Raisonnement classificatoire dans une représentation à objets multi-points de vue

Olga Marino Drews

THÈSE

PRESENTEE PAR

Olga MARIÑO DREWS

POUR OBTENIR LE TITRE DE DOCTEUR DE

L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

(ARRETES MINISTERIELS DU 5 JUILLET 1984 ET DU 30 MARS 1992)

SPECIALITE INFORMATIQUE

==========

Raisonnement classificatoire dans une représentation à objets multi-points de vue

==========

DATE DE SOUTENANCE : 4 OCTOBRE 1993

COMPOSITION DU JURY :

M. Yves CHIARAMELLA (président)
M. Daniel KAYSER (rapporteur)
M. Amedeo NAPOLI (rapporteur)
M. Roland DUCOURNAU (examinateur)
M. François RECHENMANN (directeur)

THESE PREPAREE AU SEIN DU LABORATOIRE LIFIA/IMAG

A Pierre et à mes parents pour leur soutien et leur patience

Je tiens à remercier :

Monsieur Yves Chiaramella, Professeur à l'Université Joseph Fourier de Grenoble, de m'avoir fait l'honneur de présider le jury de cette thèse.

Monsieur Daniel Kayser, Professeur à l'Université Paris Nord pour avoir accepté d'être rapporteur de cette thèse. Je le remercie pour l'intérêt qu'il lui a portée et pour l'évaluation qu'il en a fait.

Monsieur Amedeo (pas Amadeo!) Napoli, Chargé de recherche CNRS, Docteur d'état, du CRIN à Nancy, qui a accepté d'être rapporteur de ce travail. Je le remercie aussi pour tout le temps qu'il a bien voulu lui consacrer et très particulièrement pour l'importante bibliographie qu'il m'a procurée et qui m'a permis d'avoir une vision globale de l'état de l'art.

Monsieur Roland Ducournau, Ingénieur SEMA Group, d'avoir bien voulu participer à ce jury. J'apprécie sa lecture détaillée de mon rapport et ses critiques très enrichissantes.

François Rechenmann, Directeur de recherche INRIA. Je le remercie de m'avoir acceptée dans son équipe. Je tiens à lui témoigner toute ma reconnaissance pour le soutien permanent qu'il m'a manifesté. Ses commentaires, ses critiques et surtout son encouragement dans les moments difficiles, m'ont permis de mener ce travail à son terme.

Je remercie aussi l'INRIA (France), l'Université des Andes (Bogotá-Colombie) et COLCIENCIAS (Colombie) pour le financement qu'ils m'ont accordé pour finir cette thèse.

Je tiens aussi à exprimer toute ma gratitude à :

Patrice Uvietta pour son soutien permanent. Plus que les nombreuses corrections de ma thèse, les suggestions et commentaires, je le remercie pour son amitié et ses qualités humaines qu'il met toujours au service des autres.

Jerôme Gensel, dont l'érudition en français m'a permis d'enrichir mon vocabulaire de "vieille pouille". Je n'oublierai pas son humour, ses jeux de maux, les lectures détaillées qu'il a fait de ma thèse et les pauses café..bref, merci.

Mes remerciements vont aussi aux autres membres du projet SHERPA : Claudine Mérieux et Michèle San Martin pour leur disponibilité et accueil. Jerôme Euzenat pour ses critiques toujours pertinentes ; Bruno Orsier et Mathias Chaillot qui ont adouci mes étés de travail à Grenoble ; Danielle Ziebelin, avec son sourire franc ; Nathalie Beaubouche, sa parole directe m'a toujours charmée ; Cécile Capponi et Nina Tayar pour leur soutien et amitié ; et toutes les autres personnes qui ont fait régner une très bonne ambiance dans l'équipe : Volker, Johannes, Jutta, Olivier S., Pierre F., Florence, Sophie, Christian, Olivier H., Sylvain, Thierry, François, ….

Je voudrais aussi dire un énorme merci à tous mes amis

à Pierre, bien sûr ! A Pepe, Arnie, Pascal, Carlos, Denis et Pierre qui m'ont appris le sens du mot amitié. Rubby, ma "conseillère" et ma "conscience". A toute la colonie colombo-latino qui a été ma famille ici : Alejandro, Harold, Coquita, Bruno, Rodrigo, Fernando, Claudia, Mary, Mario, Xica (!), Esperanza, Pepe2, Vicente, Germán, Ricardo, Pollo, Mabel, Yolanda, Arturo, Carmen … A eux et à tous ceux qui m'ont accompagnés pendant cette très belle période de ma vie, je les remercie pour toute l'amitié et le bonheur qu'ils m'ont offerts. A todos mil gracias.

Enfin je remercie mes amis de l'Université des Andes et ma famille pour leur soutien et leur confiance.

Résumé

Une taxinomie est une organisation de la connaissance en différentes catégories d'objets semblables. Ces catégories sont organisées dans une structure allant des catégories générales aux catégories spécifiques. Cette organisation permet de suivre un raisonnement classificatoire. Raisonner par classification consiste à trouver la catégorie la plus spécialisée à laquelle appartient un individu, puis récupérer des connaissances liées à cette localisation. Les taxinomies développées dans des domaines aussi variés que la botanique et la minéralogie montrent l'intérêt de cette approche.

Notre travail concerne le raisonnement classificatoire et la représentation taxinomique de la connaissance supportant ce raisonnement. Nous avons choisi la technique de représentation de connaissances à objets, car elle offre des éléments appropriés à une organisation taxinomique. De plus le raisonnement classificatoire trouve ici un espace naturel. Cependant, ces modèles comportent deux aspects problématiques. D'une part, ils représentent, dans une seule et grande taxinomie, différentes familles d'objets telles que "voitures" et "personnes". D'autre part, bien que les caractéristiques d'un objet correspondent à différents aspects ou points de vue, ces points de vue ne sont pas explicites dans la représentation.

Nous proposons une représentation à objets multi-points de vue, TROPES. Dans ce modèle, chaque concept ou famille d'objets a une structure taxinomique indépendante. Un concept peut être observé selon différents points de vue : un point de vue détermine un ensemble de caractéristiques du concept et une taxinomie de catégories. Les points de vue peuvent être liés par des passerelles. Par ailleurs, l'introduction des points de vue élimine les problèmes de multi-héritage d'attributs. TROPES est doté d'un algorithme de classification d'instances qui tire parti des originalités du modèle. A l'intérieur d'un concept, la classification se déroule sur un ou plusieurs points de vue et exploite les passerelles comme des raccourcis.

Mots-clés

points de vue, représentation à objets, taxinomies, classification, raisonnement classificatoire, raisonnement taxinomique, objets composites, intelligence artificielle.

Abstract

In a taxonomy, knowledge is structured in categories grouping similar objects. These categories are organized in a specialization graph, going from general to more specific classes. Taxonomies allow for classification reasoning. To reason by classification is to find the more accurate class for an instance and to infer knowledge from this localization. Taxonomies developed in areas as different as botanic and mineralogy show the interest of this approach.

Our work concerns classification reasoning and the taxonomic representation supporting it. We chose object oriented representation for it provides the elements required for modeling taxonomies. Moreover, it is a natural classification framework. However, these models present two problems: on one hand, different object families, such as "cars" and "persons", are represented in a single big taxonomy. On the other hand, although object features correspond to different aspects or viewpoints, these viewpoints are not explicitly represented in the models.

We propose an object oriented multi-perspectives representation system, TROPES. In this system, each concept or object family has an independent taxonomic structure. A concept may be observed from different viewpoints, each one focusing on a set of features and a structure of categories. Viewpoints may be linked by bridges. Furthermore, the viewpoint notion of TROPES rules out multiple inheritance problems. TROPES is provided a multi-perspectives instance classification algorithm which takes advantage of the model originalities. Inside a concept, the classification takes place on one or more viewpoints. Bridges are used to fasten the classification.

Key-words

viewpoints, object oriented representation, taxonomies, classification, classification reasoning, taxonomic reasoning, composite objects, artificial intelligence.

Table des matières

Introduction

Introduction

L'organisation de la connaissance en des catégories d'individus semblables est une activité naturelle. En effet, dès que l'enfant commence à manipuler des opérations concrètes, il groupe des objets réels dans des catégories différentes en fonction de certaines caractéristiques. Plus les caractéristiques d'une catégorie sont précises, plus cette catégorie est spécifique et ses membres semblables. Les catégories sont organisées dans une structure, allant des catégories générales aux catégories spécifiques. Les catégories spécifiques représentent des sous-ensembles des catégories plus générales. Cette structure est en général appelée taxinomie.

La grande variété de taxinomies développées dans des domaines aussi divers que la botanique, la zoologie, et la minéralogie témoignent de l'utilité d'une telle structuration des connaissances. De plus, la représentation de la connaissance en termes d'une structure de spécialisation de catégories d'objets favorise un raisonnement classificatoire.

Le raisonnement classificatoire consiste à confronter une nouvelle connaissance à un ensemble de connaissances connues pour déduire des informations liées à cette nouvelle connaissance. Le raisonnement classificatoire est un mécanisme d'inférence primordial. Face à une nouvelle situation, une personne tire partie des expériences vécues pour effectuer le choix des actions à entreprendre. Pour ce faire, elle détermine la position la plus appropriée pour cette nouvelle situation dans la structure où il mémorise celles déjà connues, puis il infère des connaissances induites par cette localisation.

Ce type de raisonnement est très souvent utilisé en résolution de problèmes : la connaissance du domaine s'exprime par une taxinomie de types de problèmes connus, une taxinomie de types de solutions et des liens heuristiques entre eux. Pour résoudre un problème, une personne le classe dans la taxinomie de problèmes, puis lui associe, par une heuristique, le type de solution le plus approprié dans la taxinomie de solutions et enfin, elle affine la solution par classification. Cette démarche de résolution de problèmes est très utilisée, bien que souvent de façon implicite, dans les problèmes de diagnostic. Par exemple, en diagnostic médical où un problème est donné par l'ensemble de symptômes d'un patient, l'heuristique est l'association entre les symptômes, la maladie et le traitement général, et la solution est alors un traitement particulier.

Parmi les différentes techniques de représentation de connaissances, les représentations de connaissances à objets (RCO) offrent les éléments nécessaires à une représentation taxinomique : elles structurent la connaissance du monde autour de deux types d'objets: les classes et les instances. Les classes représentent des catégories d'objets semblables et elles sont organisées par une relation de spécialisation dans une taxinomie. Les instances décrivent des individus, des membres des classes. Le raisonnement classificatoire trouve son «espace naturel» dans les représentations à objets. Ainsi, le mécanisme de raisonnement principal d'une telle représentation est la classification d'instances. Classer une instance consiste à trouver ses classes d'appartenance les plus spécialisées dans la taxinomie, pour ensuite inférer des connaissances liées à cette localisation.

Notre travail se place dans le cadre du raisonnement classificatoire et il concerne la classification d'instances dans une représentation de connaissances à objets. Notre contribution se situe aussi bien au niveau de la représentation à objets que du mécanisme de classification d'instances.

Au niveau de la représentation, nous allons aborder deux problèmes présents dans la plupart des systèmes de RCO :

- la structuration de toute la connaissance dans une seule taxinomie

  Une base de connaissance contient en général des familles d'objets de natures différentes et disjointes, comme "Personne" et "Voiture". Les attributs d'une famille ainsi que les classes dans lesquelles sont organisées ces instances, ne concernent pas les autres familles de la base. La description de ces différentes familles dans une même structure ne permet pas de rendre compte de ces propriétés. De plus, ceci oblige la classification à considérer la base entière alors que seule une famille d'objets l'intéresse.
- et le conflit d'héritage dû à la spécialisation multiple

  Lorsque dans une taxinomie une classe a plusieurs sur-classes et que celles-ci ont un attribut commun, le système doit décider de quelle sur-classe hériter l'attribut. Nous affirmons que la source de ce conflit est la combinaison, dans un seul graphe de classes, de plusieurs graphes de classes correspondant à diverses considération d'un même objet, des points de vue différents.

La notion de point de vue est absente de la plupart de systèmes de RCO. Pourtant les grandes bases de connaissances contiennent en général des objets concernant plusieurs disciplines. Chaque discipline observe et structure les objets selon sa perspective, son point de vue.

Comme réponse à ces besoins, nous proposons un modèle multi-points de vue, TROPES, qui partitionne la base en familles disjointes appelées concepts et qui permet une vision partielle des objets d'un concept, selon un point de vue particulier. Un point de vue détermine une taxinomie de classes et les attributs du concept qui vont être visibles sous ce point de vue. Les différents points de vue peuvent être reliés par une ou plusieurs passerelles, élément de la représentation qui établit une relation entre des classes de points de vue différents.

