- $O \in \mathcal{A}f[\overline{C1}]$ avec $\overline{C1}$ la composante primitive de C1

alors la classe définie C6 devient sûre également.

Exemple avec homonymie (2)

Enfin, prenons un exemple plus concret (Fig. 91). La classe Personne est une classe descriptive. Nous pouvons définir deux sous-classes : Salarié et Etudiant. Est Salarié toute Personne ayant un salaire ; est Etudiant toute Personne inscrite dans une Université. Les slots `salaire` et `inscription` sont donc SNE, car la valeur de ces slots n'est pas nécessaire pour décider de l'appartenance aux deux classes. Chômeur est une classe descriptive, en exclusion avec Salarié et Etudiant. Une sous-classe définie de la classe Etudiant peut être Boursier, avec le slot `bourse` : SNE. La sous-classe Salarié-au-Smic de Salarié est défini selon la valeur du salaire : le slot `salaire` devient donc SNV.

Il y a homonymie entre les deux slots `nbheures` de Salarié et `nbheures` de Etudiant. De plus, ces slots sont affinés tous les deux dans la sous-classe Salarié-FC, qui représente les salariés en formation continue. Cette classe est définie en fonction des affinements du nombre d'heures de travail et du nombre d'heures d'études. Les deux slots `nbheures` deviennent SNV et leurs domaines sont restreints.

Notons que la valeur du slot `salaire` de Salarié pourra être calculée par un réflexe `$if-needed`, par exemple selon la `fonction`. Les réflexes peuvent être exploités dès que la classe est sûre pour un objet, i.e. quand l'objet est représentant de la classe.

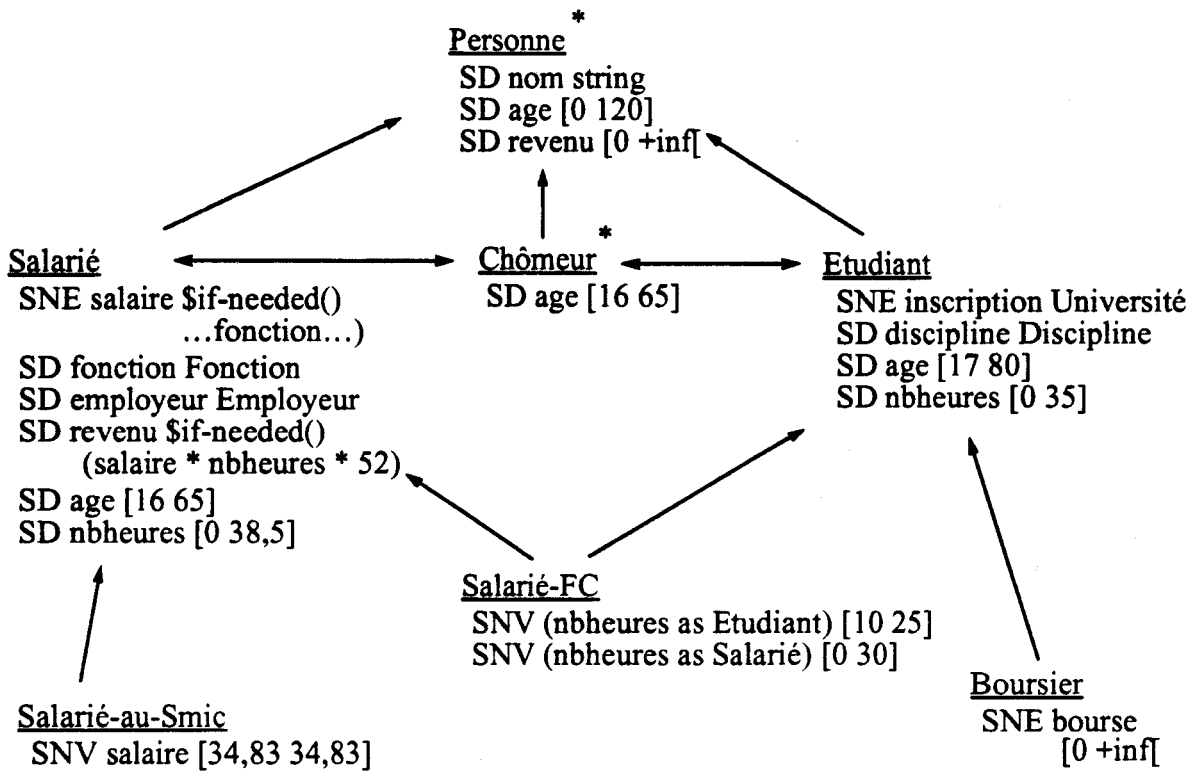


Fig. 91 Classes définies, classes descriptives, homonymie, exclusion

Description 1

Décrivons l'objet à classifier :

- P1 est une Personne --- Personne $\in \mathcal{REP}_{P1}$
- P1 est agé de 30 ans --- age $\in \mathcal{V}_{P1}{}^{i}$
- P1 n'a pas d'employeur --- employeur $\in \mathcal{A}_{P1}$

Classification

Après appariement on trouve :

- Classes sûres : Personne
- Classes possibles : Chômeur, Etudiant, Boursier
- Classes impossibles : Salarié, à cause de l'absence d'employeur, et toutes ses sous-classes.

Description 2

- P2 est une Personne --- Personne $\in \mathcal{REP}_{P2}$
- P2 est agé de 26 ans --- age $\in \mathcal{V}_{P2}{}^{i}$
- P2 a un salaire --- salaire $\in \mathcal{L}_{P2}{}^{i}$
- P2 a un nombre d'heures de 30 --- nbheures $\in \mathcal{V}_{P2}{}^{i}$

Classification

Après appariement on trouve :

- Classes sûres : Personne, Salarié
- Classes possibles : Salarié-au-Smic, Salarié-FC, Etudiant, Boursier
- Classes impossibles : Chômeur, à cause de l'exclusion avec Salarié

Notons que grâce à l'ambiguïté sur le slot `nbheures`, la classe Salarié-FC est une classe possible. La classe Salarié-au-Smic pourra être sûre ou impossible en fonction de la valeur calculée par le réflexe associé au slot `salaire` dans Salarié.

Description 3

- P3 est une Personne --- Personne $\in \mathcal{REP}_{P3}$
- P3 est agé de 26 ans --- age $\in \mathcal{V}_{P3}{}^{i}$
- P3 a un salaire --- salaire $\in \mathcal{E}_{P3}{}^{i}$
- P3 a un nombre d'heures en tant qu'étudiant de 30 --- nbheures $\in \mathcal{V}_{P3}{}^{i}$

Classification

Après appariement on trouve :

- Classes sûres : Personne, Salarié
- Classes possibles : Salarié-au-Smic, Etudiant, Boursier
- Classes impossibles : Chômeur, Salarié-FC, à cause de l'incompatibilité du slot (`nbheures as Etudiant`).

La désignation non-ambiguë d'un slot permet le passage de possible à impossible ou sûre (abstraction faite des autres conditions d'appartenance).

3.8 Eléments d'implantation

Nous discutons les éléments qui permettent la réalisation de la classification selon l'approche mixte et nous esquissons les grandes lignes de sa conception.

Nous retenons de l'approche objet le contrôle distribué, programmé comme le comportement spécifique des objets impliqués. Ainsi, l'appariement est défini comme un comportement des classes et fait intervenir l'appariement des slots, défini à son tour comme un comportement des classes de slots. Les types de slots que nous avons distingués selon leur rôle dans l'appariement correspondent à des classes spécifiques avec des comportement d'appariement propres.

Le "vocabulaire" des slots disponibles pour la description des objets est détenu par les classes. La manipulation des descriptions associées a donc un caractère méta.

La description libre des objets nécessite la manipulation des caractéristiques des objets en dehors des classes. Nous proposons un objet particulier avec un rôle méta pour gérer cette description.

3.8.1 Description libre des objets

Les objets sont décrits à l'aide des slots du vocabulaire. L'accès à ce vocabulaire passe par une interrogation des classes, qui détiennent la description des slots qu'elles définissent : leur nom, leur type et autres contraintes sur la valeur. Cette interrogation permet également de déduire quels slots sont univoques, quels slots homonymes. L'accès à la structure même qui est détenue par les classes est possible grâce aux facilités d'ordre méta offertes par le langage. On a effectivement accès au niveau méta, car les classes elles-mêmes sont des objets, avec un comportement et avec des attributs qui contiennent la description de la structure de leurs instances.

C'est grâce à cette facilité de manipulation de la structure que l'on peut proposer un choix de slots à l'utilisateur, avec toutes les informations de type, de domaine, et même de point de vue potentiel associés[1].

3.8.1.1 L'Ombre : un méta-objet pour la classification

La description des objets peut être considérée elle-même comme relevant d'un méta-niveau, car elle contient des méta-informations sur l'objet, telles l'existence de certains attributs, leurs (domaines de) valeurs. Dans l'approche classique classe-instance, les informations d'ordre méta se trouvent toujours dans la classe de l'objet. Ces méta-informations concernent indistinctement *toutes* les instances de la classe. Il n'y a pas de place pour des méta-informations qui seraient spécifiques à une seule instance. Les objets n'ont pas de méta-objet privé (comme il en existe en 3-KRS, système basé sur le modèle prototype-spécialisation [Mae87a]) ; leur méta-objet est la classe, partagée par toutes ses instances. Ferber [Fer89] semble être le seul à avoir introduit un méta-objet "privé", en plus de

1. Nous envisageons la définition d'une interface appropriée pour faciliter la saisie de la description. La description consisterait alors à sélectionner les slots disponibles (ceux des sous-classes de la classe initiale) et à les valuer ou déclarer leur absence ou présence. L'importance des slots SNE et SNV pourrait être visualisée pour aider l'utilisateur.

la classe dans le modèle classe-instance. Ce méta-objet est chargé de la réflexivité opératoire (computationnelle) et s'occupe du comportement de l'objet, alors que les classes ne décrivent que la structure (réflexivité structurale) de leurs instances (cf. aussi 2.5).

Nous reprenons de Ferber l'idée d'associer un objet privilégié à un objet, destiné à gérer la description de celui-ci (ses méta-informations "privées") durant sa classification. Nous appelons cet objet son *Ombre*. Notons que le méta-objet de Ferber est dédié à la réification du processus de communication (gestion des envois de messages), alors que l'Ombre est destiné à la gestion de la description de son objet. Le seul point commun est donc le caractère dédié à un objet unique.

Ombre

Nous proposons une définition simplifiée de l'objet ombre associé à un frame, donnée par la classe `shadow` ci-dessous.

```
[meta-i-frame 'new 'shadow 'supercl 'rframe
 'slots '(dico ($type dictionary $supercl rslot)
          object ($type shadowed-rframe))
 'methods '(init (lambda()
                  {<<- 'dico
                    [[dictionary 'new (genobj 'dico)] 'init]}
                  self)
          destruc-fobj (lambda()
                  [{?? 'dico} 'destruction] (super))
          put (lambda(a v) [{?? 'dico} 'put a v])
          get (lambda(a) (or [{?? 'dico} 'get a]
                  {rome-error {?? 'object}
                   " does not have a slot " a}))
          has (lambda(s) [{?? 'dico} 'has s])
          has? (lambda(s)[{?? 'dico} 'has? s])
          doesnothave (lambda(s)[{?? 'dico} 'doesnothave s])
          doesnothave? (lambda(s)[{?? 'dico} 'doesnothave? s])
 'selectors '(put put get get init init has has has? has? doesnothave
          doesnothave doesnothave? doesnothave?)]
```

Une ombre a deux slots :

- `dico`, contenant un objet de type `dictionary`
- `object`, référençant le frame pourvu d'une ombre (de type `shadowed-rframe`, introduit dans 3.8.1.3).

La classe `dictionary` est définie par ailleurs et comporte des définitions pour les méthodes associées aux sélecteurs `get` et `put`, pour la lecture et l'affectation des slots gérés par l'ombre. Un objet dictionnaire est créé lors de l'initialisation de l'ombre et est détruit avec elle. Les accès aux slots gérés par l'ombre (méthodes `put` et `get`), ainsi que la déclaration et la demande de la présence ou de l'absence de slots sont traduits par des envois de message au dictionnaire de l'ombre. Nous ne détaillons pas l'implantation de la gestion de ces données par le dictionnaire.

3.8.1.2 Expression et gestion de la description

La description libre de l'objet est cautionnée par l'ombre qui lui est associée et qui prend en

charge les informations concernant la structure et les valeurs de l'objet durant la classification. L'ombre gère entièrement les attributs décrits tant que l'objet n'est pas rattaché à une classe qui les prend en compte. Ces attributs restent en arrière-plan, "dans l'ombre". Leurs lecture et écriture sont prises en charge par l'ombre à travers l'objet, de manière transparente.

La description de l'objet est implantée comme son instanciation dans la classe initiale et par des envois de messages complétant sa description. Ces messages sont soit des affectations de slots :

```
[<frame> '<- <slot> <value>] ou [<frame> '<- '(<slot> as <classe>) <value>]
```

par exemple :

```
[O '<- 'bruit "moteur"]
```

soit des affirmations concernant l'existence ou l'absence d'un slot dans le frame :

```
[<frame> 'has <slot>] ou [<frame> 'has '(<slot> as <classe>)]

[<frame> 'doesnothave <slot>]
```

par exemple :

```
[O 'has 'nbAiles]

[O 'doesnothave 'nbPattes]
```

Ces informations sont redirigées vers l'ombre, qui en garde la trace dans son dictionnaire : les slots absents, slots existants, avec leur valeur éventuelle et le point de vue exprimé.

Hypothèse d'accès à la description

L'accès aux slots dans l'ombre repose sur l'hypothèse d'absence de déclenchement de réflexes.

C'est un choix que nous faisons sur la description des slots dans l'ombre. Tant que l'objet n'est pas rattaché à ses classes d'appartenance, les réflexes associés aux slots dans les classes ne lui appartiennent pas. En effet, rien ne permet de choisir une définition de réflexe à appliquer, quand celui-ci possède un ou plusieurs affinements. Naturellement, l'appartenance sûre autorise l'exécution des réflexes définis dans les classes d'appartenance de l'objet en cours de classification, en particulier des réflexes de calcul de valeurs, dès le rattachement de l'objet.

3.8.1.3 Les frames "ombragés"

En I 4.3.4 nous avons décrit les mixins pour la représentation, qui permettent d'étendre la description d'une instance avec un comportement particulier, qui peut être temporaire. L'extension obtenue par mixin ne relève pas d'un niveau de représentation conceptuelle, comme la description contenue dans les autres classes d'un objet. Nous définissons le comportement et les caractéristiques d'un objet en cours de classification comme une extension particulière de son comportement et de ses caractéristiques habituels. Pour prendre en compte cette extension, nous introduisons la classe "shadowed-frame", définie comme une classe mixin pour la représentation multiple. Elle attribue (temporairement) une ombre à n'importe quel frame. Voici une définition minimale de la mixin `shadowed-rframe` :

```
[mixin-frameclass 'new 'shadowed-rframe 'supercl 'rframe
 'slots '(shadow ($type shadow $supercl rslot))
 'methods '(after-mix+ (lambda(c)
                         {(init as shadowed-rframe)})
            before-mix- (lambda(c)
                          [{?? 'shadow} 'destruction])
            init (lambda()
                   {<<- 'shadow
                    [[shadow 'new (genobj 'shadow) 'object self] 'init]}
                   self)
            allatts (lambda() (let ((at))
                      (mapc (lambda(l)
                              (setq at (append (:l-att [l 'attributesg]) at)))
                            {? 'rclasses}) at)))
            i-have-attribute (lambda(att-name)
                               (memq att-name {allatts}))
            has (lambda(s) (ifn {i-have-attribute s}
                                [{?? 'shadow} 'has s] ()))
            has? (lambda(s) (cond ({i-have-attribute s} t)
                                  (t [{?? 'shadow} 'has? s] )))
            doesnothave (lambda(s)
                          (cond ({i-have-attribute s} {rome-error …})
                                (t [{?? 'shadow} 'doesnothave s] )))
            doesnothave? (lambda(s)
                           (cond ({i-have-attribute s} ())
                                 (t [{?? 'shadow} 'doesnothave? s] )))
            ?? (lambda(a) (if {i-have-attribute a}
                              (super)
                              [(applymeth '?? 'shadow) 'get a]))
            <<- (lambda(a v) (if (applymeth 'i-have-attribute a)
                                 (super)
                                 [(applymeth '?? 'shadow) 'put a v])))
 'selectors '(init init allatts allatts)]
```

N'importe quel frame (représentant de la classe `rframe+mix` cf. I 4.3.4) est doté d'une ombre par l'envoi du message :

```
[<frame> 'mix+ 'shadowed-rframe]
```

La classe `shadowed-rframe` redéfinit les méthodes d'accès aux attributs, de façon à tester si le frame possède l'attribut (la méthode `i-have-attribute` est définie ici comme l'interrogation de toutes les classes de représentation directes du frame pour collecter la liste des attributs définis[1]), auquel cas la super-méthode est appliquée. Dans le cas contraire, l'accès est délégué vers l'ombre, qui gère les attributs "hors classe". Quant à la communication avec le frame, la gestion des accès aux slots par son ombre se fait de manière entièrement transparente. Ainsi, la demande de valeurs de slots à un frame ne nécessite pas son appartenance établie à la classe définissant cet attribut. En particulier, lors de l'appariement effectué par une classe, celle-ci peut s'adresser directement au frame pour obtenir les valeurs nécessaires, ou la présence ou l'absence de slots (`has?` <slot>,

1. La méthode `i-have-attribute` a été volontairement simplifiée ici. Elle ne prend en compte que des noms d'attributs. En réalité, elle est un peu plus complexe, car elle prend en compte l'expression de points de vue sur les attributs.

`doesnothave?` <slot>). Celui-ci redirige la demande vers son ombre, s'il ne peut pas la traiter lui-même.

L'envoi de message

```
[<frame> 'mix- 'shadowed-rframe]
```

a pour effet le détachement du shadow et sa destruction (méthode `$before-mix-`).

3.8.2 Contrôle distribué de la classification

L'algorithme de classification étant défini de façon récursive sur les classes, il peut facilement être distribué selon ces mêmes classes. Chaque classe disposant des éléments décrivant la structure de ses instances, elle est capable de comparer cette structure avec la description de l'objet à classifier. La classification est donc définie pour une classe comme une méthode, qui effectue un appariement local, avant d'activer ses sous-classes par envoi de message.

La description des slots étant réifiée sous forme de classes de slots, ces dernières peuvent prendre en compte plus finement l'appariement de chaque slot de la classe avec un slot décrit pour l'objet. L'appariement local sur chaque classe est donc en partie redistribué sur ses slots, également par envoi de messages. Cette redistribution est partielle car l'appariement dépend non seulement des slots, mais aussi de l'exclusion entre classes et de l'affirmation de composantes primitives (c. à d. l'appartenance sûre à des super-classes descriptives).

Appariement pour une classe

Pour chacun des slots décrits dans la classe, il s'agit de trouver un slot correspondant dans la description de l'objet et si ce slot existe, d'effectuer son appariement avec la classe de slots. Pour chaque slot d'une classe, nous obtenons un résultat "sûr", "possible" ou "impossible". En prenant en compte cette redirection sur les classes de slots, la méthode de classification avec appariement local (celle définie formellement Fig. 84 (page 191)) pour une classe de frames (self) se réécrit comme suit :

classify (O)

Si O n'est pas déjà représentant de self alors
  Si O appartient à une classe exclue par self
    ou si un des slots de self répond impossible alors **impossible**
  sinon si tous les slots de self répondent sûr et self est une classe définie
    et O appartient à toutes les super-classes descriptives de self alors **sûre**
  sinon **possible**

Si sûre ou possible alors envoyer un message classify(O) à toutes les sous-classes directes de self

Fig. 92 Méthode de classification pour une classe de frames

De la même façon que le message :

```
[O 'r+ 'C]
```

fait évoluer O vers la classe C ($o \in \mathcal{E}[c]$), nous pouvons introduire :

[o 'i+ 'c] pour o ∈ $\mathcal{I}$[c]
[o 'p+ 'c] pour o ∈ $\mathcal{P}$[c].

Appariement des slots

L'appariement des slots peut être vu comme l'appariement entre une classe et un objet. En effet, le slot décrit pour l'objet a ses propres attributs qui le décrivent, et ses attributs doivent satisfaire les conditions de la classe de slots. Le slot de l'objet est décrit par trois attributs qui interviennent pour l'appariement : nom, valeur et as (cf. page 198). Leurs valeurs doivent correspondre aux contraintes de la classe de slots, exprimées dans les attributs de classe : nom, domaine et ref (cf. également page 198). On peut attribuer un type à ces attributs (ou facettes) selon le rôle qu'ils jouent pour l'appariement du slot à la classe de slots, de manière analogue aux slots : FNV, FNE et FD. Selon qu'il s'agisse de slots SNE, SNV, ou SD et de slots homonymes ou univoques, les facettes sont typées différemment (cf. Tableau 3).

Tableau 3 : typage des facettes pour appariement

	SNV		SNE		SD	
	univoque	homonyme	univoque	homonyme	univoque	homonyme
nom	FNV					
valeur	FNV		FNE		FD	
as	FD	FNV	FD	FNV	FD	FNV

Les classes de slots peuvent être considérées comme des classes définies. L'appariement entre une classe de slots et un slot correspondant dans la description de l'objet dépend donc des valeurs (ou absences de valeurs) pour les facettes `nom`, `valeur` et `as`, et de leur typage. Selon le résultat, la classe de slots déduit la conséquence pour l'appariement au niveau de la classe de frames. Ainsi, l'appariement manqué pour un slot SNE (à cause d'une manque d'identification ou d'une violation de contrainte sur la valeur) se traduit par la réponse "impossible" à la classe de frames. Par contre, l'appariement manqué ou seulement "possible" pour un slot SD peut, dans certains cas, résulter en une réponse "sûr" à la classe de frames (dans les cas où l'identité du slot de l'objet est différente de celle de la classe, ou inconnue). Chaque classe de slots traduit donc le résultat de son appariement en un résultat partiel pour l'appariement de la classe de frames.

Bien entendu, pour tous les types de slots, l'appariement n'a lieu que quand un slot de même nom est trouvé dans la description de l'objet[1] :

Soit s mon nom
S'il existe un slot de nom s dans les slots absents pour O ($\mathcal{A}_o$)
alors **impossible**
sinon s'il existe un slot de nom s dans les slots présents pour O ($\mathcal{E}_o$)
alors **apparier**
sinon **possible** --- j'appartiens à $\mathcal{U}_o$
fsi

La déclaration d'absence de slot entraîne directement la réponse "impossible" à la classe de frames. Les slots non évoqués sont bons pour la réponse "possible".

L'appariement proprement dit est fondé sur les facettes `valeur` et `as` (car on n'apparie que les slots dont la facette `nom` est valide, i.e. identique à celle de la classe de slots). Les réponses fournies à la classe de frames sont répertoriées dans les tables de décision ci-dessous, pour chaque type de slots, dans le cas homonyme et univoque. Pour chacune des facettes, la valeur peut être valide, invalide, ou indéterminée ("??"). Etant donné que la description des objets est fondée sur un choix de descriptions de slots possibles, nous excluons le choix d'un point de vue inexistant sur un slot univoque. La facette `as` peut donc soit être valide, soit indéterminée dans le cas des slots univoques.

P, I, et S désignent "possible", "impossible" et "sûr".