En ce qui concerne le mécanisme de classification d'instance nous proposons une classification multi-points de vue pour le modèle TROPES qui profite des originalités de ce modèle afin d'enrichir le résultat de la classification et tirer le plus grand nombre de déductions lors de la classification tout en minimisant le nombre de questions posées.

Le mécanisme de raisonnement que nous avons développé sur le modèle TROPES est la classification multi-points de vue d'une instance. Cette classification prend en compte les caractéristiques propres au modèle : les familles disjointes, les points de vue et les passerelles. L'espace de recherche de la classification est le concept. La classification se déroule en même temps dans les graphes de classes des différents points de vue. Enfin, les résultats obtenus dans un point de vue peuvent être utilisés par un autre point de vue grâce aux passerelles. L'algorithme de classification se base sur des principes de paramétrage, modularité, efficacité et correction. C'est, de plus, une classification incrémentale et interactive.

Dans le premier chapitre de la thèse nous donnons une vision globale des systèmes de représentation de connaissances. Après un survol de l'évolution de l'intelligence artificielle et une présentation de certaines notions de base, nous décrivons les différentes techniques de représentation et leurs mécanismes de raisonnement. Dans le deuxième chapitre, nous présentons le cadre dans lequel s'inscrit notre modèle : les représentations

à objets ayant deux types d'objets, les classes et les instances. Nous décrivons ici les différents éléments qui caractérisent une telle représentation, puis nous analysons le problème du multi-héritage dans les RCO. Enfin, nous situons ce type de systèmes par rapport aux langages à objet, aux modèles de prototypes et aux langages terminologiques. Le chapitre trois est dédié à la notion de perspectives. Après une réflexion sur les différentes interprétations que l'on peut accorder à ce terme, nous décrivons diverses façons de modéliser les perspectives dans les représentations à objets.

Le mécanisme de classification est abordé dans le chapitre quatre. Les opérations principales de tout algorithme de classification sont l'appariement et le parcours du graphe de classes. Ces procédures dépendent de la sémantique donnée aux éléments de la représentation (catégories et individus) et aux relations qui les lient. À partir de quatre paramètres: la représentation, la taxinomie ou graphe des classes, l'appariement et le parcours du graphe, nous faisons dans ce chapitre une description des différents algorithmes de classification.

Dans le chapitre cinq, nous présentons le modèle de représentation de connaissances à objets multi-points de vue, TROPES. Les différents éléments d'une base de connaissances TROPES sont décrits par leur sémantique, leur syntaxe et leurs relations avec les autres éléments de la base.

Nous décrivons l'algorithme de classification de TROPES dans le chapitre six. Le cœur de cet algorithme multi-points de vue est une boucle composée de cinq étapes. Nous donnons un aperçu général de l'algorithme et des structures d'information utilisées avant de détailler chaque étape de la boucle. La preuve de la correction de l'algorithme et l'analyse de sa complexité sont le sujet du chapitre sept. Enfin, dans le chapitre huit nous présentons trois extensions de la classification qui permettent de prendre en compte le caractère évolutif, incomplet et complexe d'une base de connaissances. Ces extensions sont la relocalisation d'une instance, la classification hypothétique et la classification d'objets composites.

Chapitre 1

Systèmes à base de connaissances

Chapitre 1

Systèmes à base de connaissances

La grande différence entre l'homme et la bête est l'intelligence. Comme le rire, l'intelligence est le propre de l'homme.

Introduction

Le terme intelligence artificielle couvre actuellement des sujets de recherche aussi variés que la robotique, le traitement de langage naturel, les réseaux neuronaux, les systèmes experts, etc. Parmi les aspects fondamentaux d'un système intelligent se trouvent la représentation de la connaissance du domaine de l'application, les techniques de raisonnement que le système utilise pour inférer des nouvelles connaissances et la communication homme-machine, en particulier pour que le raisonnement suivi soit compris par l'utilisateur.

Depuis la naissance de l'intelligence artificielle, plusieurs techniques de représentation de connaissances ainsi que des mécanismes de raisonnement associés ont été développés. Ainsi on voit apparaître les systèmes de logique classique qui raisonnent par des inférences logiques monotones, les systèmes à base de règles qui utilisent des mécanismes de chaînage avant et arrière pour simuler un raisonnement cause-effet, les réseaux sémantiques et les graphes conceptuels qui permettent de représenter la connaissance pas des graphes bipartites, les systèmes de schéma qui suivent un raisonnement par analogie et enfin les logiques terminologiques et les représentations centrées objet qui utilisent la classification comme mécanisme de raisonnement de base.

Nous commençons le chapitre par une présentation historique de la notion d'intelligence artificielle ; puis nous discutons les notions de connaissances, représentation de connaissances et raisonnement, ainsi que les différentes hypothèses que l'on peut faire pour représenter et raisonner avec des connaissances incomplètes ou inconsistantes. Dans la section trois nous traitons le sujet principal du chapitre, à savoir, les différentes techniques de représentation de connaissances, en indiquant pour chacune les mécanismes de raisonnement associés, les avantages et inconvénients, les hypothèses sous-jacentes et les applications pour lesquelles cette représentation est la plus adaptée. Finalement nous faisons une conclusion sur la problématique du choix d'un paradigme de représentation.

1.1. Intelligence artificielle

Les sources de l'intelligence artificielle remontent aux années 30 et 40 avec le développement de la logique mathématique et les systèmes logiques de Frege, Russell, Tarski, etc. En utilisant ces systèmes logiques pour formaliser le raisonnement, et des idées nouvelles sur la nature symbolique de l'information, Turing propose une conception

abstraite du traitement informatique et préconise la possibilité d'avoir un mécanisme informatique capable d'agir d'une façon qu'on qualifierait d'intelligente. Dix ans plus tard, il propose son test de Turing [TUR50] qui consiste à faire répondre une personne et un ordinateur à une série de questions posées par un observateur externe ; si celui-ci ne distingue pas les réponses de l'ordinateur de celles de l'être humain, on peut parler d'un ordinateur intelligent.

Cette définition purement comportementale (de boîte noire) de l'ordinateur intelligent est enrichie par la naissance de la **science de la connaissance** ; en effet, dans ces mêmes années 50, des travaux de Piaget, Chomsky et Minsky dans des disciplines aussi variées que la psychologie, la linguistique et l'informatique convergent vers un même but : l'étude et la compréhension de l'esprit humain. L'un des apports de cette période est le émergence de la cybernétique, science qui étudie le comportement des systèmes complexes et dont les idées d'auto-organisation et régulation partielle ont été reprises plus tard par les modèles connexionnistes [VAR89].

La science de la connaissance prend son essor avec l'invention de l'ordinateur de von Neuman dont les notions d'unité centrale de traitement, de mémoire et de canaux de communication établissent certains modèles cognitifs, comme le "general problem solver" de Newell et Simon [NEW&63] et le modèle ACT* de Anderson [AND83].

Le terme **intelligence artificielle** apparait pour la première fois en 1956 lors de la conférence de Dartmouth, organisée par Chomsky et Minsky et dans laquelle Newell et Simon présentent un premier système intelligent : "The logic theorist" [NEW&63].

Dès la naissance de l'intelligence artificielle on considère les deux branches qui dérivent de cette union de l'informatique et des sciences de la connaissance : la première branche conçoit l'IA comme "l'**étude des facultés mentales** à travers l'utilisation de modèles calculatoires" [CHA&85] ; elle se sert de l'ordinateur pour simuler et tester des processus de l'intelligence humaine. Pour la deuxième branche, **l'ordinateur lui-même est l'objet de recherche** et l'IA cherche à fournir des systèmes capables d'accomplir les tâches dites "intelligentes" :

> "Artificial Intelligence is the science of making machines do things that would require intelligence if done by humans" [MIN68] : L'intelligence artificielle est la science qui fait faire aux machines, des tâches qui auraient requis de l'intelligence si elles avaient été faites par l'homme.

Dans la suite de ce travail nous nous référons à l'intelligence artificielle sous cette deuxième acception.

Les premiers systèmes d'intelligence artificielle voient dans la vitesse de calcul et les capacités de stockage des ordinateurs une possibilité de reproduire dans un système tous les processus cognitifs de l'homme à l'aide d'algorithmes généraux. Ces systèmes généraux s'avèrent irréalisables à cause de l'énorme quantité d'information qu'ils nécessiteraient et à l'inefficacité des algorithmes généraux pour résoudre des tâches complexes particulières.

Des systèmes suivants spécialisent les mécanismes de raisonnement selon le type de problème à résoudre (diagnostic, planification, etc.) et traitent des problèmes des domaines de connaissances bien délimités (médecine, planification, mathématique) ; dans ces systèmes le raisonnement n'est pas prédéfini ; la plupart sont des systèmes déclaratifs, c'est-à-dire des systèmes qui séparent la représentation de la connaissance (tant du domaine que du problème et des stratégies de solution) de la manipulation de cette connaissance, le raisonnement.

Bien qu'un des buts principaux d'un système intelligent soit l'aide à la résolution de problèmes complexes, la solution obtenue ainsi que le raisonnement suivi doivent être explicables à l'utilisateur ; une bonne explication rend crédible la réponse, informe l'utilisateur des méthodes et des heuristiques du raisonnement employé et lui permet de tester et corriger le système. En effet, un système complexe n'est intelligent que dans la mesure où il peut expliquer son comportement à autrui.

> " Ainsi, la question de savoir si un système est de l'IA ou non, n'est pas tant qu'il modèle ou non une intelligence humaine, mais qu'une intelligence humaine puisse comprendre son comportement en termes de concepts clairement définis. [...] l'"approche I.A." est caractérisée par la prééminence de l'explicabilité sur l'efficacité". [KOD86]

Un système intelligent doit donc être capable de représenter et de résoudre des problèmes complèxes particulier, en donnant des explications claires de la démarche suivi.

1.2. Représentation et raisonnement

1.2.1. Séparation entre représentation et raisonnement

L'intelligence artificielle aborde une classe de problèmes pas encore abordés par l'informatique classique : des problèmes pour lesquels on n'a pas de méthodes de résolution prédéfinies, relevant des domaines complexes tels que la compréhension du langage, l'apprentissage, le raisonnement, la vision, etc. Pour résoudre ces problèmes l'IA propose un nouveau paradigme de programmation, la programmation déclarative, par opposition à la programmation procédurale classique.

Dans la programmation procédurale, la représentation de la connaissance et le raisonnement (c'est à dire les mécanismes d'inférence, le contrôle) sont intimement mélangés dans les instructions d'un programme. Le contrôle est présenté sous forme de procédures fixes agissant sur des données également figées. Bien que cette expression procédurale du raisonnement rende le système plus efficace et facile à expliquer, l'écriture des procédures demande une complète connaissance des données utilisées et de la façon de résoudre le problème traité, ce qui n'est pas le cas dans l'IA. De plus, la mise à jour d'un tel système est difficile, car les données et le contrôle sont contextuels et liés.

Dans les **systèmes déclaratifs**, aussi nommés "systèmes à base de connaissances", la connaissance est séparée du contrôle (Fig. 1.1). Cette séparation facilite la modification des connaissances et l'ajout de nouvelles informations à la base. Le raisonnement est entièrement dirigé par les données ; il utilise des mécanismes d'inférence qui permettent la résolution des problèmes pour lesquels il n'existe pas de procédures explicites dans le programme.

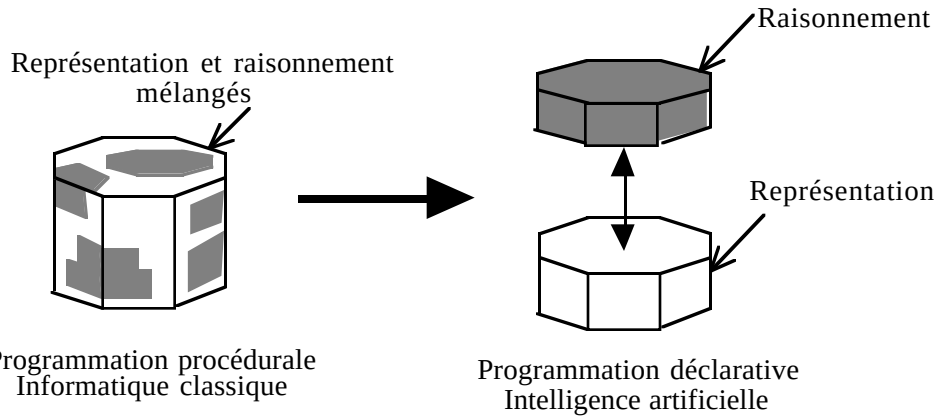


Fig. 1.1 : Un des grands apports de l'intelligence artificielle est la programmation déclarative, qui sépare la représentation de la connaissance de sa manipulation.