Slots homonymes

as \ valeur	valide	¬valide	??
valide	S	I	P
¬valide	P	P	P
??	P	P	P

SNV

as \ valeur	valide	¬valide	??
valide	S	I	S
¬valide	P	P	P
??	P	P	P

SNE

as \ valeur	valide	¬valide	??
valide	S	I	S
¬valide	S	S	S
??	S	P	S

SD

Slots univoques

as \ valeur	valide	¬valide	??
valide	S	I	P
¬valide			
??	S	I	P

SNV

as \ valeur	valide	¬valide	??
valide	S	I	S
¬valide			
??	S	I	S

SNE

as \ valeur	valide	¬valide	??
valide	S	I	S
¬valide			
??	S	I	S

SD

Métaclasses de slots et de frames

Chaque classe de slots possède sa méthode d'appariement selon ces tables. Ces méthodes sont définies dans les métaclasses de slots correspondantes. Nous définissons pour chaque type de slots introduit une métaclasse de slots (cf. le schéma Fig. 93)

1. Rappelons que la définition de la classe `shadowed-rframe` dans le paragraphe précédent n'est pas complète. Des méthodes retournant l'ensemble des slots absents, l'ensemble des slots présents, l'ensemble des slots de nom s, etc. sont définies pour faciliter l'accès à ces informations pour l'appariement des slots.

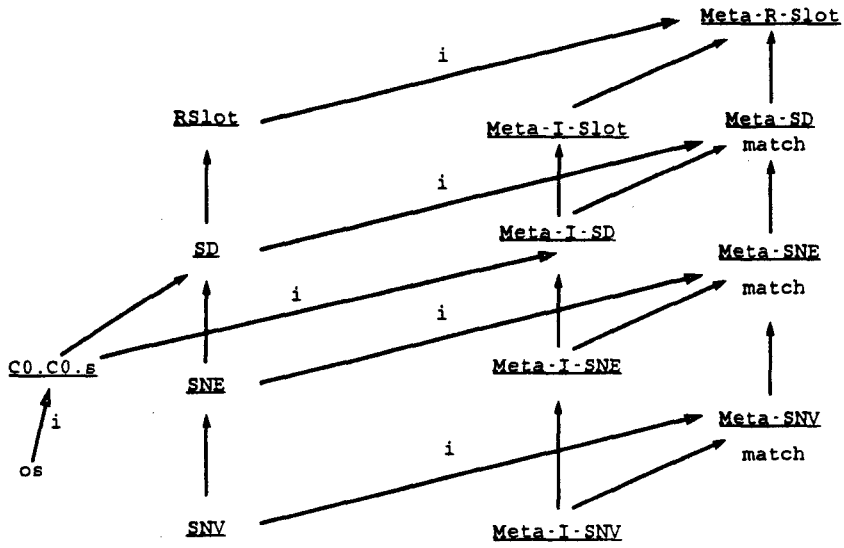


Fig. 93 Les slots dédiés à la classification : classes et métaclasses

Les classes définies et descriptives ont également leurs métaclasses dédiées, qui définissent les méthodes de classification et d'appariement (Fig. 94). Grâce aux métaclasses différentes, le caractère de chaque classe (définie ou descriptive) est facile à retrouver.

Les classes qui ont un comportement de classification sont sous-classes de la classe Concept. Elles sont distinguées en classes définies et classes descriptives, selon les sous-classes Defined et Descriptive. La métaclasse Meta-Concept définit le comportement de classification, qui est hérité par ses sous-métaclasses. Pour pouvoir créer des classes instanciables (comme C0), il y a des méta-i-classes pour les classes définies et les classes descriptives.

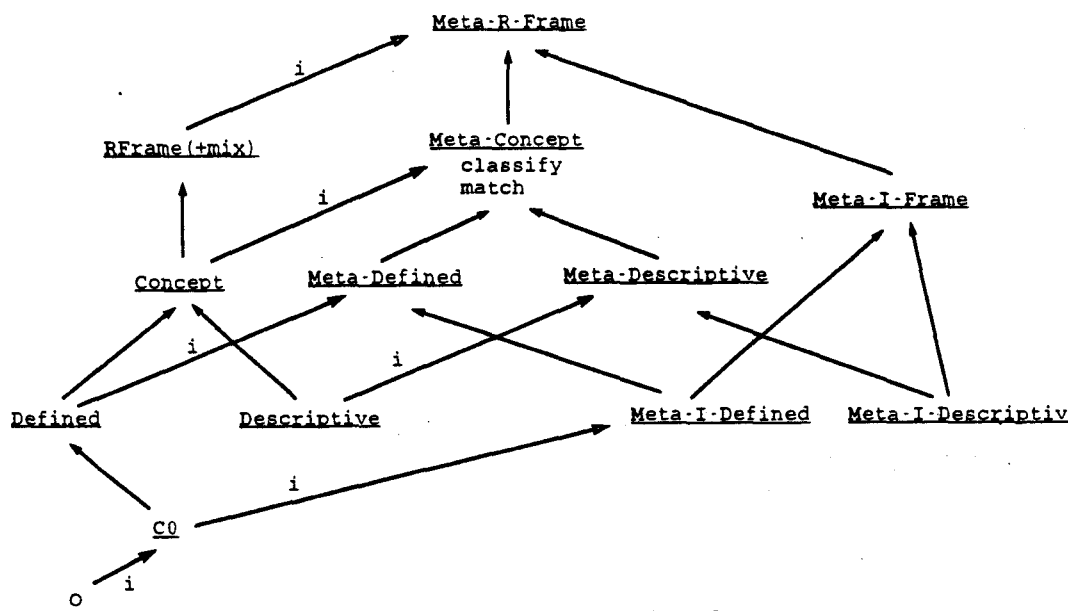


Fig. 94 Les classes qui savent classifier et leurs métaclasses

Dans la description des slots lors de la définition d'une classe, le rôle de chaque slot (SD, SNE ou SNV) est spécifié à l'aide de la facette `$supercl` : on spécifie l'héritage des classes de slots. Le calcul automatique des méta-classes de slots (en fonction des méta-classes des super-classes de slots) assure la création de la classe de slots comme instance de la bonne méta-(i)-classe.

```
[Meta-I-Defined 'new 'C0 'supercl 'Defined
 'slots '(s ($supercl SD $type …)…]
```

⇒

```
[Meta-I-SD 'new 'C0.C0.s 'supercl 'SD
 '$frame 'C0 '$name 's '$type …]
```

4
Conclusion de la partie II

Nous avons successivement étudié les différentes interprétations de la notion de classe et l'appariement associé à chacune des interprétations. La classe prise comme une description de ses instances contient des conditions nécessaires et non suffisantes d'appartenance. C'est pourquoi l'appariement d'un objet avec une classe descriptive ne peut donner plus que l'appartenance possible ou impossible de l'objet à la classe. C'est ce que nous avons appelé l'interprétation de l'approche objet.

L'approche terminologique introduit la notion de classe définie, avec des conditions nécessaires et suffisantes d'appartenance. L'appariement dans un environnement défini fournit l'appartenance sûre en cas de succès. L'échec de l'appariement ne fournit pas d'information particulière, à savoir si l'échec est dû à un manque d'information ou à une violation de contrainte. Dans les systèmes terminologiques, l'appariement est fondé sur la subsomption.

Nous avons défini un environnement plus riche que celui de l'approche descriptive pure, en y introduisant des classes définies. Le bénéfice pour la classification est l'appariement sûr qui peut être établi dans certains cas pour les classes définies de la hiérarchie. D'autre part, par rapport à une approche purement définie, fondée sur la subsomption ("tout ou rien"), l'avantage de cette approche mixte est d'obtenir la qualification "possible" pour des classes définies, de la même façon que pour les classes descriptives.

Nous avons retenu la description libre et donc potentiellement complète de l'objet à classifier, ce qui permet de prendre en compte toutes les caractéristiques de l'objet dans l'appariement, sans avoir recours à un processus interactif guidé par la structure de la hiérarchie des classes.

La description non seulement en termes de valeurs d'attributs mais aussi en termes de structure permet de donner des informations quant à l'existence ou l'absence de certains attributs.

Nous avons rendu possible une prise en compte plus nuancée de ces informations sur la structure dans l'appariement, par la distinction des attributs selon leur importance pour l'appartenance à la classe. Ainsi, il y a des slots dont la valeur est indispensable (SNV) et doit figurer dans la description de l'objet pour conclure l'appartenance sûre. D'autres slots doivent nécessairement exister mais la valeur ne joue pas de rôle décisif (SNE). Enfin, les slots déduits (SD) jouent un rôle descriptif et n'ont pas besoin d'être mentionnés dans la description de l'objet.

Enfin, nous avons étudié le problème de l'homonymie des attributs dans le cadre de la classification. La description libre des objets pouvant introduire des problèmes d'ambiguïté dans la désignation de slots différents mais de même nom, nous avons montré que le processus de classification peut prendre en compte cette homonymie dans l'appariement. L'ambiguïté résulte en une qualification "possible" de plusieurs classes alternatives. La désambiguïsation par désignation non-ambiguë des slots dans la description de l'objet permet le passage de "possible" à "sûre". Cette désignation est fondée sur une expression de point de vue, exprimé

par une classe. La classe pouvant être utilisée pour la désignation non-ambiguë n'est pas unique et ne nécessite pas une connaissance absolue de la hiérarchie des classes, mais exprime un point de vue (une partie du graphe).

Nous avons évoqué les éléments qui permettent une implantation de la classification décrite. Les facilités d'ordre méta offertes par le langage sont indispensables pour réaliser la description libre des objets et l'accès à la structure détenue dans les classes. Le processus de classification lui-même est défini par métaprogrammation : les métaclasses appropriées définissent les méthodes de classes qui prennent en compte l'appariement, aussi bien au niveau des classes de frames qu'au niveau des classes de slots. La classification est ainsi entièrement distribuée sur les objets (classes) concerné(e)s, de manière orientée objet. De plus, elle est obtenue par spécialisation et extension du langage.

La classification des instances dans une hiérarchie de classes consiste à faire des appariements successifs entre les classes et l'instance à classifier. Nous avons insisté sur l'importance de l'interprétation de la notion de classe pour cet appariement, d'une part, et sur les éléments comparés (structure, valeurs), d'autre part.

Si on offre la possibilité de distinguer classes définies et classes descriptives, combinée avec la qualification des attributs en nécessairement valués, nécessairement existants ou déduits, cela ne résout pas le problème de la conception de la hiérarchie elle-même. Une méthodologie pour identifier les différents éléments selon leur nature (comparable à la tentative de [Bra&91]) serait nécessaire. En effet, dans la description de la connaissance d'un domaine, et en vue de son utilisation pour identifier des instances, comment déterminer les classes définies, quelles sont les classes descriptives, quels attributs sont nécessaires, lesquels peuvent être déduits ?

Il faut souligner que le choix de faire d'une classe un "concept défini" peut être un choix contextuel : dans certaines circonstances, on peut accepter que la classe représente une définition qui, "dans l'absolu", n'aurait pas été acceptable. On fait (presque) toujours abstraction de certaines caractéristiques et toute classe définie (et même descriptive) est entâchée de compromis.

Il faut donc avant tout être conscient des implications des choix que l'on fait.

En plus d'une aide méthodologique à la conception, il serait souhaitable d'introduire plus de souplesse, pour pouvoir changer facilement de modalités :

- changer des classes descriptives en classes définies ou vice-versa
- changer des SNV en SNE ou en SD

etc.

De plus, à l'aide méthodologique, il conviendrait d'ajouter une aide pratique pour prévenir les incohérences lors de la définition des classes, en particulier concernant la description des slots SD et la déclaration des relations d'exclusion entre classes.

Sur le plan de la description libre, on peut prévoir plusieurs extensions. Plutôt que de décrire des valeurs précises pour les attributs, on pourrait affirmer des domaines de valeurs (cf. facettes d'instance *$domain*, *$type*,... I 4.4.1). Il faudrait en particulier approfondir la négation dans tous ses aspects :

- différence de valeur pour un attribut
- négation de point de vue
- négation d'appartenance

- les conditions d'exploitation du tiers-exclu

ainsi que la disjonction.

Conclusion Générale

Bilan

Les deux parties de ce mémoire sont consacrées respectivement à la définition d'un modèle de représentation par objets et à son exploitation par un raisonnement de classification.

Après l'étude, dans le chapitre 2, des concepts issus des courants PPO et RPO, avec quelques éléments d'inspiration PPL, nous avons présenté les concepts retenus qui sous-tendent notre modèle hybride FROME. Ce modèle est exposé dans le chapitre 3.

Rappelons l'importance du modèle classe-instance pour la représentation à différents niveaux d'abstraction. Nous combinons les aspects structurels et comportementaux des frames avec les méthodes et l'envoi de message issus de la PPO.

L'héritage multiple avec points de vue et la représentation multiple et évolutive sont les aspects fondamentaux du projet ROME, que nous avons retenus et appliqués aux frames.

Nous insistons sur la conception orientée objet du modèle FROME, exposée dans le chapitre 4. Grâce au caractère méta-circulaire du langage ROME, nous avons obtenu le modèle FROME par extension méta-programmée, assurant l'héritage des notions fondamentales citées ci-dessus. Le modèle obtenu est extensible à son tour, comme nous l'avons montré entre autres par la conception orientée objet du filtrage.

La partie II examine en détail les problèmes liés à la classification d'instances.

Le chapitre 2 met en évidence la dépendance de l'appariement par rapport à la sémantique de la notion de classe. Celle-ci peut être comprise comme une description ou une définition. En fonction de la sémantique, l'appartenance d'un objet à une classe ne peut être décidée de la même façon.

La description de l'objet à classifier dépend du modèle sous-jacent. Nous avons constaté que l'appariement est le mieux servi par la disponibilité d'une description complète de l'objet. Cependant, la description complète est contraire à l'instanciation.

Nous avons exposé les problèmes posés par la présence d'attributs homonymes, en particulier dans la classification. Aucun système étudié ne semble traiter l'homonymie dans ce cadre.

Le chapitre 3 tente de prendre en compte les problèmes identifiés dans le cadre de FROME. Nous y introduisons des classes définies et des classes descriptives. L'appariement tient compte de la description complète de l'instance et se fonde sur la structure et sur les valeurs d'attributs. Les attributs sont qualifiés selon leur importance pour l'appariement.

L'homonymie des attributs est traitée par une désignation non-ambiguë ou prise en compte comme élément non-déterministe dans la classification.

Nous avons donné les éléments pour une implantation méta-programmée de la classification.

Extensions et Perspectives

De nombreuses voies restent à explorer, tant dans la définition du modèle lui-même, que dans son application pour le filtrage et la classification d'instances.

Méta-circularité du modèle

Nous avons vu qu'un modèle métacirculaire a la propriété très intéressante d'être facilement extensible par la définition de nouvelles métaclasses. On peut ainsi définir de nouvelles structures (par exemple des frames) et de nouveaux comportements (par exemple du filtrage). Cependant, ces extensions successives s'opèrent "en couches" : la nouvelle fonctionnalité est disponible sur des objets spécialisés et non pas sur le système entier. Il en résulte un système de plus en plus compliqué, que les combinaisons croisées des extensions risquent d'alourdir.

Veut-on obtenir un modèle aussi uniforme que le noyau métacirculaire de départ, il faut alors réintégrer dans un noyau plus riche les extensions que l'on a apportées. Le modèle métacirculaire de FROME serait un noyau "tout frame" (classes, slots, facettes sont des frames).

Dans l'optique de la classification en FROME, il serait envisageable de prévoir dans ce noyau une distinction entre classes définies et classes descriptives. Une étude plus approfondie sera nécessaire pour déterminer jusqu'où ces notions doivent/peuvent être définies de manière métacirculaire.

Maintien de la classification

Inspirés par [Neb90], nous avons unifié l'appariement pour nos classes définies et descriptives par l'introduction de composantes primitives, matérialisant la partie définitoire manquante des classes descriptives. Ces composantes primitives doivent être affirmées par l'utilisateur, par l'expression de l'appartenance à une classe, pour permettre le passage d'une classe possible à sûre. D'autres éléments qui conditionnent ce passage sont l'identification complète de slots ambigus, la valuation de slots nécessairement valués, l'affirmation de l'existence de slots nécessairement existants...

Plutôt que de distinguer les étapes "appartenance possible" et "appartenance sûre", nous envisageons d'approfondir la notion d'*appartenance conditionnelle* à une classe. Lors de la classification, l'objet classifié établirait pour chaque classe examinée une conjonction de conditions, le séparant de l'appartenance sûre à cette classe. Si cette conjonction contient une contradiction, la classe en question est impossible. Si la conjonction est réduite à vide (i.e. les conditions sont vérifiées), l'appartenance sûre est établie. Un ensemble de conditions non-vide et non-contradictoire équivaut à une appartenance possible. De cette manière, on peut obtenir à tout moment l'ensemble des conditions qui restent à vérifier pour qu'une classe soit sûre pour l'objet. Une modification intervenant sur l'objet (nouvelle affirmation, valuation d'attribut,...) amènerait à une reconsidération de ces conditions, pour voir si certaines d'entre elles sont violées ou vérifiées et entraîneraient respectivement l'appartenance impossible ou (le rapprochement de) l'appartenance sûre. Cette technique s'apparente à celle des TMS et ressemble également au mécanisme mis en œuvre dans le *recognizer* de LOOM (cf. II 2.2.2.1).

Stratégie guidée par les attributs

Au paragraphe 3.8 traitant de la conception orientée objet de la classification, nous avons évoqué la redistribution de l'appariement vers les classes de slots. L'analogie de l'appariement au niveau de la classe de slots et des slots décrits pour l'objet ouvre la voie à une autre stratégie de la classification : une stratégie guidée par la classification des attributs. A partir de la classification des slots présents dans la description de l'objet, on pourrait retrouver les classes de frames qui les définissent. L'intersection des classes de frames ainsi retenues pourrait constituer l'ensemble des classes candidates pour l'appariement. La prise en compte dans cette approche des différents types de slots, de l'appariement sûr, possible, impossible avec les classes de slots doit être étudiée plus en détail. Elle présente des analogies avec l'approche de classification des classes par les types, proposée par [Cap92].

Apports mutuels des approches objet et terminologique

L'intégration de classes définies dans le modèle ouvre des perspectives pour un approfondissement de notre solution au filtrage. Les classes définies rendent possible la classification des classes, pratiquée par les systèmes terminologiques. La classification pourrait opérer le placement des classes filtres, qui n'est pas réalisé dans la solution actuelle.

D'autres perspectives liées à la classification des classes sont l'acquisition par insertion de nouvelles classes et la maintenance de hiérarchies de classes, ainsi que la fusion de plusieurs bases de connaissances. Nous évoquons dans ce cadre les travaux en cours sur l'intégration des schémas dans le domaine des bases de données [Pat&92][Are&93][She&93][Bla&94]. Ces travaux s'inspirent également fortement des langages terminologiques.

La prise en compte de l'homonymie dans le cadre de la classification contribue également à la réactualisation des problèmes d'homonymie et de points de vue dans l'approche terminologique.

Notre travail constitue un premier pas vers une intégration des deux approches.

Prise en compte de méthodes et contraintes

Notre solution pour la classification des instances repose exclusivement sur l'appariement de structure et de valeurs d'attributs. Comme les autres systèmes étudiés dans la partie II, nous faisons entièrement abstraction des aspects comportementaux (méthodes) dans le cadre de la classification. La prise en compte de ces aspects pourra s'inscrire dans une optique de génie logiciel et s'intéressera par exemple au maintien d'une base de composants logiciels, classifiés selon leur comportement. Elle nécessite la description et la manipulation de ces aspects au niveau méta, en spécifiant par exemple la signature des méthodes, afin de pouvoir les comparer et les ordonner selon une sémantique de spécialisation bien définie. Les simples lambda-expressions des méthodes FROME ne permettent pas cette prise en compte. On rejoint ici les remarques faites à propos de la méta-circularité du modèle FROME lui-même.

D'autres éléments à étudier pour une prise en compte dans la classification sont les contraintes faisant intervenir plusieurs attributs. A la manière des "role-value-maps" de la tradition terminologique, celles-ci sont susceptibles d'introduire des complications pour la décision de l'appariement.

Classifications multiples et contextualisation

Dans l'étude de la multiplicité des représentations, nous avons évoqué l'une des origines de cette multiplicité : l'objectif qui a inspiré la hiérarchie décrite. Dans le cadre des travaux de l'équipe ROME, la combinaison de la notion de point de vue avec celle, plus élaborée, de contexte, est actuellement à l'étude.

Il s'agit de décrire un domaine sous la forme d'une hiérarchie de classes, laquelle constitue une première représentation des objets de ce domaine et de contextualiser ensuite les objets ainsi représentés pour des applications différentes. La contextualisation consiste en une apposition de point de vue sur la hiérarchie de base et d'enrichir celle-ci dans le contexte particulier d'une application, chaque application nécessitant un affinement des connaissances représentées. Chaque contextualisation induit une classification supplémentaire, affinée et "parallèle" à la hiérarchie de base.

D'un point de vue méthodologique, cette approche fournit un guide à la conception incrémentale par contexte.

Un premier prototype (CROME) a été implanté en ROME [Rio&93], en utilisant directement ses capacités de représentation multiple et évolutive pour la contextualisation des objets. La contextualisation du domaine pour la description de hiérarchies spécialisées est obtenue par projection de la hiérarchie de base. Le domaine d'application choisi était la CAO électronique, avec des contextualisations pour la simulation et pour la documentation des circuits de composants.

Nous envisageons d'appliquer cette approche aux frames et d'étendre FROME aux contextes. L'objectif est d'étudier plus finement les relations entre hiérarchies afin de les exprimer et de les exploiter. Il s'agit également d'élargir la première approche qui s'est restreinte à la projection de la hiérarchie de base, classe par classe et qui produit donc des hiérarchies contextualisées de même structure. Un contexte d'application privilégié sera la représentation des connaissances médicales, et plus particulièrement les connaissances intervenants pour l'élaboration du plan de soins. Cette application viendra s'inscrire dans le cadre d'une étude menée actuellement au CERIM[1] sur la coopération dans les Unités de Soins Intensifs. Nous étudierons la possibilité de représenter la connaissance des différents intervenants (médecins, infirmières...) sur le malade, les médicaments,... sous la forme de classifications de base du domaine commun contextualisées selon les spécificités des tâches de chacun.

Hiérarchies multiples et classification d'instances

L'idée d'une classification de base accompagnée de classifications contextualisées peut être appliquée à la classification d'instances. En fait, les procédures d'identification couramment utilisées en Botanique (identification des plantes, identification des champignons) suivent rarement la hiérarchie des catégories que constitue la taxonomie scientifique du domaine [Leb&91], partant de la racine vers les feuilles. D'une part, cette taxonomie contient souvent des caractéristiques non observables par la personne sur le terrain, et d'autre part, elles ne sont pas décrites dans l'ordre qui permet d'identifier rapidement un individu. La classification

1. Centre d'Etudes et de Recherches en Informatique Médicale. Une thèse du LIFL y est en cours sur les aspects du contrôle de la communication dans les Unités de Soins.

d'instances selon un parcours interactif a donc peu d'intérêt de s'appuyer sur une telle taxonomie, appelée aussi "clé naturelle"[1]. Les manuels présentent des clés artificielles, ne mentionnant que les caractéristiques pertinentes pour l'identification et dans un ordre qui favorise l'identification rapide [Bon88][Cha82].

Même si les clés naturelles et les clés artificielles ne doivent pas être confondues pour l'identification des individus, il s'agit de plusieurs représentations du même domaine. Il nous semble intéressant de mettre ces représentations en relation, afin de profiter des informations qu'elles contiennent. La clé naturelle pourrait jouer le rôle de hiérarchie de base (ou hiérarchie "profonde"), alors que les clés artificielles seraient des hiérarchies spécialisées dans le contexte d'une application particulière : l'identification d'individu. Les différentes hiérarchies (hiérarchies "superficielles") induites par les clés artificielles accentuent alors des aspects différents ou privilégient un ordre particulier dans la structuration des caractéristiques.