Un système à base de connaissances est donc un système ayant une partie représentation de connaissances et une partie raisonnement. Dans les deux sections suivantes nous présentons la signification et les propriétés des notions de connaissances et de représentation ; le problème du raisonnement est abordé dans la section (§ 1.2.4).

1.2.2. La connaissance

On peut voir la connaissance comme une "manière de comprendre et percevoir" le monde et une théorie de la connaissance comme "l'explication des rapports entre la pensée et le monde extérieur" [définition du Petit Larousse éd.1991].

Newell définit la connaissance d'un agent intelligent en fonction des buts à atteindre :

> "[Knowledge is] whatever can be ascribed to an agent, such that its behavoir can be computed according to the principle of rationality. […] principle of rationality : if an agent has knowledge that one of its actions will lead to one of its goals, then the agent will select that action" [NEW81] : [La connaissance est] tout ce qu'on peut attribuer à un agent, de façon à ce que le comportement de cet agent puisse être calculé selon le principe de rationalité.[…] Principe de rationalité : si un agent a la connaissance qu'une de ses actions peut l'amener à un de ses objectifs, alors il choisit cette action.

La connaissance peut donc être définie comme la perception et la **compréhension** qu'un **agent intelligent** a d'un **monde externe** ; cette connaissance va lui permettre d'avoir un comportement rationnel orienté vers l'obtention d'un but (Fig. 1.2).

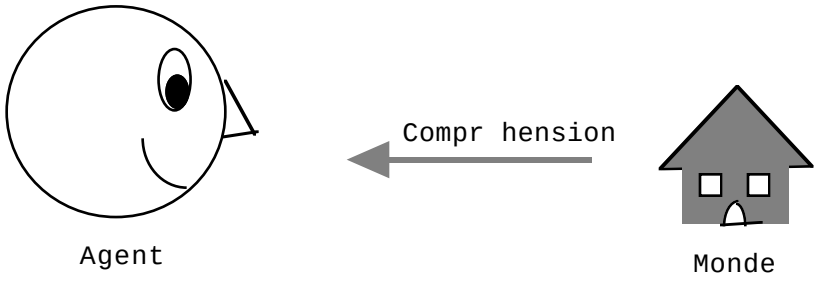


Fig. 1.2 : la connaissance est la perception et la compréhension qu'un agent intelligent a du monde.

1.2.3. La représentation de la connaissance

Pour résoudre un problème, un agent raisonne sur une abstraction de la connaissance liée au monde et à la situation du problème. Pour représenter cette connaissance, l'agent élabore un modèle des éléments du monde, leurs relations et les lois du comportement.

La représentation de la connaissance est donc la modélisation des différents éléments du monde réel et la détermination de procédures d'interprétation faisant le lien entre le monde et le modèle, tant au moment de l'acquisition de connaissances et l'élaboration du modèle, que pendant la manipulation de la représentation (pour donner des explications) et, finalement, lors de l'application des résultats du modèle au monde réel (Fig. 1.3). A partir de cette représentation et d'une capacité de raisonnement appropriée, le système doit pouvoir s'adapter et exploiter son environnement [BRA90], [BAR&81].

La connaissance représentée peut être de différents types : concept, fait, méthode, modèle, heuristique, événement, prototype, objet, etc. Elle peut avoir différentes modalités : statique ou évolutive, fixe ou modifiable, certaine ou incertaine, valide ou périmée. De plus elle peut être objective ou subjective. Par exemple, une phrase du style "je crois que…" exprime une connaissance subjective et incertaine ; quelques systèmes comme KLONE [GOO79] et OWK [SZO&77] prennent en compte ces modalités.

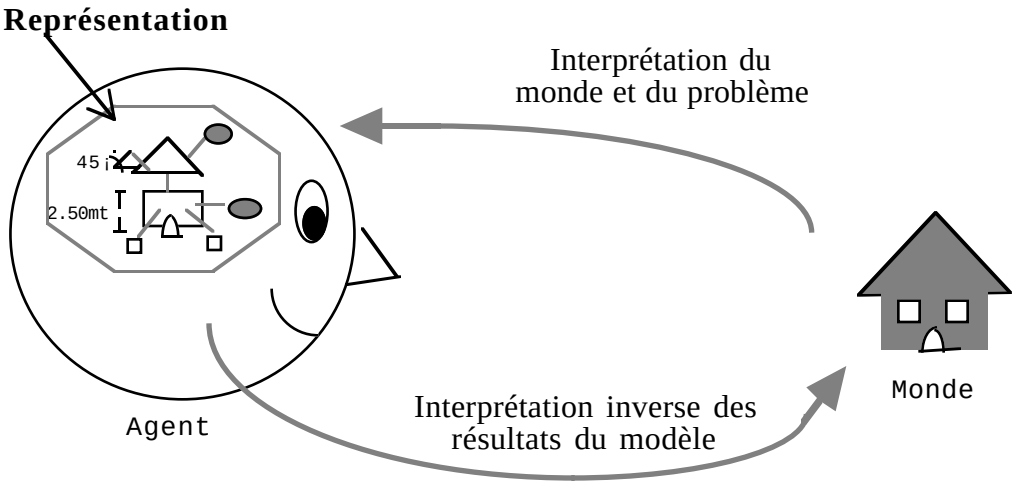


Fig. 1.3 : L'agent interprète les objets, relations et lois du monde dans une représentation ; il raisonne sur cette représentation pour résoudre un problème, puis interprète les résultats du modèle dans le monde.

Syntaxe et sémantique

L'interprétation de la connaissance pour l'élaboration d'une représentation comporte deux niveaux : le niveau syntaxique et le niveau sémantique. Le niveau syntaxique détermine le langage de description des différentes connaissances du monde : c'est à partir de ce langage que le système construit la représentation informatique (structures de données) ; le niveau sémantique donne une signification dans le monde réel aux éléments de la représentation, établissant le lien entre le monde et la représentation.

La syntaxe d'un système de représentation doit pouvoir exprimer les différents types et modalités de la connaissance du monde traité (§ 1.2.2) ; la sémantique doit faire une interprétation correcte des éléments, relations et lois du monde réel.

L'agent et le monde à représenter

La perception que l'agent peut avoir du monde se base sur l'une des quatre hypothèses suivantes :

- **le monde est unique et les agents le perçoivent tous de la même manière** : le monde externe est unique et indépendant de l'agent qui l'observe. Deux observateurs d'un même monde en ont la même perception. Ils voient tous les mêmes propriétés du même point de vue. On peut parler dans ce cas d'un monde unique et un observateur unique. Cette hypothèse est prise par la plupart de logiciels classiques et par quelques systèmes de représentation de connaissances.
- **le monde est unique et les agents le perçoivent de manière complémentaire :** comme dans le cas précédent, ce cas suppose que le monde externe est unique et indépendant de l'observateur ; chaque observateur percoit un sous-ensemble de propriétés et relations de ce monde, en fonction de ses intérêts. Même si les observateurs perçoivent le monde selon des perspectives différentes, leurs perceptions sont complémentaires, cohérentes. Dans ce courant se placent deux catégories de systèmes ; d'une part, les systèmes comme les gestionnaires des bases de données qui supposent que l'on peut faire une représentation complète du monde et qui gèrent des vues partielles du monde ; d'autre part, les systèmes de représentation de connaissances qui manipulent des objets incomplets et qui permettent des perceptions partielles de cette connaissance incomplète. [FER88], [DAV87].
- **le monde est unique, sa représentation est incomplète et il est perçu différemment par les différents agents :** le monde externe est unique et indépendant de l'agent qui l'observe. La différence avec l'approche précédente est qu'ici l'agent fait une représentation approximative du monde ; chaque agent percoit le monde à sa façon. Pour retrouver la représentation correcte il faut mettre en correspondance les représentations de tous les agents par un processus particulier de transformation. Ainsi, par exemple, dans les systèmes de robotique, les modules de perception de l'environnement peuvent arriver à des valeurs de données différentes mais proches ; des fonctions prédéfinies d'approximation calculent la bonne valeur.
- **les agents ont une vision subjective du monde :** on ne peut pas parler d'un monde unique, chaque agent percoit un monde différent. Le paradigme de cette approche, appelé "Enaction" est que le monde (l'objet) et l'entité cognitive (sujet) se définissent mutuellement [VAR89] ; les chercheurs de ce nouveau courant essaye de résoudre des problèmes pour lesquels le contexte et le sens commun jouent un rôle important.

L'incomplétude de la représentation

Le monde réel est très complexe, tant dans le nombre de propriétés et caractéristiques de ses éléments que dans la quantité d'éléments et relations existants. Pour résoudre un problème, on fait une abstraction du monde en faisant résortir les aspects et les éléments pertinents du problème et en omettant tous les autres. La représentation produite est, en général, incomplète. Pouvoir omettre des connaissances et raisonner avec une représentation incomplète est une des possibilités importantes des systèmes à base de connaissances [LEV&87].

1.2.4. Le raisonnement

Le raisonnement, tel qu'on le comprend en I.A. concerne la manipulation de la connaissance déjà acquise pour produire une nouvelle connaissance. Pour un système intelligent, le raisonnement est en général dirigé par un but que l'on veut atteindre pour résoudre un problème ; ainsi, les mécanismes de raisonnement dirigés par un but ne déduisent pas toutes les connaissances mais seulement les connaissances "intéressantes" pour le but. Différents mécanismes de raisonnement sont utilisés en I.A. pour produire des connaissances nouvelles. Ainsi, un système peut s'appouyer sur un

raisonnement logique, qualitatif, par classification, par analogie, par exemples, par contraintes, par réfutation, par induction, par abduction, etc [HAT&91].

Lorsqu'on travaille avec des connaissances cohérentes et complètes, le raisonnement consiste à rendre explicites des connaissances qui sont implicites dans la base ; c'est le cas des systèmes de logiques monotones qui raisonnent par déduction. Or, dans la plupart de situations problématiques réelles, le système doit manipuler des connaissances imprécises et incomplètes. Le raisonnement avec des connaissances imprécises utilise des techniques d'approximation comme les probabilités et les ensembles flous. Raisonner avec des connaissances incomplètes nécessite des hypothèses, c'est à dire des affirmations temporaires susceptibles de devenir fausses. On identifie deux approches du raisonnement hypothétique : le raisonnement par défaut et le raisonnement avec un ensemble de possibilités.

Le raisonnement par défaut : en l'absence d'information, la situation la plus vraisemblable est prise. La détermination de cette situation a donné lieu à différentes théories telles que la théorie du monde clos CWA [REI78], la circonscription [McCAR80], la théorie des valeurs plus probables ou défauts [REI80] et les défauts dans les taxinomies [GEN&87].

Dans la théorie de défaut de Reiter, les règles de défaut indiquent les inférence par défaut que l'on peut faire en absence d'information précise. Par exemple la règle de défaut :

$$\frac{oiseau(x) \ : \ Mvole(x)}{vole(x)}$$

dit que tout oiseau pour lequel supposer qu'il vol n'entraîne aucune insistence, vol .

Le raisonnement avec un ensemble de possibilités : cette approche, qui peut être vue comme une généralisation de l'approche précédente, ne choisit pas une valeur préférentielle mais un ensemble de valeurs possibles. Deux techniques coexistent : le raisonnement non monotone qui consiste à ajouter des hypothèses à la base de connaissances, au risque d'avoir à les remettre en cause après, et le raisonnement monotone qui crée des mondes hypothétiques en "parallèle"[COR86].

Ainsi, pour l'exemple des oiseaux, le raisonnement non monotone consiste à supposer qu'un oiseau particulier, Titi, vole : vole(Titi). Si après, le système apprend que Titi est une autruche et que les autruches ne volent pas, alors, pour résoudre l'inconsistance générée, le système enlève l'hypothèse vole(Titi).

Pour ce même exemple, le raisonnement monotone crée deux mondes hypothétiques : l'un avec l'hypothèse vole (Titi) et l'autre avec l'hypothèse inverse ¬ vole (Titi). L'ajout du fait autruche(Titi) qui entraîne ¬ vole (Titi) rend inconsistant le premier monde hypothétique, qui est alors effacé.

Bien que nettement séparés, la représentation de la connaissance et les mécanismes d'exploitation de cette connaissance sont interdépendants : un raisonnement adapté à une représentation peut s'avérer lourd et compliqué pour une autre. Par la suite nous présentons les différentes techniques de représentation de connaissances, en indiquant pour chacune ses éléments de base, les mécanismes de raisonnement les plus adaptés et ses avantages et inconvénients.