L'exploitation de ces hiérarchies multiples passe, encore une fois, par l'identification et l'expression des relations entre elles. Lors de l'avancement du processus d'identification, les relations avec la hiérarchie de base permettrait d'avoir des informations intermédiaires sur le résultat partiel. C'est justement ce résultat partiel qui est inaccessible à l'utilisateur d'une clé artificielle. Les moyens d'expression de ces relations restent donc à explorer. Quelques pistes sont fournies par TROPES au moyen des passerelles et par l'expression de règles de correspondance décrites dans [Har&93].

On pourra confronter ces idées également au diagnostic dans le domaine médical.

1. Notons que dans la classification pour FROME, nous avons voulu nous affranchir de l'ordre imposé par la hiérarchie, en proposant une description libre des objets.

Références bibliographiques

[Ada&88] Adam-Nicolle A., Victorri B., Madelaine J., Porquet C., Revenu M., *AIRELLE Rapport de Présentation*, Les Cahiers du LIUC no. 88-3, Laboratoire d'Informatique de l'Université de Caen, 1988.

[Ada90] Adam-Nicolle A., *Le Métalangage AIRELLE*, Les Cahiers du LIUC no. 90-5, Laboratoire d'Informatique de l'Université de Caen, avril 1990.

[Age&92] Agesen O., Bak L., Chambers C., Chang B., Hölzle U., Maloney J., Smith R.B. and Ungar D., *How to Use SELF 2.0*, Sun Microsystems Inc. and Stanford University, 1992.

[Aik83] Aikins J., "Prototypical Knowledge for Expert Systems," *Artificial Intelligence*, 20:163-210, 1983.

[Aït&86] Aït-Kaci H. and Nasr R., "LOGIN: A Logic Programming Language with Built-in Inheritance," *Journal of Logic Programming*, 3:185-215, 1986.

[Aït91a] Aït-Kaci H., "A Glimpse of Paradise," in *Next Generation Information System Technology*, J.W. Schmidt and A.A. Stogny, editors, Proc. of the 1st. International East/West Data Base Workshop, Kiev, 1990, LNCS 504, pp. 15-25, Springer-Verlag, Berlin, Germany, 1991.

[Aït91b] Aït-Kaci H., "An Overview of LIFE," in *Next Generation Information System Technology*, J.W. Schmidt and A.A. Stogny, editors, Proc. of the 1st. International East/West Data Base Workshop, Kiev, 1990, LNCS 504, pp. 42-58, Springer-Verlag, Berlin, Germany, 1991.

[Alb88] Albert P., "Kool: merging object frames and rules," in Demongeot J., Hervé T., Rialle V. and Roche C., Eds., *Artificial Intelligence and Cognitive Sciences*, pp. 15-21, Manchester University Press, 1988.

[Ame87] America P., "Inheritance and Subtyping in a Parallel Object-Oriented Language," in *Proc. of ECOOP'87*, pp. 281-289, 1987.

[Ame90] America P., "Designing an Object-Oriented Programming Language with Behavioural Subtyping," in *Proc. of the Workshop on Foundations of Object-Oriented Languages (FOOL)*, Noordwijkerhout, The Netherlands, Lecture Notes in Computer Science No. 489, Springer-Verlag, 1990.

[Are&93] Arens Y., Chee C.Y., Hsu C. and Knoblock C.A., "Retrieving and Integrating Data From Multiple Information Sources," *International Journal of Intelligent and Cooperative Information Systems*, Vol. 2, No. 2, pp. 127-158, World Scientific Publishing Company, 1993.

[Att&85] Attardi G. and Simi M., "Metalanguage and Reasoning Across Viewpoints," in *Proceedings of ECCAI 1985, Advances in Artificial Intelligence*, T. O'Shea, Ed., Elsevier Science Publishers B.V. (North-Holland), 1985.

[Att&86] Attardi G., Corradini A., Diomedi S. and Simi M., *Taxonomic Reasoning*, Technical Report ESP/86/2, DELPHI SpA, Viareggio, Italy, 1986.

[Att&89] Attardi G., Bonini C., Boscotrecase M.R., Flagella T. and Gaspari M., "Metalevel Programming in CLOS," in *Proc. of ECOOP'89*, pp. 243-256, 1989.

[Aug&91] Augeraud M. and Freeman-Benson B., "Dynamic Objects," in *Proc. of UIST '91*, November 11-13, South California, pp. 135-140, ACM, 1991.

[Baa&92] Baader F., Hollunder B., Nebel B., Profitlich H.-J. and Franconi E., "An Empirical Analysis of Optimization Techniques for Terminological Representation Systems or: Making KRIS get a move on," In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, Oct. 25-29, B. Nebel, C. Rich, W. Swartout, Eds., pp. 270-281, Cambridge, Massachusetts, 1992.

[Bat&91] El Batti C. et Champignon P., *GEROME Graphic Environment for Rome*, Rapport de Projet, Ecole Universitaire D'Ingénieurs de Lille, LIFL, 1991.

[Bec&89] Beck H.W., Gala S.K. and Navathe S.B., "Classification as a Query Processing Technique in the CANDIDE Semantic Data Model," *Proc. of the Fifth International Conference on Data Engineering*, pp. 572-581, Los Angeles, CA, 1989.

[Ben&89] Benoit Ch., Bidoit M., Henninger L. and Velly R., "LORE: An Object-Based Programming Environment," In *Proc. of the First International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'89)*, Paris, november 1989.

[Bla&87] Blake E. and Cook S., "On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk," In *Proceedings of ECOOP'87*, pp. 45-54, 1987.

[Bla&92] Blair P., Guha R.V. and Pratt W., *Microtheories: An Ontological Engineer's Guide*, MCC Technical Report Number CYC-050-92, Austin, March 1992.

[Bla&94] Blanco J.M., Illarramendi A., Goñi A. and Pérez J.M., "Using a Terminological System to Integrate Relational Databases," in *Proc. of Basque International Workshop on Information Technology (BIWIT'94)*, Biarritz, Ed. C. Chrisment, Cépaduès-Editions, Toulouse, 1994.

[Bob&77] Bobrow D.G. and Winograd T., "An Overview of KRL, a Knowledge Representation Language," *Cognitive Science*, Vol. 1, 1977.

[Bob&83] Bobrow D.G. and Stefik M.S, *The LOOPS Manual*, Memo KB-VLSI-81-13, Xerox PARC, 1983.

[Bon88] Bon M., *Champignons d'Europe occidentale*, Les Editions Arthaud, 1988.

[Bor&82] Borning A.H. and Ingalls D.H., "Multiple Inheritance in Smalltalk-80," in *Proc. of the AAAI'82 Conference*, pp. 234-237, Pittsburgh, 1982.

[Bor&89] Borgida A., Brachman R.J., McGuinness D.L., Resnick L.A., "CLASSIC: A Structural Data Model for Objects," *Proc. of the ACM./SIGMOD International Conference on the Management of Data, Portland, Oregon, SIGMOD RECORD*, 18(2), pages 59-67, 1989.

[Bor79] Borning A., *Thinglab — A Constraint-Oriented Simulation Laboratory*, Report SSL-79-3, Stanford, 1979.

[Bor86] Borning A.H., "Classes Versus Prototypes in Object-Oriented Languages," in *Proc. Fall Joint Computer Conference '86*, pp. 36-40, ACM/IEEE, Dallas, Texas, 1986.

[Bra&83] Brachman R.J., Fikes R.E. and Levesque H.J., "Krypton: A Functional Approach to Knowledge Representation," *Computer*, IEEE, October 1983.

[Bra&84] Brachman R.J. and Levesque H.J., "The Tractability of Subsumption in Frame-Based Description Languages," In *Proceedings of the 4th National Conference of the AAAI*, pp. 34-37, Austin, Texas, 1984.

[Bra&85] Brachman R.J. and Schmolze J.G., "An Overview of the KL-ONE Knowledge Representation System," *Cognitive Science*, 9(2):171-216, 1985.

[Bra&90] Bracha G. and Cook W., "Mixin-based Inheritance," *Proc. of the First Joint ECOOP/OOPSLA Conference*, pp. 303-311, Ottawa, Canada, 1990.

[Bra&91] Brachman R.J., McGuinness D.L., Patel-Schneider P.F., Resnick L.A. and Borgida A., "Living with CLASSIC: When and How to Use a KL-ONE-Like Language," in *Principles of Semantic Networks, Explorations in the Representation of Knowledge*, Ed. J.F. Sowa, Morgan Kaufmann Publishers, San Mateo, California, 1991.

[Bra77] Brachman R.J., "What's in a Concept: Structural Foundations for Semantic Networks," *International Journal of Man-Machine Studies*, 9:127-152, 1977.

[Bra83] Brachman R.J., "What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks," *IEEE Computer*, 16(10):30-36, 1983.

[Bra85] Brachman R.J., ""I Lied about the Trees" Or, Defaults and Definitions in Knowledge Representation," *The AI Magazine*, pp. 80-93, Fall, 1985.

[Bra92] Brachman R.J., ""Reducing" CLASSIC to Practice: Knowledge Representation Theory Meets Reality," In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, Oct. 25-29,

B. Nebel, C. Rich, W. Swartout, Eds., pp. 247-258, Cambridge, Massachusetts, 1992.

[Bri&89] Briot J.P. et Cointe P., "Programming with Explicit Metaclasses in Smalltalk-80," in *Proc. of OOPSLA '89*, pp. 419-431, 1989.

[Bul90] *Intelligence Artificielle ; Introduction à Kool*, Bull S.A. CEDOC-DILOG, Val de Reuil, avril 1990.

[Byl&85] Bylander T., Mittal S. and Chandrasekaran B., "CSRL: A Language for Expert Systems for Diagnosis," *Comp. & Maths. with Appls.*, Vol. 11, No. 5, pp. 449-456, 1985.

[Byl&86] Bylander T. and Mittal S., "CSRL: A Language for Classificatory Problem Solving and Uncertainty Handling," *AI Magazine*, August 1986.

[Byl&89] Bylander T., Johnson T.R. and Goel A., "Structured Matching: A Task-Specific Technique for Making Decisions," *IEEE Int. Workshop on Tools for AI*, Fairfax, Virginia, 1989.

[Cap92] Capponi C., *Acquisition des connaissances et dynamique des classes dans une représentation de connaissances par objets*, DEA IMAG, LIFIA, Grenoble, 1992.

[Car&85] Cardelli L., Wegner P., "On understanding types, data abstraction, and polymorphism", *ACM Computing Surveys*, Vol. 17, No. 4, PP. 471-522, December 1985.

[Car&90a] Carré B., Dekker L. and Geib J.M., "Multiple and Evolutive Representation in the ROME Language," *Proc. of the Second International Conference TOOLS*, Bézivin J., Meyer B., Nerson J.M., Eds., Paris, 1990.

[Car&90b] Carré B. and Geib J.M., "The Point of View Notion for Multiple Inheritance," *Proc. of the First Joint ECOOP/OOPSLA Conference*, Ottawa, Canada, 1990.

[Car&91a] Carré B. and Dekker L., *FROME The Manual*, Technical Report ERA 92, LIFL, 1991.

[Car&91b] Carré B. and Dekker L., "Inheriting Object-Oriented Features through Meta-Programming, A Frame Extension to ROME," *Proc. of the East European Conference on Object-Oriented Programming,* Bratislava, Czecho-Slovakia, 1991.

[Car89] Carré B., *Méthodologie orientée objet pour la représentation des connaissances, concepts de point de vue, de représentation multiple et évolutive d'objet*, Thèse de Doctorat en Informatique, LIFL, Université de Lille, 1989.

[Car92] Carn N., *Représentation orientée objet de système opérationnel avec application au domaine spatial*, Thèse de doctorat en informatique, I.N.P. Toulouse, 1992.

[Cas85] Caseau Y., "LORE, la définition par sélection," In *Actes des Journées AFCET-GROPLAN sur les langages et leurs environnements*, Bigre+Globule No. 46, Novembre 1985.

[Cas91a] Casais E., *Managing Evolution in Object Oriented Environments: An Algorithmic Approach*, Thèse de la Faculté des sciences économiques et sociales de l'Université de Genève, 1991.

[Cas91b] Caseau Y., "An Object-Oriented Language for Advanced Applications," in *Proc. of TOOLS USA '91*, pp. 153-166, 1991.

[Cha88] Chabre B., *Réalisation d'une fonction de filtrage dans le langage orienté objet YAFOOL*, Rapport de stage ENST, SEMA-METRA, 1988.

[Cha&91] Chambers C., Ungar D., Chang B. and Hölzle U., "Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF," in *Lisp and Symbolic Computation: An International Journal*, 4, 3, 1991, Kluwer Academic Publishers.

[Cha83] Chandrasekaran, B., "Towards a Taxonomy Of Problem Solving Types," *AI Magazine*, Winter/Spring, 1983.

[Cha85] Chandrasekaran, B., "Generic Tasks in Knowledge-Based Reasoning: Characterizing and Designing Expert Systems at the "Right" Level of Abstraction," *IEEE*, 1985.

[Cha82] Chassain M., *Choisir ses champignons*, Technique et Documentation Lavoisier, Paris, 1982.

[Che&89] Chein M. et Cogis O., "Un modèle pour la programmation orientée réflexe," *Actes du 7ème Congrès AFCET Reconnaissance des Formes et Intelligence Artificielle*, pp. 1623-1630, Paris, 1989.

[Coi87] Cointe P., "Metaclasses are First Class: The ObjVlisp Model," *Proc. of the 2nd OOPSLA* , Orlando, Florida, *Special issue of ACM Sigplan Notices*, 22(12):156-167, 1987.

[Coi&92] Cointe P., Malenfant J., Dony C. et Mulet P., "Etude de la réflexion de comportement dans le langage SELF," *Actes de la 1ère Conférence sur la Représentation par Objets*, La Grande Motte, 22-23 Juin 1992, EC2, Ed., 1992.

[Dav87] Davis H.E., *VIEWS: Multiple Perspectives and Structured Objects in a Knowledge Representation Language*, Bachelor and Master's Thesis, Department of Electrical Engineering and Computer Science, MIT, 1987.

[Dek89] Dekker L., *Des objets aux frames en (passant par) ROME*, Mémoire de DEA, LIFL, Université de Lille, 1989.

[Dek92] Dekker L., *Les tâches génériques selon Chandrasekaran*, Rapport Interne ERA-92-116, LIFL, Université de Lille, 1992.

[Dek93] Dekker L., "La réification du filtrage en Frome," *Actes de la 2ème Conférence sur la Représentation par Objets*, La Grande Motte, 17-18 juin 1993, EC2, Ed., 1993.

[Dek&92] Dekker L. et Carré B., "Multiple and Dynamic Representation of frames with Points of View in FROME," *Actes de la 1ère Conférence sur la Représentation par Objets*, La Grande Motte, 22-23 Juin 1992, EC2, Ed., 1992.

[DeM&87] DeMichiel L.G. and Gabriel R.P., "The Common Lisp Object System: An Overview," in *Proc. of ECOOP'87*, pp. 201-220, 1987.

[Dev&90] Devanbu P., Brachman R.J., Selfridge P.G. and Ballard B.W., "LaSSIE: a Knowledge-based Software Information System," *Proc. of the International Conference on Software Engineering 1990*, IEEE Computer Society 1990.

[Don&92] Dony C., Malenfant J. and Cointe P., "Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation," in *Proc. of OOPSLA'92*, pp. 201-217, 1992.

[Doy&91] Doyle J. and Patil R.S., "Two theses of knowledge representation: language restrictions, taxonomic classification, and the utility of representation services," *Artificial Intelligence*, 48(3):261-297, 1991.

[Duc&86] Ducournau R., Quinqueton J., *YAFOOL, encore un langage objet à base de frames !*, Version 2.1, Rapport Technique No. 72, INRIA Rocquencourt, 1986.

[Duc&87] Ducournau R. and Habib M., "On Some Algorithms For Multiple Inheritance in Object Oriented Programming," in *Proc. of ECOOP'87*, pp. 291-300, 1987.

[Duc&89] Ducournau R. et Habib M., "La multiplicité de l'héritage dans les langages à objets," *Technique et Science Informatiques*, Vol. 8, No. 1, AFCET-Bordas, 1989.

[Duc&92a] Ducournau R., Habib M., Huchard M., Mugnier M.L. and Napoli A., "Le Point sur l'Héritage Multiple," Draft, mai 1992, à paraître dans TSI.

[Duc&92b] Ducournau R., Habib M., Huchard M. and Mugnier M.L., "Monotonic Conflict Resolution Mechanisms for Inheritance," in *Proc. of OOPSLA '92*, pp. 16-24, 1992.

[Duc91] Ducournau R., *Y3 : Yafool, Yaflog, Yafen*, Version 3.6, Manuel de référence, SEMA GROUP, Montrouge, 1991.

[Duc93] Ducournau R., *Héritages et Représentations*, Habilitation à diriger des recherches, Spécialité Informatique, Université de Montpellier II, 1993.

[Dug88] Dugerdil P., *Contribution à l'Etude de la Représentation des Connaissances fondée sur les Objets, Le Langage Objlog*, Thèse de Doctorat en Informatique, Université d'Aix-Marseille, 1988.

[Dug91] Dugerdil P., "Inheritance Mechanisms in the OBJLOG Language: Multiple Selective and Multiple Vertical with Points of View," in *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, M. Lenzerini, D. Nardi and M. Simi, (eds.), John Wiley & Sons Ltd, 1991.

[Esc&90] Escamilla J. and Jean P., Relationships in an Object Knowledge Representation Model," In *Proceedings of the Conference on Tools for Artificial Intelligence (TAI'90)*, IEEE, Herndon, VA, USA, 1990.

[Euz93] Euzenat J., "On a purely taxonomic and descriptive meaning for classes," In *Proc. of the IJCAI Workshop on Object-Based Representation Systems*, 28 aug. 1993, pp. 81-92, Chambéry, France, 1993, (également disponible comme Rapport No. 93-R-156, CRIN, Nancy).

[Fah&81] Fahlman S.E., Touretzky D.S. and van Roggen W., "Cancellation in a Parallel Semantic Network," in *Proc. of IJCAI'81*, 1981.

[Fau91] Faucher C., *Elaboration d'un langage extensible fondé sur les schémas, le langage OBJLOG+*, Thèse de Doctorat en Informatique, Université d'Aix-Marseille-III, 1991.

[Fer&88] Ferber J. and Volle P., "Using Coreference in Object Oriented Representations," In *Proceedings of the 8th ECAI*, Munich, pp. 238-240, 1988.

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

[Fer88] Ferber J., "Coreferentiality: the Key to an Intentional Theory of Object-Oriented Knowledge Representation," *Artificial Intelligence and Cognitive Sciences*, Manchester University Press, Demongeot J., Hervé T., Rialle V. and Roche C., Eds., 1988.

[Fer89] Ferber J., *Objets et agents : une étude de représentation et de communications en Intelligence Artificielle*, Thèse d'Etat, Paris VI, 1989.

[Fik&85] Fikes R. and Kehler T., "The Role of Frame-based Representation in Reasoning," *Communications of the ACM*, Vol. 28, No. 9, 1985.

[Fin86] Finin Timothy W., "Interactive Classification: A Technique for Acquiring and Maintaining Knowledge Bases," *Proceedings of the IEEE*, Vol. 74, No. 10, October 1986.

[For&89] Fornarino M., Pinna A. et Trousse B., "Approche orientée objet pour la mise en œuvre des relations dans un langage de schémas," In *Actes du 7ème Congrès RFIA* , AFCET/INRIA, 1989.

[Free90] Freeman-Benson B.N., "Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming," In *Proc. of the First Joint ECOOP/OOPSLA Conference*, pp. 77-88, Ottawa, Canada, 1990.

[Gol&77] Goldstein Ira P. and Roberts R.B., *Nudge: A Knowledge-based Scheduling Program*, Report AIM-405, MIT, Cambridge, February 1977.

[Gol&80] Goldstein, Ira P. and Bobrow, Daniel G., "Extending Object Oriented Programming in Smalltalk", *Proceedings of the Lisp Conference*, Stanford University, 1980.

[Gol&83] Goldberg A. and Robson D., *Smalltalk-80, the Language and its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

[Goo79] Goodwin J.W., *Taxonomic Programming with KLONE*, Research Report LITH-MAT-R-79-5, Informatics Laboratory, Linköping University, Linköping, Sweden, 1979.

[Gra88] Granger C., *CLASSIC - Générateur de systèmes experts en classification et en diagnostic*, Manuel de l'Utilisateur V. 2.2 ILOG, 1988.

[Gri&92] Grivaud S. et Rechenmann F., "Navigation dans les bases de connaissances associant objets et hypertexte," *Actes de la 1ère Conférence sur la Représentation par Objets*, La Grande Motte, 22-23 Juin 1992, EC2, Ed., 1992.

[Guh&90] Guha R.V. and Lenat D.B., "Cyc: A Midterm Report," *AI Magazine*, 11 (3), pp. 33-59, 1990.

[Guh91] Guha R.V., *Contexts: A Formalization and Some Applications*, PhD Thesis, Stanford, MCC Technical Report Number ACT-CYC-423-91, 1991.

[Hal&87] Halbert D.C. and O'Brien P.D., "Using Types and Inheritance in Object-Oriented Languages," in *Proc. of ECOOP'87*, pp. 23-34, 1987.

[Ham91] Hamroun H., *FROMER ou les règles de production en FROME*, Mémoire de DEA, LIFL, Université de Lille, 1991.

[Han92] Hanschke Ph., "Specifying Role Interaction in Concept Languages," In *Proceedings of the 3rd International Conference on Principles of Knowledge Representation and Reasoning (KR'92)*, Oct. 25-29, B. Nebel, C. Rich, W. Swartout, Eds., pp. 318-329, Cambridge, Massachusetts, 1992.

[Har&93] Harrison W. and Ossher H., "Subject-Oriented Programming (A Critique of Pure Objects)," in *Proc. of OOPSLA'93*, pp. 411-428, 1993.

[Hat&91] Haton J.-P., Bouzid N., Charpillet F., Haton M.-C., Lâasri B., Lâasri H., Marquis P., Mondot T., Napoli A., *Le raisonnement en Intelligence Artificielle*, InterEditions, 1991.

[Hen86] Hendler J., "Enhancement for multiple-inheritance," in *Proceedings of the 1st OOPSLA, Portland, Oregon, ACM SIGPLAN Notices* 21(10), pp. 98-106, 1986.

[Hol&90] Hollunder B., Nutt W. and Schmidt-Schauß M., "Subsumption Algorithms for Concept Description Languages," In *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI'90)*, 1990.

[Hop&93] Hoppe Th., Kindermann C., Quantz J.J., Schmiedel A. and Fischer M., *BACK V5 Tutorial & Manual*, Technische Universität Berlin, Institut für Software und theoretische Informatik, Projekt KIT-BACK, Berlin, Germany, March 1993.

[ILO91] *SMECI Manuel de référence*, Version 1.5, ILOG, 1991.

[Inf85] *ART Reference Manual*, Inference Corporation, Los Angeles, 1985.

[Int85] *KEE Software Development System User's Manual*, KEE Version 2.1, IntelliCorp, 1985.

[Joh86] Johnson Ralph E., "Type-Checking Smalltalk," *Proc. of the 1st OOPSLA*, pp. 315-321, Portland, Oregon, 1986.

[Jos&87] Josephson J.R., Chandrasekaran B., Smith J.W. Jr., and Tanner M.C., "A Mechanism for Forming Composite Explanatory Hypotheses," *IEEE Transactions on Systems, Man and Cybernetrics*, Vol SMC-17, No. 3, May/ June 1987.

[Kac&86] Kaczmarek Th. S., Bates R. and Robins G., "Recent Developments in NIKL," *Proceedings of AAAI'86*, Philadelphia, Pennsylvaia, pp. 978-985, 1986.

[Kee89] Keene Sonya E., *Object-Oriented Programming in Common Lisp; A Programmer's Guide to CLOS*, Addison-Wesley, 1989.