1.3. Techniques de représentation

Une technique de représentation des connaissances est une façon de représenter la connaissance dans un ordinateur pour qu'elle soit utilisée par un programme pour inférer des nouvelles connaissances. Dans le cours des 20 dernières années, l'I.A. a vu apparaître différentes techniques de représentation de connaissances tels que les prédicats logiques, les systèmes de production, les réseaux sémantiques et les schémas (frames)[1]. Les prédicats logiques et les règles de production représentent la connaissance sous forme de propositions ou affirmations sur le monde. Les réseaux sémantique, tout en gardant cette approche propositionnelle, commencent à prendre en compte la structure et les relations entre ces propositions. Enfin, les schémas pousse l'approche structurelle à fond: ils représentent le monde en termes de ses objets et leurs propriétés et relations.

1.3.1. Logique

La logique mathématique figure parmi les premiers outils utilisés par l'I.A. pour formaliser la connaissance. La logique la plus utilisée est la logique des prédicats du premier ordre ; en effet un des premiers systèmes intelligents, "The logic Theorist" (§ 1.1) est basé sur cette logique. Parmi les systèmes d'I.A. utilisant la logique comme formalisme de représentation, les plus réussis sont les démonstrateurs de théorèmes et les systèmes de planification développés sur des tels démonstrateurs (i.e. STRIPS [FIK&71]). Un des grands apports de la logique à l'I.A. est le langage PROLOG [COL&83] fondé sur la logique du premier ordre et utilisé comme langage de base de plusieurs systèmes experts développés par la suite (§ 1.3.2).

Représentation de la connaissance

Une base de connaissances en logique est constituée exclusivement d'un ensemble de **formules logiques bien formées** décrivant l'univers du discours. Ces formules ont une syntaxe précise, elles sont formées à partir de prédicats, de connecteurs logiques (et, ou, implication et négation), de quantificateurs existentiel et universel et de variables et constantes. Un prédicat correspond à une fonction donnant une valeur de vérité (vrai ou faux), les variables et les constantes sont interprétées dans des contextes spécifiques.

Par exemple, les expressions suivantes sont des formules bien formées d'un langage contenant les prédicats à un paramètre *chien* et *aboie* et la constante *Olafo* :

1) chien (Olafo)
2) $\forall$x chien(x) => aboie(x)

Mécanismes de raisonnement

Le raisonnement logique est un **raisonnement déductif** qui essaie d'expliciter les informations implicites contenues dans une base de connaissances ; vérifier si une assertion est vraie dans un univers représenté par une théorie (l'ensemble de formules de la base), consiste à vérifier si la formule qui représente l'assertion est une conséquence logique de cette théorie. Le mécanisme de déduction utilise un ensemble de règles d'inférence pour déduire des nouveaux faits à partir des faits connus. Ces règles, le **modus ponens**, le **modus tollens** et la **spécialisation universelle** sont combinées avec des manipulations syntaxiques des formules (filtrage et unification) pour élaborer la déduction [MAS&89].

La règle de spécialisation universelle dit que si une formule est vraie pour tous les individus, elle est vraie pour un individu particulier. Cette règle appliquée aux formules précédentes 1) et 2) donne 3) :

[1]Une histoire de l'évolution des techniques de représentation de connaissances est présentée dans [BRA90].

Prémisses : 1) chien (Olafo)
2) $\forall x$ chien(x) $\Rightarrow$ aboie(x)
conclusion : 3) chien (Olafo) $\Rightarrow$ aboie(Olafo)

La règle de modus ponens affirme que si un fait entraîne un deuxième et que le premier est vrai, alors le deuxième est vrai aussi. Appliquer le modus ponens aux formules 1) et 3) donne 4) :

Prémisses : 1) chien (Olafo)
3) chien (Olafo) $\Rightarrow$ aboie(Olafo)
conclusion : 4) aboie(Olafo)

Avantages

Le cadre formel de la logique mathématique permet de manipuler directement certaines notions cognitives élémentaires en leur donnant un sens précis. La logique fournit un formalisme clair et non ambigu ; cette clarté vient d'une part du fait que la signification d'une formule ne dépend que de sa structure et de la signification donnée à ses composants atomiques et d'autre part du fait que le langage d'expression logique est proche du langage naturel [HAT&91]. De plus les connecteurs logiques et les quantificateurs permettent une riche description du monde, et les inférences faites avec la logique du premier ordre sont correctes, complètes et fondées [STI&89].

Inconvénients

Malgré ses avantages, la logique présente de gros inconvénients qui ont ralenti son utilisation dans les systèmes intelligents. D'une part la représentation d'objets complexes est difficile et peu naturelle. En effet, un objet est décrit par son nom qui ne porte pas d'information sur son sens et sa structure. Les composants, propriétés et relations des objets sont, eux aussi, décrits par des noms. Ces éléments sont liés entre-eux et avec l'objet par des formules logiques. Comme l'ensemble des formules de la base n'a pas de structure, la connaissance d'un objet est disséminée dans des formules différentes, non organisées. Il y a donc une grande distance entre le modèle et les éléments du monde, qui complique l'acquisition de connaissances, la compréhension et la modification de la représentation, et la vérification de sa cohérence. D'autre part, les mécanismes de raisonnement utilisent des algorithmes généraux qui ne sont pas toujours assez efficaces pour la résolution de problèmes nécessitant un grand volume de connaissances. Enfin, la logique des prédicats du premier ordre ne permet pas de représenter des connaissances incomplètes ni des exceptions.

1.3.2. Systèmes à base de règles

Les systèmes à base de règles de production évoluent à partir des systèmes basés sur la logique. Les grands développements des systèmes à base de règles commencent dans les années 70 avec des systèmes experts comme MYCIN [SHO76], expert en diagnostic médical, et DENDRAL [BUC&78] qui étudie les structures RMN des molécules chimiques.

Représentation de la connaissance

L'élément de base des systèmes à base de règles est **la règle de production** ; une règle a la forme :

SI <condition> ALORS <action>

La partie condition est exprimée par un prédicat logique correspondant à une affirmation sur la base de connaissances qui doit être vraie au moment de valider la règle

pour que l'action soit déclenchée ; la partie action, qui est la partie exécutable de la règle indique des ajouts ou modifications à faire à la base.

Un système à base de règles comporte trois parties: une base de règles, un contexte ou base de faits et un moteur d'inférence.

La **base de règles** contient l'ensemble de règles de production du système ; cette base représente la connaissance opératoire que l'expert a du domaine du problème.

Par exemple une base de règles d'un zoo peut contenir entre autres les règles :

R1 :si ¬ attaque(x,y) ∧ ¬ attaque(y,x) alors mettre_ensemble(x,y)
R2 :si mange(x,y) alors attaque(x,y)
R3 :si mange(x,z) ∧ mange(y,z) alors attaque(x,y)
R4 :si longdent(x) ∧ mammifère(y) alors mange(x,y)[1]

Le **contexte** ou base de faits représente l'état actuel des conclusions du système, c'est à dire les faits de départ et ceux déjà inférés par le système. Un fait est une proposition vraie.

Pour la base de règles précédente, des faits de départ possibles sont

F1 :longdent(lion)
F2 :mammifère(chat)
F3 :mange (chat,souris)

Enfin, le **moteur d'inférence** contrôle l'activité du système, souvent couplé à un module qui explique le raisonnement du système. Le moteur d'inférence est séparé des bases de règles et de faits ; son fonctionnement est général et indépendant de la connaissance stockée dans les règles.

Mécanismes de raisonnement

Pour répondre à une question, le moteur d'inférence suit un cycle de détection des règles applicables, de choix de la règle à déclencher et d'exécution de l'action associée à cette règle (Fig. 1.4).

1) **Détection des règles applicables :** le moteur sélectionne toutes les règles dont la condition est vraie ; pour cela il essaie d'unifier le coté gauche de la règle avec les faits du contexte. Si le nombre de règles de la base est élevé, cette opération peut être coûteuse ; pour pallier ce problème, certains systèmes organisent les règles en paquets que le moteur considère ou ignore en fonction du contexte et d'heuristiques prédéfinies.

   Avec les règles et les faits précédents, le premier cycle du moteur d'inférence sélectionne les règles :

R2 / F3 : si mange(chat,souris) alors attaque(chat,souris) [x / chat, y / souris]
R4 / F1, F2 : si longdent(lion) ∧ mammifère(chat) alors mange(lion,chat) [x / lion, y / chat]

2) **Choix d'une règle :** si plusieurs règles applicables sont détectées, le système doit choisir laquelle appliquer. Le choix est dirigé par une stratégie du style "choisir la règle qui parle du fait le plus récemment appris" ou "choisir une règle qui ajoute un fait à la base" ou bien "la règle ayant le plus petit nombre de prémisses". Ces stratégies ne reflètent pas toujours le comportement d'un expert, et elles sont une des sources de problèmes lors de l'explication du raisonnement.

   Dans l'exemple des animaux, si le système utilise le troisième critère énoncé, il choisit la règle 2 unifiée avec le fait 3 :

[1] attaque(x,y) veut dire : x attaque y ; mange (x,y) signifie x mange y et longdent(x), x a des dents longues.

R2 / F3 : si mange(chat,souris) alors attaque(chat,souris)

3) **Exécution de l'action associée à cette règle** : exécuter l'action associée à la règle consiste à ajouter le(s) fait(s) indiqués dans la partie droite de la règle à la base de faits ; dans certains systèmes experts, qui supportent un raisonnement non monotone, l'action d'une règle peut modifier des faits déjà présents dans la base.

Après l'exécution de la règle 2, le fait attaque(chat,souris) est ajouté à la base.

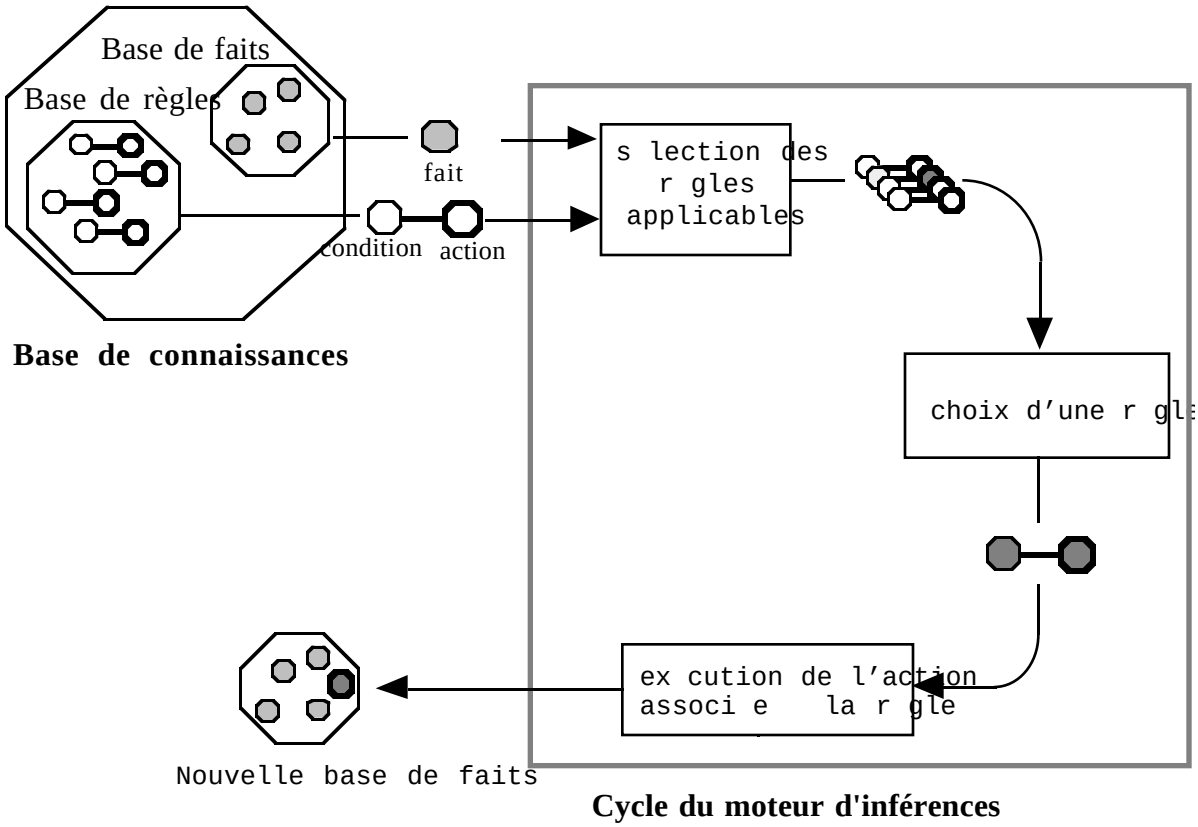


Fig. 1.4 : en chaînage avant un cycle du moteur d'inférence sélectionne les règles applicables, par unification de la condition avec les faits, choisit une règle et l'exécute en ajoutant sa partie droite à la base de faits.