[Kic&91] Kiczales G., Des Rivières J. and Bobrow D.G., *The Art of the Metaobject Protocol*, MIT Press, 1991.

[Kim&87] Kim W., Banerjee J. and Garza J.F., "Composite Object Support in an Object-Oriented Database System," in *Proc. of OOPSLA '87*, pp. 118-125, 1987.

[Kim&89] Kim W., Bertino E., Chou H., Garza J.F. and Woelk D., "Composite Objects Revisited," in *Proc. of SIGMOD '89*, pp. 337-347, 1989.

[Knu88] Knudsen J.L., "Name Collision in Multiple Classification Hierarchies," in *Proc. of ECOOP'88*, pp. 93-109, 1988.

[Kri&87] Kristensen B.B., Madsen O.L., Møller-Pedersen B. and Nygaard K., "The BETA Programming Language," In B. Shriver and P. Wegner, eds., *Research Directions in Object Oriented Programming*, pp. 7-48, MIT Press, Cambridge, Mass., 1987.

[LaL&86] LaLonde W.R., Thomas D.A. and Pugh J.R., "An Exemplar Based Smalltalk," in *Proc. of OOPSLA'86*, pp. 322-330, 1986.

[Lau&87] Laursen J. and Atkinson R., "Opus, A Smalltalk Production System," in *Proc. of OOPSLA'87*, pp. 377-387, 1987.

[Leb&91] Lebbe J. et Vignes R., "Génération de graphes d'identification à partir de descriptions de concepts," in *Induction symbolique et numérique çà partir de données*, Y. Kodratoff et E. Diday, Editions Cépaduès, 1991.

[Leb&92] Lebrun O. et Duthilleul X., *GEROME*, Version 2.0, Rapport de Projet, Ecole Universitaire D'Ingénieurs de Lille, LIFL, 1992.

[Len&86] Lenat D., Prakash M. and Shepherd M., "CYC: Using Common Sense Knowledge to Overcome Brittleness and Knowledge Acquisition Bottlemecks," AI Magazine, 6, pp. 65-85, 1986.

[Lie86a] Lieberman H., "Delegation and Inheritance: Two Mechanisms for Sharing Knowledge in Object-Oriented Systems," in J. Bézivin, P. Cointe, (Eds.), *3èmes Journées d'Etudes Langages Orientés Objets, Bigre+Globule 48*, pp. 79-89, AFCET, Paris, 1986.

[Lie86b] Lieberman H., "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems," in *Proc. of OOPSLA'86*, pp. 214-223, Portland, Oregon, 1986.

[Lip82] Lipkis T., "A KL-ONE Classifier," In *Proceedings of the 1981 KL-ONE Workshop, Flair Report No. 4*, pp. 126-143, June 1982.

[Loo&87] Loomis M.E.S., Shah A.V. and Rumbaugh J.E., "An Object Modeling Technique for Conceptual Design," In *Proc. of ECOOP'87*, pp. 325-335, 1987.

[Mae87a] Maes P., *Computational Reflection*, VUB AI-Lab TR 87-2, Ph.D. thesis V.U. Brussels, 1987.

[Mae87b] Maes P., "Concepts and experiments in Computational Reflection," *Proc. of OOPSLA '87, Sigplan Notices*, Vol. 22 No. 12, December 1987.

[Mar&90] Mariño O., Rechenmann F., and Uvietta P., "Multiple Perspectives and Classification Mechanism in Object-Oriented Representation," *Proc. of ECAI*, Stockholm, 1990.

[Mar&93] Marcaillou S., Coulette B. and Vo D.P., "An Approach to View Point Modelling," in *Proceedings of TOOLS Europe '93*, pp. 151-164, 1993.

[Mar88] Marcke K. Van, *The Use and Implementation of the Representation Language KRS*, VUB AI-Lab TR 88-2, Ph.D. thesis V.U. Brussels, 1988.

[Mar89] Mariño O., *Classification d'objets dans un modèle multi-points de vue*, DEA, Grenoble, 1989.

[Mar91] Mariño O., "Classification d'objets composites dans un système de représentation de connaissance multi-points de vue," *Actes du 8ième Congrès RFIA*, Villeurbanne, 1991.

[Mas&89] Masini G., Napoli A., Colnet D., Léonard D., et Tombre K., *Les langages à objets*, InterEditions, 1989.

[McA&86] McAllester D. and Zabih R., Boolean Classes," *Proc. of the 1st OOPSLA*, Portland, Oregon, 1986.

[McG88] Mac Gregor Robert M., "A Deductive Pattern Matcher," *Proc. AAAI-88*, pp. 403-408, St. Paul, MN, August 1988.

[McG91] Mac Gregor Robert M., "The Evolving Technology of Classification-based Knowledge Representation Systems," in *Principles of Semantic Networks, Explorations in the Representation of Knowledge*, Ed. J.F. Sowa, Morgan Kaufmann Publishers, San Mateo, California, 1991.

[Mey88] Meyer B., *Object-Oriented Software Construction,* Prentice Hall, New York, 1988, (traduction française : *Conception et programmation par objets, pour du logiciel de qualité*, InterEditions, Paris, 1990).

[Min75] Minsky M., "A Framework for Representing Knowledge," *The Psychology of Computer Vision*, McGraw Hill, Computer Sciences Series, Winston Ed., 1975.

[Min88] Minsky M., *La Société de l'Esprit*, InterEditions, Paris, 1988 (Edition originale :*TheSociety of Mind*, Simon and Schuster, New York, 1985).

[Moo86] Moon D.A., "Object-Oriented Programming with *Flavors*," in *Proc. of OOPSLA '86*, pp. 1-8, Portland, Oregon, 1986.

[Moo89] Moon D.A., "The Common Lisp Object-Oriented Programming Language Standard," in *Object-Oriented Concepts, Databases and Applications*, Eds. Kim, Lochovsky, pp. 49-78, 1989.

[Mot93] Motschnig-Pitrik R., "The Semantics of Parts Versus Aggregates in Data/ Knowledge Modelling," in *Proc. of CAISE '93*, pp. 352-373, 1993.

[Mul&93] Mulet Ph. et Cointe P., "Définition d'un noyau réflexif pour un langage à prototypes," *Actes de la 2ème Conférence sur la Représentation par Objets*, La Grande Motte, 17-18 juin 1993, EC2, Ed., pp. 101-116, 1993.

[Nap&91] Napoli A., Laurenço C. and Ducournau R., "Techniques de classification pour modéliser la synthèse de molécules organiques," In D. Hérin-Aimé, R. Dieng, J.P. Regourd and J.P. Angoujard, Eds., *Proceedings of the First International Conference on Knowledge Modeling and Expertise Transfer (KMET'91), Sophia-Antipolis*, pp. 241-252, IOS Press, Amsterdam, 1991.

[Nap&92] Napoli A. and Ducournau R., "Subsumption in Object-Based Representations," In *Proceedings of the ERCIM Workshop on Theoretical and Practical Aspects of Knowledge Representation, Pisa*, pp. 1-9, 1992.

[Nap92] Napoli A., *Représentation à objets et raisonnement par classification en intelligence artificielle*, Thèse de Doctorat d'Etat, Université de Nancy, 1992.

[Neb88] Nebel B., "Computational Complexity of terminological Reasoning in BACK," *Artificial Intelligence*, 34(3):371-383, 1988.

[Neb90] Nebel B., *Reasoning and Revision in Hybrid Representation Systems*, Lecture Notes in Artificial Intelligence, Subseries of Lecture Notes in Computer Science No. 422, Springer-Verlag, Berlin, 1990.

[Ngu&92] Nguyen G.T., Rieu D., Escamilla J., "An Object Model for Engineering Design," *Proc. of ECOOP '92*, Utrecht, 1992.

[Nic91] Nicolle A., *Les attributs et les schémas en Airelle*, Laboratoire d'Algorithmique et d'Intelligence Artificielle de Caen, 1991.

[Nic92] Nicolle A., *Etude d'une notation de domaines*, Les Cahiers du LAIAC, no. 1, Laboratoire d'Algorithmique et d'Intelligence Artificielle de Caen, 1992.

[OMS68] *Classification Internationale des Maladies ; Manuel de la Classification Statistique Internationale des Maladies, Traumatismes et Causes de Décès*, Révision 1965, Volume I, Organisation Mondiale de la Santé, Genève, 1968.

[Pae93] Paepcke A. editor, *Object-Oriented Programming: The CLOS Perspective*, The MIT Press, Cambridge, Massachusetts, , 1993.

[Pac92] Pachet F., *Représentation de connaissances par objets et règles: le système Néopus*, Thèse de Doctorat en Informatique, Université de Paris 6, 1992.

[Pan&93] Pantel M. et Sallé P., "FOL : Extensions et Formalisations des Modèles à Objets," In *Actes de la 2ème Conférence sur la Représentation par Objets*, La Grande Motte, 17-18 juin 1993, EC2, Ed., pp. 65-76, 1993.

[Pat&84] Patel-Schneider P.F., Brachman R.J., Levesque H.J., "ARGON: Knowledge Representation meets Information Retrieval," *Proc. First Conference on Artificial Intelligence Applications*, IEEE, 1984.

[Pat&92] Patil R.S., Fikes R.E., Patel-Schneider P.F., McKay D., Finin T., Gruber Th. and Neches R., "The DARPA Knowledge Sharing Effort: Progress Report," in Proc. of KR'92, pp. 777-788, 1992.

[Pat84] Patel-Schneider P.F., *Small can be Beautiful in Knowledge Representation*, Fairchild Technical Report Number 660, FLAIR Technical Report Number 37, Palo Alto, California, October 1984.

[Pat89] Patel-Schneider P., "Undecidability of Subsumption in NIKL," *Artificial Intelligence*, 39(2):263-272, 1989.

[Pat91] Patel-Schneider P.F., "What's Inheritance got to do with Knowledge Representation," in *Proc. of the Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages*, Viareggio, pp. 1-11, M. Lenzerini, D. Nardi, M. Simi, (Eds.), John Wiley & Sons, Chichester, 1991.

[Per&92] Perrot J.F. et Wolinski F., "Modélisation par objets en robotique," *Technique et Science Informatiques*, Vol. 11, No. 1, pp. 97-115, 1992.

[Pit90] Pitrat J., *Métaconnaissance Futur de l'intelligence artificielle*, Editions Hermès, Paris 1990.

[Por&89] Porcheron M., Hrynkiewicz C. et Darmois E., *ELOISE 3.0.0 Manuel de référence du chaînage avant*, Rapport LO/EL/1.0, Laboratoires de Marcoussis, Centre de Recherche de la CGE, Marcoussis, 1989.

[Rat91] Rathke C., "Implementing Frames in an Object-Oriented Programming Language," in *Proc. of the East European Conference on Object-Oriented Programming*, pp. 88-94, Bratislava, Czecho-Slovakia, 1991.

[Rat93] Rathke C., "Implementing Subsumption in an Object-Oriented Programming Language," in *Proc. of the IJCAI Workshop on Object-Based Representation Systems*, 28 aug. 1993, pp. 101-109, Chambéry, France, 1993, (également disponible comme Rapport No. 93-R-156, CRIN, Nancy).

[Rat&93] Rathke C. and Redmiles D.F., *Multiple Representation Perspectives for Supporting Explanation in Context*, Technical Report CU-CS-645-93, University of Colorado, Department of Computer Science, Boulder, Colorado, March 1993.

[Rec&85] Rechenmann F., Vignard P., *CRIKA quand les règles rencontrent les schémas*, Rapport de Recherche No. 468, INRIA Sophia Antipolis, 1985.

[Rec&90] Rechenmann F., Fontanille P., et Uvietta P., *SHIRKA: Système de gestion de bases de connaissances centrées objets*, Manuel d'utilisation, INRIA, 1990.

[Rec91] Rechenmann F., "Taxonomic reasoning in object-oriented knowledge bases," Draft, 1991.

[Res&93] Resnick L.A., Borgida A., Brachman R.J., McGuinness D.L., Patel-Schneider P.F. and Zalondek K.C., *CLASSIC Description and Reference Manual For the COMMON LISP Implementation Version 2.1*, May 15, AT&T Bell Laboratories, Murray Hill, New Jersey, 1993.

[Rie&91a] Rieu D., Culet A., "Classification et représentation d'objets," Congrès INFORSID '91, Paris.

[Rie&91b] Rieu D., Nguyen G.T., Culet A., Escamilla J., Djeraba C., "Instanciation Multiple et Classification d'Objets," *VIIèmes Journées Bases de Données Avancées*, Lyon, 1991.

[Rio&93] Rio Ph. et Vanwormhoudt G., *Modélisation en CHROME, Contexte et Hyperplan en Représentation Objet Multiple et Evolutive*, Mémoire de DEA, LIFL, 1993.

[Rob&77a] Roberts R.B., and Goldstein I.P., *The FRL Primer*, Report AIM-408, MIT, Cambridge, 1977.

[Rob&77b] Roberts R.B., and Goldstein I.P., *The FRL Manual*, Report AIM-409, MIT, Cambridge, 1977.

[Rum87] Rumbaugh J., "Relations as Semantic Constructs in an Object-Oriented Language," In *Proc. of OOPSLA '87*, pp. 466-481, ACM, 1987.

[Sak89] Sakkinen M., "Disciplined Inheritance," in *Proc. of ECOOP '89*, pp. 39-56, 1989.

[Sch&83] Schmolze J.G. and Lipkis T.A., "Classification in the KL-ONE Knowledge Representation System," in *Proc. of the 8th IJCAI*, Karlsruhe, pp. 330-332, 1983.

[Sch&91] Scholl M.H., Laasch C. and Tresch M., "Updatable Views in Object-Oriented Databases," in *Proc. of DOOD '91*, pp.189-207, 1991.

[She&93] Sheth A.P., Gala S.K. and Navathe S.B., "On Automatic Reasoning for Schema Integration," *International Journal of Intelligent and Cooperative Information Systems*, Vol. 2, No. 1, pp. 23-50, World Scientific Publishing Company, 1993.

[Shi&89] Shilling J.J. and Sweeney P.F., "Three Steps to Views: Extending the Object-Oriented Paradigm," in *Proceedings of OOPSLA '89*, pp. 353-361, 1989.

[Sny87] Snyder A., "Inheritance and the Development of Encapsulated Software Components," in *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, (Eds.), The MIT Press, Computer Systems Series, Cambridge, Mass., 1987.

[Sny91] Snyder A., "Inheritance in Object-oriented Programming Languages," in *Proc. of the Workshop on Inheritance Hierarchies in Knowledge Representation and Programming Languages*, Viareggio, 1989, pp. 153-171, M. Lenzerini, D. Nardi, M. Simi, (Eds.), John Wiley & Sons, Chichester, 1991.

[Sou&93] Souza dos Santos C. Abiteboul S. and Delobel C., "Virtual Schemas and Bases," *Neuvièmes Journées Bases de Données Avancées*, 27-30 septembre 1993, Toulouse, F. Bry (ed.), pp. 335-354, 1993.

[Sow84] Sowa J.F., *Conceptual Structures: Information Processing in Mind and Machine*, Addison-Wesley, Reading, Massachusetts, 1984.

[Ste&85] Stefik, Mark and Bobrow, Daniel G., "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, AAAI, Winter 1985.

[Ste&86] Stefik, M.J., Bobrow, D.G., Kahn, K.M., "Integrating Access-Oriented Programming into a Multi-Paradigm Environment," *IEEE Software*, 10-18, January 1986.

[Ste&88] Stein L.A., Lieberman H. and Ungar D., *A Shared View of Sharing: The Treaty of Orlando*, Technical Report No. CS-88-15, Department of Computer Science, Brown University, Providence, Rhode Island, 1988 (also in W. Kim and F. Lochovsky, editors, *Object Oriented Concepts, Databases, and Applications*, chapter 3, pp. 31-48, Addison-Wesley, 1989).

[Ste87] Stein L.A., "Delegation Is Inheritance," in *Proc. of OOPSLA'87*, pp. 138-146, 1987.

[Sti&85] Sticklen, J., Chandrasekaran, B., Smith, J.W., Svirbely, J., "MDX-MYCIN: The MDX Paradigm Applied To The MYCIN Domain," *Comp. & Maths. with Appls.*, Vol. 11, No. 5, 1985.

[Str&89] Strasser A. and Dodenhöft D., "ProObj — a Tool for Knowledge Representation," In *Proceedings of the Workshop on Foundations of Models and Languages for Data and Objects*, TU Clausthal, 1989.

[Str&90] Strasser A., Strobl G. and Dodenhöft D., "Flexible Classification in ProObj," Forschungsgruppe Künstliche Intelligenz / Intellektik, Technische Universität München, 1990.

[Ung&87] Ungar D. and Smith R.B., "Self: The Power of Simplicity," in *Proc. of OOPSLA'87*, pp. 227-241, 1987.

[Ung&91] Ungar D., Chambers C., Chang B. and Hölzle U., "Organizing Programs Without Classes," in *Lisp and Symbolic Computation: An International Journal*, 4, 3, 1991, Kluwer Academic Publishers.

[Ver93] Verdin C., *Filtrage en FROME*, Rapport de projet, Ecole Universitaire D'Ingénieurs de Lille, Département I.M.A. Option Génie Logiciel, LIFL, 1993.

[Vig84] Vignard P., *CRIQUET Un outil de base pour construire des systèmes experts*, Rapport de Recherche No. 316, INRIA, Sophia Antipolis, 1984.

[Vil85] Vilain M., "The Restricted Language Architecture of a Hybrid Representation System," *Proc. of the 9th IJCAI*, Los Angeles, California, 1985.

[Vol89] Volle Ph., "Coréférence et mécanismes déductifs dans un langage de représentation par objets," *Actes des Journées du pôle 1 : Représentation par Objets*, pp. 13-30, PRC-IA, Caen, juin 1989.

[Weg86] Wegner P. *Classification as a Paradigm for Computing*, Technical Report No. CS-86-11, Brown University, Dept. of Computer science, Providence, Rhode Island, May, 1986.

[Weg90] Wegner P., "Concepts and Paradigms of Object-Oriented Programming," *ACM OOPS Messenger* 1(1):7-87, June 1990.

[Win&87] Winston M.E., Chaffin R. and Herrmann D., "A Taxonomy of Part-Whole Relations," *Cognitive Science* Vol. 11, pp. 47-444, 1987.

[Wol&91] Wolinski F. and Perrot J.-F., "Representation of Complex Objects: Multiple Facets with Part-Whole Hierarchies," In *Proceedings of ECOOP'91*, pp. 288-306, 1991.

[Wol90] Wolinski F., *Etude des capacités de modélisation systémique des langages à objets appliquées à la représentation de robots*, Thèse de l'Université Paris VI, 1990.

[Woo75] Woods W.A., "What's in a Link: Foundations for Semantic Networks," in *Representation and Understanding; Studies in Cognitive Science*, D.G. Bobrow and A. Collins, Eds., Academic Press, New York, 1975.

[Woo91] Woods W.A., "Understanding Subsumption and Taxonomy: A Framework for Progress," in *Principles of Semantic Networks, Explorations in the Representation of Knowledge*, Ed. J.F. Sowa, Morgan Kaufmann Publishers, San Mateo, California, 1991.

[Wri&83] Wright J.M. and Fox M.S., *SRL/1.5 User Manual*, Draft of 1 December 1983, Intelligent Systems Laboratory, The Robotics Institute, Carnegie-Mellon University, Pittsburgh, USA, 1983.

[Yel92] Yelland Ph. M., "Experimental Classification Facilities for Smalltalk," in *Proceedings of OOPSLA'92*, pp. 235-246, 1992.

Index des références bibliographiques

[Duc&92b] 50
[Duc91] 19, 20, 27, 30, 34, 49, 64, 69
[Duc93] 50
[Dug88] 19, 20, 30, 47, 49, 53, 69, 70, 82, 172
[Dug91] 47, 53

E

[Esc&90] 25, 26, 172
[Euz93] 143

F

[Fah&81] 146
[Fau91] 20, 28
[Fer&88] 44
[Fer83] 58, 69, 126
[Fer88] 44
[Fer89] 20, 33, 44, 74, 77, 110, 205
[Fik&85] 17, 19, 34, 69, 146, 179
[Fin86] 146, 163, 169
[For&89] 29
[Free90] 23

G

[Gol&77] 20
[Gol&80] 19, 42, 54
[Gol&83] 19, 31
[Goo79] 24, 171
[Gra88] 152, 166
[Gri&92] 55
[Guh&90] 18
[Guh91] 44

H

[Hal&87] 33
[Ham91] 103, 126
[Han92] 23
[Har&93] 223
[Hat&91] 73, 139
[Hen86] 47
[Hol&90] 22
[Hop&93] 147, 148, 172

I

[ILO91] 29, 32, 33, 47, 69
[Inf85] 44, 69
[Int85] 20, 49, 59, 61

J

[Joh&86] 19
[Jos&87] 167

K

[Kac&86] 21, 22
[Kee89] 19, 32, 51
[Kic&91] 76
[Kim&87] 30
[Kim&89] 30
[Knu88] 50
[Kri&87] 19

L

[LaL&86] 34
[Lau&87] 73
[Leb&91] 222
[Leb&92] 84
[Len&86] 18
[Lie86a] 34, 35
[Lie86b] 34
[Lip82] 22, 23, 169, 172
[Loo&87] 25

M

[Mae87a] 19, 74, 110, 205
[Mae87b] 110
[Mar&90] 47, 142
[Mar&93] 39
[Mar88] 19, 20, 34
[Mar89] 47, 49, 142
[Mar91] 25, 27, 30, 47, 142, 143, 161
[Mas&89] 15, 17, 19, 27, 55
[McA&86] 100
[McG88] 21, 24, 163, 164, 169
[McG91] 148, 163, 164
[Mey88] 19, 50
[Min75] 17, 31, 48, 50
[Min88] 38
[Moo86] 49
[Moo89] 76
[Mot93] 30
[Mul&93] 77

N

[Nap&91] 173
[Nap&92] 170

[Nap92] 25, 30, 34, 144, 145, 170
[Neb88] 22
[Neb90] 22, 24, 145, 146
[Ngu&92] 165
[Nic91] 33
[Nic92] 20

O

[OMS68] 37

P

[Pac92] 69, 73
[Pae93] 76
[Pan&93] 23
[Pat&84] 63
[Pat&89a] 34
[Pat&92] 221
[Pat84] 22, 24
[Pat89] 22
[Per&92] 54
[Pit90] 74
[Por&89] 69

R

[Rat&93] 39
[Rat91] 76
[Rat93] 170, 171
[Rec&85] 69
[Rec&90] 19, 20, 31, 33, 59, 67, 70, 141, 158, 160
[Rec91] 159, 160
[Res&93] 31, 172
[Rie&91a] 47, 149, 150
[Rie&91b] 47, 149, 150
[Rio&93] 222
[Rob&77a] 17, 20, 34, 60
[Rob&77b] 17, 20, 31, 34, 60
[Rum87] 25

S

[Sak89] 33
[Sch&83] 145
[Sch&91] 95
[She&93] 221
[Shi&89] 39
[Sny87] 33, 50, 51
[Sny91] 33
[Sou&93] 95
[Sow84] 17, 37, 38
[Ste&85] 20, 29, 42
[Ste&86] 20, 31, 81
[Ste&88] 35
[Ste87] 35
[Sti&85] 154
[Str&89] 24
[Str&90] 24

U

[Ung&87] 34, 35
[Ung&91] 35

V

[Ver93] 129
[Vig84] 70
[Vil85] 22, 24, 163
[Vol89] 44, 49

W

[Weg86] 37
[Weg90] 17
[Win&87] 30
[Wol&91] 43
[Wol90] 30
[Woo75] 17, 43
[Woo91] 148, 163, 169
[Wri&83] 26, 31

Y

[Yel92] 170

Index des matières

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

Y