Le moteur d'inférence peut fonctionner dans deux modes différents, chaînage avant et chaînage arrière. En **chaînage avant**, le raisonnement est dirigé par les données : le système part d'un ensemble de faits de départ et il enrichit cet ensemble par l'application des différentes règles possibles. Le cycle de base est celui présenté auparavant, et il s'arrête quand le système ne trouve plus aucune règle à déclencher ou lors de l'apparition de certains faits dans la base.

Le **chaînage arrière** correspond plus à l'utilisation que fait un novice ou un élève du système, par rapport au mode déductif qui est le mode de recherche d'un expert ; ici le raisonnement est dirigé par les buts : le système veut prouver un but ou un ensemble de buts donnés comme information de départ ; le cycle d'inférence est similaire à celui du chaînage avant, mais l'appariement se fait entre la partie droite de la règle et un des buts à prouver. L'exécution d'une règle, dont la conclusion (action) s'unifie avec un but, consiste à remplacer le but par la partie gauche de la règle. Le cycle termine lorsque tous les buts ont été prouvés, c'est à dire lorsque la liste de buts à prouver est vide.

Si dans l'exemple du zoo on veut savoir si le lion et le chat peuvent être mis dans la même cage, on utilise le système en chaînage arrière à partir du but : mettre_ensemble(chat,lion). Ce but s'unifie avec l'action de la règle R1 (en substituant x

par chat et y par lion), les nouveaux buts étant alors les prémisses de R1, : ¬ attaque(chat,lion) et ¬ attaque(lion,chat).

Avantages

Les systèmes à base de règles permettent en général de bien résoudre les problèmes de causalité ou de diagnostic traitant des objets simples. Ils offrent un cadre déclaratif pour exprimer des connaissances procédurales, de “savoir-faire”, ce qui permet de voir clairement les conditions dans lesquelles une règle est applicable. La connaissance est exprimée de façon uniforme par des règles et des faits ; ces règles et faits étant indépendants, la base est modulaire et facilement modifiable (voir [ROU88] pour une discussion sur le caractère relatif de cette modularité).

Inconvénients

Les systèmes à base de règles présentent le même problème que les systèmes basés sur la logique : la connaissance de la base n’est pas structurée mais plate, et l’information concernant un objet du monde est éparpillée dans les différentes règles qui parlent de cet objet. De plus, les règles expriment souvent une connaissance apparente, superficielle, susceptible d’occulter le raisonnement profond de l’expert humain [HAT&91] ; ce problème est un sujet de recherche actuel. En ce qui concerne le raisonnement, la suite d’inférences est difficile à suivre lors de la solution d’un problème et la cohérence n’est pas toujours facile à maintenir. Enfin, le système s’avère inefficace pour résoudre les problèmes ayant une séquence prédéterminée d’étapes de solution car ils n’offrent pas de moyens faciles d’intégrer des procédures complexes dans les règles [BAR&81]; la suite d’actions à prendre pour implémenter la procédure peut être décrite par des méta-règles mais cette solution est moins efficace qu’un programme classique.

1.3.3. Réseaux sémantiques

Les réseaux sémantiques, RS, ont été mis en œuvre par Quillian [QUI68] comme un modèle psychologique explicite de la mémoire associative humaine : la mémoire est vue comme un réseau d’unités d’information ; ces unités sont activées par un mécanisme qui propage des signaux à travers le réseau, la “procédure d’activation”. En I.A., le premier système à utiliser les techniques des RS, SIR [RAP68], répond à des questions qui demandent un raisonnement simple, avec des prédicats binaires ; dans les années 70 plusieurs systèmes de compréhension du langage utilisent les réseaux sémantiques, de même que quelques logiciels éducatifs, comme SCHOLAR [CAR&73]. Plusieurs variétés de réseaux sémantiques émergent à la fin des années 70 : les réseaux partitionnés de Hendrix [HEN79] qui divisent le réseau en sous-réseaux et qui permettent de manipuler ainsi des assertions, comme la logique; les réseaux de propagation de marqueurs [FAH79] qui font une propagation parallèle des étiquettes par un contrôleur centralisé ; les hiérarchies de thèmes [SCH&79] qui structurent les propositions dans des hiérarchies selon les sujets qu’elles traitent (coloris, temps, taille, etc) ; les réseaux procéduraux [LEV&79] issues des idées des représentations procédurales de Winograd [WIN72], etc.

Représentation de la connaissance

Un réseau sémantique est un **graphe** composé d’un ensemble de **nœuds** qui représentent des concepts d’entité, attribut, objet, événement, état, etc. et d’un ensemble d’**arcs** orientés, étiquetés, liant deux nœuds, qui représentent des relations binaires entre ces concepts ; les étiquettes dans les arcs spécifient le type de la relation modélisée. Quelques relations représentent des cas linguistiques, comme “agent”, “patient” et “récepteur” ; d’autres, des connecteurs de causalité, spatiaux ou temporaires, et d’autres

encore spécifient le rôle qu'une entité peut jouer : "mère", "propriétaire", etc. Une relation du réseau peut avoir des propriétés particulières telles que la symétrie, la transitivité, une relation inverse, etc.

Deux relations spécialement importantes, pour la possibilité qu'elles offrent de structurer la connaissance en une hiérarchie de concepts, sont la relation sorte-de (entre concepts génériques) et la relation est-un (entre un concept générique et un concept individuel). Ces deux relations sont souvent confondues dans un seul lien de spécialisation de concepts (Fig 1.5)[1]. De plus, un concept qui spécialise un autre possède toutes les relations de celui-ci, il hérite de son information. Cette idée de structurer la connaissance en une ou plusieurs hiérarchie, proposée pour la première fois par les réseaux sémantiques, est à l'origine de toute une famille de systèmes, les systèmes hiérarchiques et en particulier les logiques terminologiques (§ 1.3.5).

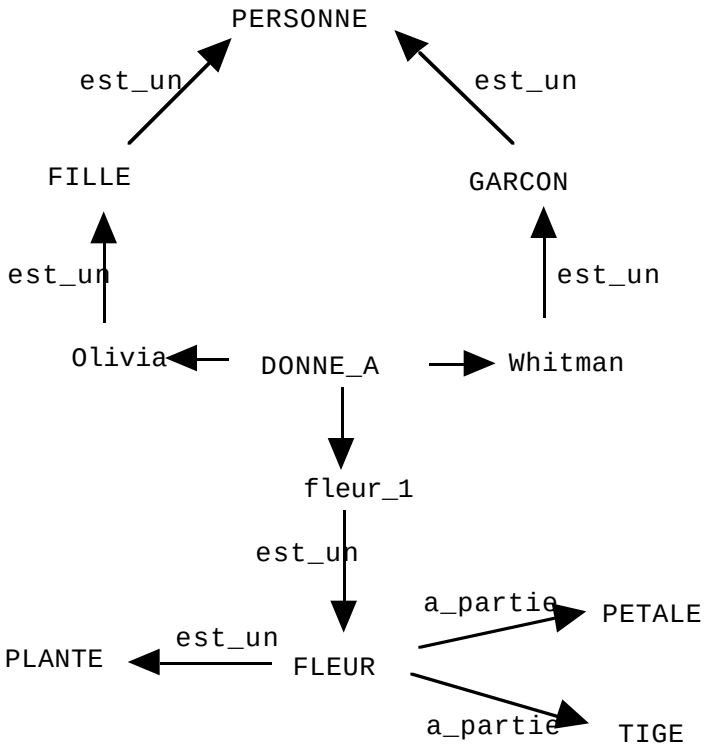


Fig. 1.5 : Réseau représentant quelques liens entre le proposition "Olivia donne une fleur à Whitman" et des connaissances conceptuelles générales sur les êtres humains et les fleurs (traduit de [STI&89]). Les relations non binaires, comme DONNE sont représentées par un nœud et non pas par un arc.

Mécanismes de raisonnement

La "procédure de propagation d'activation", proposée initialement par Quillian pour raisonner avec un réseau sémantique, cherche à établir les liens existant entre deux concepts que l'agent veut mettre en correspondance. Le point de départ de la méthode est les nœuds de deux concepts en question ; la méthode "active" tous les nœuds connectés à chacun d'eux, puis les nœuds connectés aux nœuds qui viennent d'être activés et ainsi de suite. Une connexion s'établit quand un nœud est activé par deux liens différents ; à ce moment le système reconstruit le chemin d'activation pour lier les deux nœuds initiaux. La complexité d'établir une relation entre deux nœuds du réseau est proportionnelle à la distance (nombre de liens) entre ces deux nœuds. Cette notion "spatiale" de distance

[1]Les différentes sémantiques données aux liens dans les réseaux sémantiques sont discutées dans [BRA83] et [WOO75].

entre connaissances apparait ici pour la première fois dans une représentation propositionnelle.

Bien que ce mécanisme initial d'inférence, fortement inspiré des idées en psychologie, soit encore utilisé pour le parcours du thesaurus dans certains systèmes de recherche d'information [COH&84], la plupart des réseaux sémantiques actuels favorisent le mécanisme de **filtrage**. Le filtrage consiste à parcourir le graphe et à chercher tous les sous-graphes du graphe ayant des propriétés ou une structure commune avec un graphe cible ; cette recherche correspond à un appariement de graphes. Ainsi, par exemple, une interrogation est modélisée par un sous-réseau où les nœuds représentant l'information recherchée sont étiquetés par des variables qu'une procédure de filtrage tente de lier avec des nœuds du réseau traité [MAS&89]. Par exemple, la question *"Qu'est-ce qu'Olivia donne à Whitman"* est représentée par un réseau (Fig. 1.6), qui est ensuite unifié avec le réseau général (Fig. 1.5). La réponse à la requête est la substitution faite pour les variables du graphe au moment de l'appariement ; dans l'exemple, la variable *?chose* est unifiée avec le sous-graphe contenant comme nœud principal le nœud *fleur_1*.

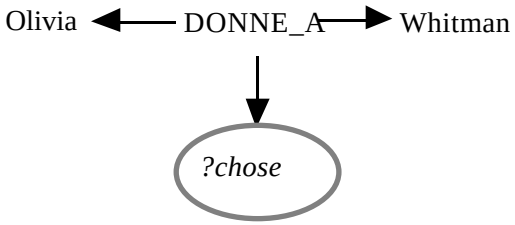


Fig. 1.6 : requête à la base : "Qu'est-ce qu'Olivia donne à Whitman" ; le nœud *?chose* est une variable qui doit être instanciée lors de l'unification de ce sous-réseau avec le réseau général (Fig. 1.5).

Le filtrage peut faire appel à l'héritage : celui-ci consiste à récupérer des informations des nœuds représentant des concepts plus généraux, pour les utiliser dans des nœuds plus spécialisés ; cette récupération se fait en suivant les liens de spécialisation est-un. Dans l'exemple précédent, on peut modifier l'interrogation pour demander "Donne Olivia quelque chose à une personne?". Le graphe de cette requête (Fig. 1.7) ne peut pas être directement unifié avec le réseau général. Une étape d'inférence par héritage est nécessaire pour déduire que Whitman peut se substituer à la variable "une personne" car il est un garçon et tout garçon est une personne ; la réponse à la requête est donc VRAIE.

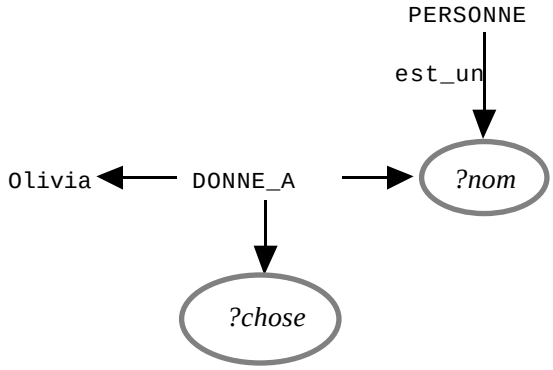


Fig. 1.7 : La requête "Est-ce que Olivia donne quelque chose à un homme?" est résolue par une combinaison de filtrage et d'héritage.

Avantages

Les réseaux sémantiques, tout en restant propositionnels comme la logique et les systèmes à base de règles, reconnaissent déjà l'importance de la structure ; ils structurent

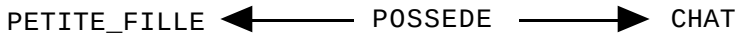
la connaissance en nœuds et arcs. Cette représentation rend visibles les diverses relations existantes entre les objets, ainsi que la notion de “distance” entre deux concepts (nombre de liens du chemin connectant les deux concepts). Ces modèles sont bien adaptés aux domaines où les concepts sont simples et fortement liés entre eux, comme les phrases en langage naturel. Sur cette représentation sous forme de graphe, le mécanisme de filtrage permet de récupérer des informations explicites ou implicites de la base à la manière des associations mentales de l’être humain.

A l’origine, les réseaux sémantiques n’ont pas une sémantique formelle bien définie. Le sens donné aux liens dépend de l’application et n’est en général pas reconnus par le système. Ce problème est traité par WOODS dans son article “What’s in a link” [WOO75] ; peu après, Ronald Brachman présente son système KL-ONE [BRA&85], précurseur des langages terminologiques (§ 1.3.5), qui comporte une syntaxe précise et une sémantique formelle basée sur la logique et la relation de subsomption entre concepts ; par ailleurs, les graphes conceptuels de SOWA (§ 1.3.6) proposent un isomorphisme entre la logique du premier ordre et une représentation par des graphes.

Inconvénients

Un premier inconvénient des réseaux sémantiques est le problème de granularité et du choix des nœuds et des liens de base. Bien que certaines relations sont traitées pour elles-mêmes (comme la relation de spécialisation sorte-de) , la plupart sont traitées de façon générale sans tenir compte de leur sémantique. De plus, le concepteur du réseau a dû mal à déterminer le niveau de détail et les structures générales nécessaires à la bonne expression d’une proposition. Par exemple, combien de liens sont importants pour saisir le sens de la phrase “Olivia donne une fleur à Whitman”? (Fig. 1.5).

Par ailleurs, la seule information attachée à un nœud est son nom, donné par une étiquette. Cette simplification pose plusieurs problèmes. D’une part, il est difficile d’exprimer des propositions ayant des quantificateurs universels, existentiels et numériques, très courants en traitement de langage naturel. Ainsi le sous-graphe



peut représenter des propositions différentes telles que : “toute petite fille possède un chat”,“quelques petites filles possèdent un chat”, “quelques petites filles possèdent quelques chats”, etc. Ce problème est traité dans les graphes conceptuels de Sowa [SOW91], qui expriment les quantificateurs de la logique à l’aide du lambda calcul.

D’autre part, la sémantique du lien d’appartenance est-un entre un individu et un concept n’est pas très bien définie : que veut dire “Olivia est-un fille”? ; être fille indique une relation établie par l’age, la taille, l’apparence, les chromosomes,…?[STI&89], [DES86].

De plus, les réseaux sémantiques offrent une représentation statique du monde, ce qui rend difficile la modélisation de l’évolution de l’information (par ex : lorsque Olivia passe du statut de fille à celui de femme). Finalement, cette technique, très utilisée dans des applications de compréhension de la langue naturelle, pose des problèmes pour des applications manipulant des objets complexes et voulant raisonner sur l’objet complet ou l’un de ses parties.

1.3.4. Schémas

Les techniques présentées auparavant, la logique, les règles de production et les réseaux sémantiques, représentent la connaissance sous forme propositionnelle. Par opposition à cette forme peu structurée de la connaissance, les modèles structurels, comme les schémas et les représentations centrées objet, regroupent des connaissances d'une situation, d'un événement ou d'un objet du monde dans des paquets ("chunks") accessibles en tant qu'unités. L'idée d'empaqueter des connaissances ("chunking") apparaît en psychologie dans les travaux de Bartlett [BAR64]. Bartlett soutient que dans son activité cognitive quotidienne, l'homme utilise des paquets de connaissances représentant des expériences précédentes pour interpréter une nouvelle connaissance. Il choisit la structure la plus proche de la situation courante et l'adapte pour la faire correspondre à la nouvelle situation.

En 1973, Schank prend les idées de paquets de connaissances de Bartlett pour construire des modèles de représentation du langage naturel. Peu après, en 1975, Minsky propose le premier formalisme informatique utilisant ces mêmes idées. L'unité de représentation des connaissances proposée par Minsky est le **schéma** ("**frame**"), structure dynamique représentant des situations prototypiques comme par exemple, aller à un restaurant ou acheter un cadeau. Le schéma contient les informations sur la situation typique, ce qui peut arriver, les actions à entreprendre dans chaque cas, etc. En tant que prototype de tout un ensemble de situations, le schéma a des informations générales valides pour toutes les situations possibles et des informations spécifiques à chaque situation pour lesquelles il a une valeur par défaut, la valeur la plus probable ; cette valeur par défaut peut être remplacée par une valeur spécifique pour une situation particulière [MIN75].

Dans le cadre de cette théorie, la rapidité des activités mentales humaines s'explique par le fait qu'une situation nouvelle est identifiée dans la mémoire avec la situation prototypique la plus appropriée ; les informations générales de ce prototype sont ensuite récupérées et éventuellement modifiées pour créer la représentation précise de la nouvelle situation.

A un niveau plus élevé, des schémas logiquement liés (par des relations cause-effet, des relations temporelles, des actions du monde, etc) sont groupés en "systèmes de schémas" ; à l'intérieur du système, un schéma peut être transformé en un autre par une fonction de transformation qui copie la relation représentée par le système (Fig. 1.8). Ainsi, en vision, un système de schémas décrivant une scène (par exemple un cube) selon des points de vue (schémas) spatiaux différents ; une transformation spatiale permet de passer d'un point de vue à un autre[1].

La théorie des schémas de Minsky a donné naissance à un grand nombre de systèmes. Tous ces systèmes utilisent le schéma comme structure de regroupement de connaissances et ils raisonnent par appariement entre une situation particulière et un schéma connu. Malgré ces similitudes de base, la plupart de ces systèmes proposent des approches différentes pour des aspects tels que le traitement des valeurs par défaut, les "systèmes de schémas", la façon de faire l'appariement et même la signification liée aux schémas.

Ce dernier aspect a donné lieu à deux types de représentations : la représentation par prototype et la représentation par catégories ou définitionnelle. Dans l'approche **prototypique** tout schéma représente une situation ou objet prototypique d'une catégorie générale implicite (par exemple le schéma de "Pelé" décrit un prototype de "joueur de football ideal") ; cette approche, tout en étant la plus fidèle aux idées de Minsky, n'est pas la plus répandue [LIE86]. Représenter la connaissance générique d'un domaine

[1] Cette notion de points de vue des schémas de Minsky est traitée plus en détail dans la section § 3.1.

d'application par des prototypes n'est pas toujours naturel ; de plus le raisonnement avec une structure de prototypes pose de graves problèmes (§ 2.6.2). Par opposition à l'approche prototypique, l'approche **définitionnelle** distingue deux types de schémas : les schémas de classe et les schémas d'instance. Les schémas de classe représentent une catégorie d'objets du monde (par exemple les "joueurs de football ideaux") ; les schémas d'instance représentent des individus particuliers (par exemple "Pele" mais aussi "Papin" et "Beckenbauer") ; à cette famille appartiennent des systèmes comme KRL [BOB&77] qui a les unités de base, abstraites et de spécialisation d'un côté et les unités individuelles de l'autre ; FRL [ROB&77] avec les unités génériques et individuelles et SHIRKA [REC88] qui distingue les schémas de classe et les schémas d'instance. Enfin dans quelques systèmes un schéma comporte une partie prototypique et une partie générique.

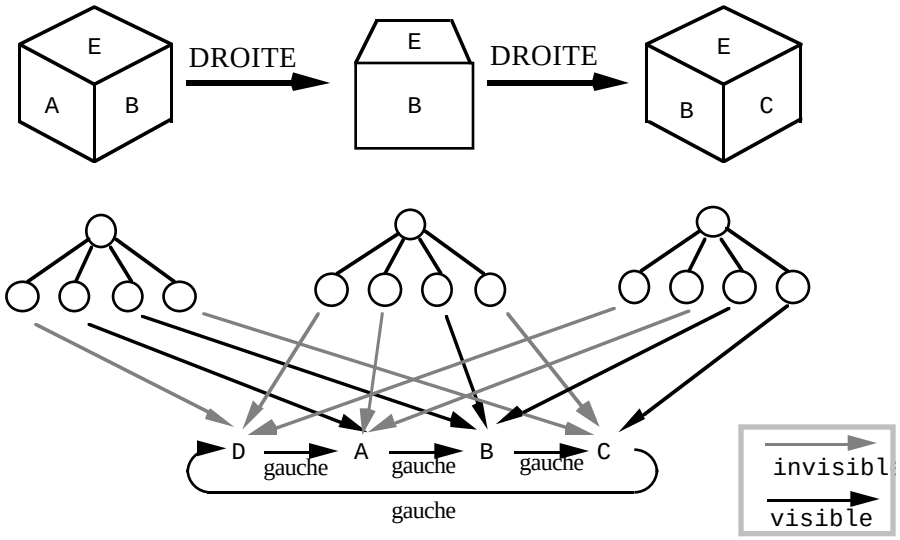


Fig. 1.8 : Système de schémas d'un cube regardé de trois points de vue différents ; un schéma est lié à un autre par une transformation spatiale ; exemple tiré de [MIN75].

Nous allons centrer l'exposé sur l'approche définitionnelle, qui est la plus répandue et la plus proche de notre travail.

Représentation de la connaissance

L'unité de représentation de la connaissance est une structure à trois niveaux : **schéma-attribut-facette** (Fig. 1.9) ; le **schéma** est composé d'une collection d'attributs. Un **attribut** (**"slot"**) représente une propriété ou relation de l'objet représenté par le schéma ; il est décrit en termes de descripteurs ou facettes. Une **facette** peut être déclarative (statique) ou procédurale (dynamique) ; une facette déclarative établit une contrainte pour la valeur de l'attribut (de type ou domaine des valeurs possibles) ou bien la valeur sure (toujours valide) ou par défaut (la plus probable) pour cet attribut. Une facette procédurale fait appel à une méthode de calcul (une procédure) ; cette méthode peut servir à calculer la valeur de l'attribut (**attachement procédural pour un calcul**) ou bien à des opérations de vérification et mise à jour de la base lors d'une modification de l'attribut (**réflexe**).

Pour les schémas d'instance, la seule facette associée à un attribut est la valeur de cet attribut pour l'instance ; cette valeur peut être une valeur simple (si l'attribut est mono-valué) ou une liste de valeurs (s'il est multi-valué) ; chacune de ses valeurs (un seul pour les attributs mono-valués) peut être une constante ou bien une autre instance de la base.

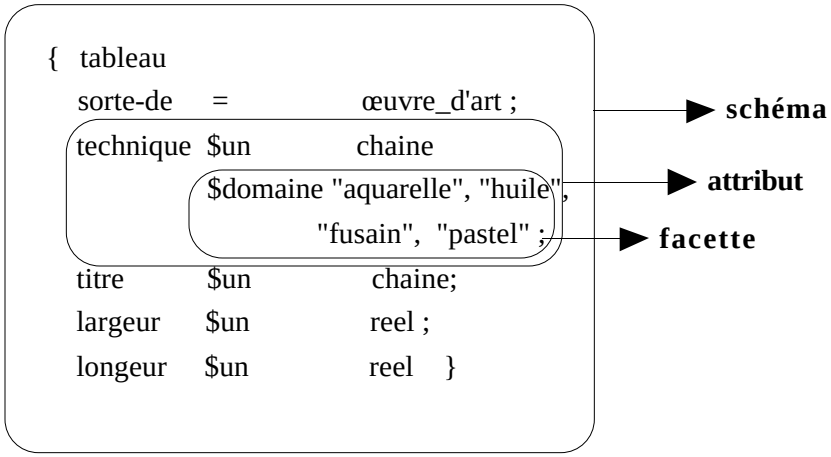


Fig. 1.9 : structure schéma-attribut-facette d'une classe SHIRKA

La théorie de Minsky propose la structuration des schémas dans des systèmes de schémas ; un schéma d'un système de schémas est lié à un autre schéma du même système par une relation propre au système. A part quelques exceptions comme le système Views [DAV87], les représentations par schémas structurent les schémas selon la relation hiérarchique de spécialisation, dans un seul système de schémas. La relation de **spécialisation** (décrite par le lien sorte-de) lie un schéma de classe (dit sous-classe) à un autre schéma de classe (dit sur-classe) et représente l'inclusion ensembliste des ensembles d'individus décrits par ces classes ; la structure de schémas de classes induite par la relation de spécialisation est appelée taxinomie. A côté de la relation de spécialisation entre classes, la relation d'**instanciation** (décrite par le lien est-un) lie un schéma d'instance au schéma de classe représentant la classe d'appartenance de cette instance (Fig. 1.10).

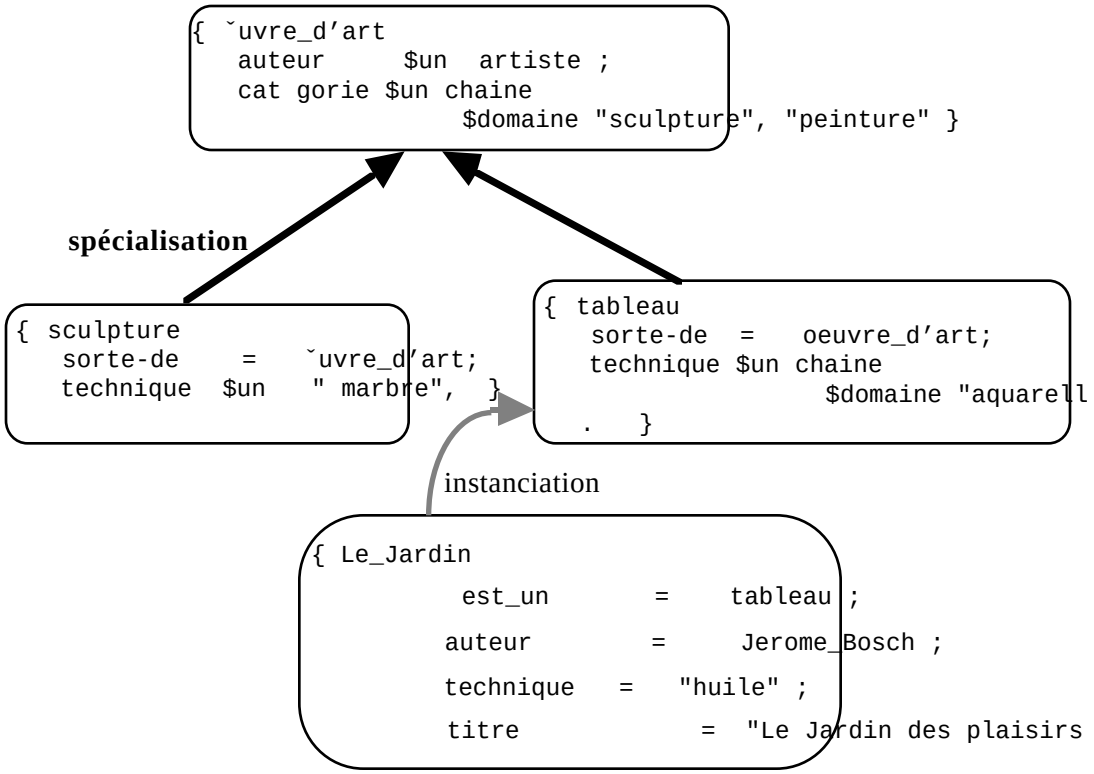


Fig. 1.10 : relations d'instanciation et de spécialisation. L'attribut est-un établit le lien entre l'instance et sa classe, l'attribut sorte-de entre une classe et sa sur-classe. L'instance "Le_Jardin" est une instance incomplète : les attributs largeur et longueur (Fig. 1.9) n'y sont pas valués.

Mécanismes de raisonnement

Les représentations par schémas offrent deux types de mécanismes de raisonnement : les mécanismes globaux qui travaillent sur toute la base et les mécanismes locaux qui font des inférences au niveau d'un schéma ou d'un attribut d'un schéma. Les mécanismes globaux sont l'héritage, le filtrage et la classification et les mécanismes locaux sont les réflexes de contrôle (sib-exec, si-modif, etc.) et les réflexes de calcul.

Mécanismes globaux :

L'héritage

La relation de spécialisation représente l'inclusion ensembliste ; cela veut dire que toutes les instances d'une classe le sont aussi de ses sur-classes ; à ce titre, elles ont les propriétés décrites dans les sur-classes. Une classe doit donc pouvoir "récupérer" l'information de ses sur-classes. Le mécanisme d'héritage de propriétés permet la récupération de cette information à travers les liens de spécialisation et évite ainsi d'avoir à recopier les attributs des sur-classes dans la sous-classe. L'héritage est dynamique, c'est-à-dire, l'information héritée d'une classe n'est pas stockée dans la sous-classe mais récupéré chaque fois que le système accède le sous-classe. Cela garantit que toute modification faite à une classe est prise en compte par ses sous-classes. Bien que cette dernière propriété facilite le contrôle de la cohérence de la base et donc le raisonnement, le mécanisme d'héritage est plus un raccourci d'écriture (car on n'a pas à recopier des informations) qu'un réel mécanisme d'inférence de nouvelles connaissances.

Dans la base d'œuvres d'art présentée précédemment, l'attribut "auteur" défini dans la classe racine, est hérité automatiquement par ses sous-classes : "sculpture" et "tableau". Ainsi une instance de "tableau" a les attributs "technique" et "titre", définis dans sa classe d'appartenance, mais aussi l'attribut "auteur", hérité de la classe "œuvre-d'art" .

Le filtrage

Le mécanisme de filtrage recherche les schémas satisfaisant certaines caractéristiques données ; ces caractéristiques sont décrites dans un schéma appelé **filtre**, qui est apparié avec des schémas de la base. Le filtrage se distingue d'une requête classique de bases de données par plusieurs aspects. D'une part, les instances d'une base de connaissances pouvant être incomplètes, le résultat du filtrage n'est pas une liste fixe d'instances mais une liste de schémas d'instances où chaque schéma peut être complété de différentes façons pour produire diverses instances. D'autre part, grâce à la structure hiérarchique des schémas de classes, l'ensemble de schémas d'instances pouvant répondre au filtrage est vite réduit à ceux dont les classes d'appartenance ne sont pas en contradiction avec les contraintes du filtre. Enfin, l'étape d'appariement entre le filtre et un schéma cible peut faire appel aux mécanismes de raisonnement de bas niveau comme les réflexes et l'attachement procédural pour un calcul. Dans certains systèmes comme SHIRKA, le filtrage peut être un mécanisme local, une facette d'un attribut d'une classe (Fig. 1.11). La valeur de cet attribut, pour une instance particulière de la classe, est la liste d'instances qui satisfont le filtre.

```
{ artiste
     sorte-de     =    personne ;
     lui-meme   $var-nom   lui ;
     ……
     œuvres      $liste-de     chaine
                  $sib-filtre { œuvre d'art
                                  auteur    $var<-  lui ;
                                  titre       $var->  œuvres }}
```

Fig. 1.11 : la valeur de l'attribut "œuvre" d'un artiste contient les titres de ses œuvres d'art ; cette liste est obtenue en filtrant parmi toutes les œuvres d'art celles ayant dans l'attribut "auteur" une référence à cet artiste.

La classification

La classification consiste à positionner un nouveau schéma dans une hiérarchie de schémas connue. En général les systèmes de schémas distinguent la classification de classes de la classification d'instances. La classification de classes modifie les liens taxinomiques des classes et constitue un mécanisme de gestion et maintien de la base ; la classification d'instances est un mécanisme de raisonnement qui permet de compléter la connaissance d'une nouvelle instance en la plaçant correctement dans la base et en récupérant l'information déduite de ce classement. Le mécanisme de classification d'instances correspond à une stratégie de résolution de problèmes spécialement utilisée en diagnostic : face à une nouvelle situation, l'être humain la classe dans sa taxinomie de classes de situations connues, puis il récupère des informations générales propres aux instances de cette classe, et enfin il les spécialise pour les faire correspondre au cas courant. Les modèles de schémas sont spécialement utiles pour la classification, car la description des classes fournit un moyen déclaratif de spécifier les critères d'appartenance à une classe, et la structure taxinomique des classes facilite la classification graduelle [FIK&85].

La classification d'instances est le mécanisme de raisonnement de base de notre travail et sera traité plus en détail à partir du chapitre 4.

Mécanismes locaux

Réflexe de calcul

La possibilité d'attacher des méthodes de calcul aux attributs est l'une des caractéristiques les plus originales des schémas. Cette caractéristique, aussi appelé attachement procédural pour un calcul consiste à attacher à un attribut d'une classe, sous la forme d'une facette, une méthode de calcul (en général cette méthode fait appel à une procédure du langage d'implémentation). Lors d'une consultation à une instance de la classe, si la valeur de cet attribut n'est pas connue, le système déclenche la méthode de calcul avec les paramètres propres à l'instance et obtient, comme réponse de la méthode, la valeur recherchée. Si la méthode échoue ou s'il n'y a pas de réflexe de calcul pour l'attribut en question, le système donne à l'instance la valeur par défaut de cet attribut.

Bien que très utile, l'attachement procédural pour un calcul ne peut être utilisé par le mécanisme de classification d'instances ; en effet, ce mécanisme, attaché à un attribut d'une classe, ne peut être déclenché que pour les instances de cette classe ; il ne peut pas être utilisé lors de la validation de l'appartenance d'une instance à la classe.

Réflexe de contrôle

Les réflexes de contrôle (appelés aussi démons) sont des fonctions qui s'activent automatiquement lors des accès aux attributs des instances (programmation dirigée par les accès [STE&86]). De même que les réflexes de calcul, les réflexes de contrôle sont décrits comme des facettes des attributs qui les activent ; ils servent à vérifier la validité des actions sur ces attributs et à déclencher d'autres actions. Les réflexes de vérification (si-possible ou à-vérifier) permettent de vérifier qu'une modification d'un attribut est valide ; la réussite ou l'échec d'un calcul d'un attribut peuvent déclencher des actions attachées aux réflexes si-succes et si-echec ; enfin, après la modification d'un attribut les réflexes si-ajout et si-supprime déclenchent des actions de maintien de la cohérence de la base ou des appels aux mécanismes globaux pour poursuivre la solution d'un problème global.

Avantages

Les systèmes de schémas sont très utiles pour modéliser les domaines où les objets sont complexes et riches ; le regroupement de toutes les informations d'un objet dans une unité fournit une représentation structurelle concise et facilement exploitable. La structuration de la connaissance en de telles unités et les liens entre ces unités facilite la manipulation des connaissances.

Dans l'approche définitionnelle, la relation de spécialisation entre classes organise la connaissance par niveaux de généralité permettant un partage optimale des connaissances. De plus, la structure en classes et instances peut tirer parti des mécanismes d'inférence puissants comme le filtrage et la classification. Cette dernière facilite la mise en correspondance entre une nouvelle situation et les situations connues. Finalement l'héritage réduit le travail de description du concepteur et facilite le maintien de la cohérence de la base.

Inconvénients

La plupart des systèmes de schémas manquent d'un formalisme de base ; ce manque de formalisme entraîne un manque de rigueur dans l'utilisation des différentes notions ; ainsi par exemple, l'attachement procédural pour un calcul est très souvent utilisé lors de la classification, et la facette défaut qui représente une valeur incertaine mais probable est en général utilisée pour faire des inférences certaines dans un contexte monotone. De plus, quelques systèmes introduisent des liens d'exception qui ne sont pas toujours bien gérés par les mécanismes de raisonnement. Par ailleurs, les systèmes de schémas ne font pas la distinction entre les caractéristiques définitionnelles d'une classe : celles qui doivent être satisfaites par les instances pour appartenir à la classe, et les caractéristiques déduites : celles qui sont vraies pour les instances de la classe [BRA83] (cette distinction est un des grands apports des langages terminologiques).

L'analyse précédente des avantages et inconvénients des schémas concerne principalement l'approche définitionnelle. Dans l'approche par prototype l'avantage principal est son utilisation dans les domaines où l'identification des classes est difficile et où les experts raisonnent en termes d'éléments typiques. En ce qui concerne les inconvénients, les prototypes présentent les mêmes problèmes de manque de rigueur énoncés pour l'approche définitionnelle. Mais leur problème principal est que le graphe de prototypes, induit par la relation de "vraisemblance" entre prototypes est construit en fonction de l'ordre d'appréhension des prototypes. La validité du raisonnement classificatoire dépend de la discipline de représentation du concepteur de la base car le système ne peut pas utiliser la relation entre prototypes pour affiner la classification d'un objet.(§ 2.6.2) [BRA85] .

1.3.5. Logiques terminologiques

Les logiques terminologiques[1] ont pour base les schémas et les réseaux sémantiques. Ces systèmes s'appuient sur un modèle logique pour bâtir des représentations autour de la notion de concept structuré et pour organiser des concepts dans une hiérarchie de subsomption ; ils manipulent et mettent à jour la hiérarchie de concepts par un algorithme de classification de concepts, le "classifieur". Le précurseur de cette famille est le système KL-ONE développé par Brachman en 1978 [BRA&85] et complété par un mécanisme de classification et une sémantique formelle en 1983 [SCH&83]. Le souci de complétude d'une part et d'efficacité du raisonnement de l'autre ont motivé le développement de toute une famille de systèmes : certains ajoutent des éléments à la représentation pour augmenter l'expressivité, comme KRYPTON qui introduit les ABOXs ou paquets d'assertions, et LOOM [McGRE&91] qui ajoute les règles et les contraintes ; d'autres augmentent l'efficacité de la classification, en réduisant les capacités d'expression du langage comme dans le cas du langage de bases de donnés CLASSIC [BRA91] ou en sacrifiant la complétude comme dans les langages NIKL [KAZ&86] et BACK [NEB88].

[1] aussi appelés systèmes hybrides, systèmes de représentation de connaissances basés sur la classification (CBS), langages de subsomption de termes (TSL), langages terminologiques, systèmes de classification de termes (TC) ou encore famille de KL-ONE.

Les logiques terminologiques incluent deux langages, le langage terminologique (TBox) et le langage assertionnel (ABox). Le langage terminologique sert à décrire les termes ou concepts du monde à partir de concepts plus simples et d'opérateurs de formation de structures ; le langage assertionnel décrit, à partir des termes, des propositions de croyance sur l'état du monde. Ainsi l'expression "un père dont les enfants sont des avocats" décrit un terme tandis que la phrase "Rover est un chien marron à trois pattes" indique une assertion, une proposition dont la valeur de vérité dépend du contexte d'évaluation.

Langage terminologique

De même que les schémas, les logiques terminologiques structurent la connaissance autour des objets ; le principal type d'objet de représentation est le concept ; un **concept** est la description d'une structure potentiellement complexe. Un concept est composé d'un ensemble de **rôles** (propriétés, parties, etc.) et un ensemble de **conditions structurelles** qui expriment des relations entre les rôles. Le langage terminologique comporte un ensemble limité d'opérateurs ("spécialisation", "restriction", "différentiation", etc) qui permettent de définir un concept complexe à partir de concepts plus simples. Un concept représente une classe d'individus, la classe des individus qui satisfont la structure et les contraintes du concept. KL-ONE distingue deux types de concepts : les **concepts génériques** qui peuvent décrire plusieurs individus dans des contextes (états du monde) différents et les **concepts individuels** qui décrivent un seul individu dans un contexte précis et qui correspondent donc à une classe ayant un seul membre. A part cette distinction ensembliste de concepts, la plupart des logiques terminologiques distinguent les concepts primitifs des concepts définis. Les **concepts primitifs** établissent des conditions nécessaires mais pas de conditions suffisantes pour l'appartenance d'un individu au concept, tandis que les **concepts définis** donnent des conditions nécessaires et suffisantes. Par exemple le concept primitif "personne" peut être défini comme "être vivant et ayant un esprit" ; toute personne doit satisfaire ces contraintes mais le fait de les satisfaire ne suffit pas pour dire qu'il s'agit d'une personne ; par contre, la définition du concept défini vin_rouge, donnée par : "vin_rouge = vin + rouge" indique que tout vin rouge doit être un vin (satisfaire les contraintes de vin) et avoir la couleur rouge et de plus que tout objet étant un vin et ayant la couleur rouge est un vin_rouge. L'appartenance d'un individu à un concept primitif doit être établie par l'utilisateur, celle d'un concept défini est établie par le système.

La relation de subsomption

Les concepts sont organisés en une hiérarchie de généralisation induite par la relation de **subsomption** : un concept A subsume un concept B si l'ensemble d'individus dénoté par A contient l'ensemble d'individus dénoté par B (A est appelé le subsumant et B le subsumé) [LEV&87].

Par opposition à cette définition **extensionnelle** de la subsomption, des définitions logique et intensionnelle on été proposées [WOO91]. Du point de vue **logique**, la définition d'un concept correspond à un prédicat logique unaire qui détermine l'appartenance d'un individu au concept ; à partir de cette définition, un concept A subsume un concept B si "être un individu décrit par B" entraîne logiquement "être un individu décrit par A", c'est à dire si $\forall x$, $B(x) \Rightarrow A(x)$ [McGRE88].

La subsomption **intensionnelle** concerne la structure des concepts : A subsume B si tout individu décrit par B l'est aussi par A ; autrement dit, si l'ensemble de propriétés d'un individu dont la description est définie par B contient l'ensemble des propriétés qui sont spécifiée par A. Tout concept se compose donc des propriétés héritées de ses subsumants, qu'il peut affiner, et des propriétés définies localement. Une description subsume

(au niveau intensionnel) une autre description composite pour toute combinaison des raisons suivantes :

1. Une catégorie primaire dans l'une des descriptions est plus générale que dans l'autre :
   [une personne dont les fils sont docteurs] subsume
   [une femme dont les fils sont docteurs]
2. Un modificateur de relation dans l'un des deux est plus générale que dans l'autre :
   [une personne dont les fils sont professionnels] subsume
   [une personne dont les fils sont docteurs]
3. Une condition générale dans l'un des deux est plus générale que dans l'autre :
   [un enfant dont un des parents nettoie sa chambre] subsume
   [un enfant dont la mère nettoie sa chambre]
4. La description la plus spécifique inclut une catégorie, modificateur ou condition qui n'est pas présent dans la description la plus générale :
   [une personne dont les fils sont docteurs] subsume
   [une personne dont les fils sont docteurs et qui aime conduire]

Il est important de noter que la subsomption intensionnelle est plus restrictive que la subsomption logique ; en effet tout concept A subsume logiquement le concept absurde $\perp$ car $\forall x$, $\perp(x)$ est faux, et donc $\forall x$, $\perp(x)$ => A(x) est vrai ; pourtant il n'y a pas de subsomption intensionnelle, car les propriétés de A ne sont pas présentes dans $\perp$[1]. La subsomption logique à son tour est plus restrictive que la subsomption extensionnelle ; c'est le cas lorsqu'on utilise deux noms différents pour designer un même concept primitif : par exemple les concepts individuels "étoile_du_soir" et "étoile_du_matin" désignent, par deux noms différents, le même individu, la même étoile ; bien qu'il y ait ici une égalité ensembliste, ni l'implication : étoile_du_soir(x) => étoile_du_matin(x) ni l'affirmation contraire : étoile_du_matin(x) => étoile_du_soir(x) ne peuvent être déduites à partir des prédicats définissant ces concepts.

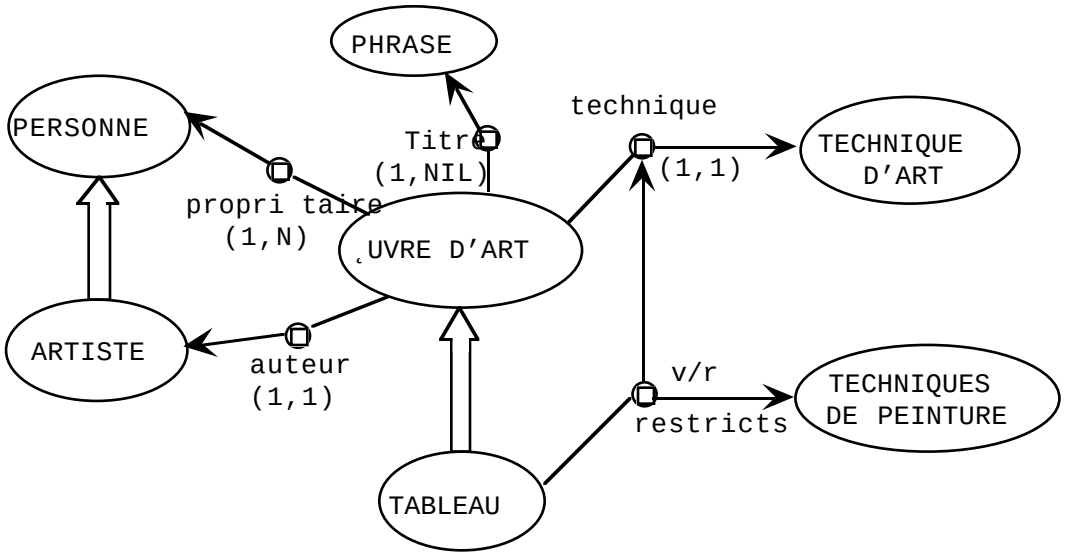


Fig. 1.12 : Représentation des concepts œuvre d'art et tableau présentés auparavant (Fig. 1.10) en KL-ONE : le concept œuvre d'art a les rôles propriétaire, auteur, titre et technique ; il subsume le concept tableau qui hérite de ses propriétés et contraint ("restricts") les techniques possibles.

[1] Bien que au niveau conceptuel on puisse voir le concept $\perp$ comme possédant toutes les propriétés de tous les concepts de la base, dans les systèmes réels sa description ne comporte pas toute cette information et les algorithmes de subsomption intensionnelle ne peuvent pas inférer que $\perp$ est subsomé par tout autre concept.

La relation de subsomption est une relation d'ordre (réflexive, antisymétrique et transitive) qui induit une structure taxinomique des concepts. Cette structure a comme élément maximal le concept THING qui subsume tous les autres concepts de la base. Les concepts primitifs concernent les types primitifs du domaine et sont en général près de la racine THING de la base ; les concepts définis, construits à partir d'autres concepts plus simples sont localisés plus bas dans la hiérarchie.

Langage assertionnel

La partie assertionnelle du système utilise des termes du langages de description pour faire des propositions, des assertions sur le monde. Le langage assertionnel permet de décrire tout ce qui peut être déduit sans servir à la classification. Un monde peut être vu selon différents contextes composés de nexus. Un nexus est une entité regroupant toutes les descriptions qui font référence au même objet du monde ; l'existence d'un individu satisfaisant une assertion sur le contexte courant du monde est établie en le connectant au nexus correspondant à l'intérieur du contexte. Ainsi par exemple, si Paul est le fils unique de Mr. Martin, alors les assertions "Paul est un adolescent" et "Le fils unique de Mr. Martin est malade" référencent le même objet et sont donc regroupées dans le même nexus d'un contexte particulier. Dans ce contexte, ces affirmations peuvent être vraies, mais dans un contexte postérieur, dans lequel Paul serait un adulte guéri, elles peuvent devenir fausses.

Mécanismes de raisonnement

Le mécanisme de raisonnement de base des logiques terminologiques est la classification de concepts, réalisée par un algorithme de classification, appelé le **classifieur** [SCH&83]. Le classifieur prend une nouvelle description de concept et la place à l'endroit correct dans la hiérarchie (§ 4.4.1.). Pour trouver la place appropriée pour le nouveau concept, l'algorithme de classification détermine les relations de subsomption entre ce concept et les autres concepts de la hiérarchie ; ces relations peuvent être spécifiées directement, trouvées par transitivité ou bien calculées à partir de la sémantique des conditions des rôles[1] (en prenant la subsomption intensionnelle). La recherche de la place correcte pour le concept comporte trois étapes : le recherche des **subsumants les plus spécifiques SPS** (concepts qui subsument le concept à classer et dont les sous-concepts ne le subsument pas), la recherche des **subsumés les plus généraux SPG** (concepts subsumés par le concept à classer et dont les sur-concepts ne sont pas subsumés par lui) et puis l'insertion du concept dans la hiérarchie (Fig. 1.13).

1. La première étape se fait en profondeur à partir de la racine : tant qu'un concept subsume le concept à classer, ses sous-concepts sont considérés. Le résultat de cette étape est une coupe de la taxinomie au-dessus de laquelle tous les concepts subsument le concept à classer.
2. La deuxième étape considère les sous-graphes des SPS et détermine, parmi les concepts ayant au moins les mêmes propriétés du concept à classer, les subsumés les plus généraux.
3. Une fois la position trouvée, la troisième étape de l'algorithme insère la nouvelle description dans la hiérarchie, en l'attachant en dessous des subsumants les plus spécialisés SPS et au dessus des subsumés les plus généraux SPG et en éliminant les liens redondants .

Depuis quelques années, les logiques terminologiques s'attaquent au problème de la **classification d'instances** qui consiste à trouver les relations d'appartenance (inclusion) entre l'instance et les descriptions du graphe. L'approche la plus répandue est d'assimiler l'instance au concept individuel correspondant et de classer ensuite ce concept avec

[1] Les algorithmes de classification utilisent pour leurs calculs la subsomption intensionnelle.

l'algorithme de classification de concepts ; cette classification n'a pas besoin de réaliser la deuxième étape de l'algorithme, car des concepts individuels sont dans les feuilles du graphe et ne subsument aucun autre concept. Bien que cette solution permette d'utiliser un même algorithme pour deux mécanismes différents, elle entraîne la création de trop de concepts ; de plus la subsomption fait des comparaisons et validations qui ne sont pas nécessaires lorsque l'on compare un individu (ayant des valeurs pour les rôles) avec un concept (qui a des contraintes pour les rôles). Une approche plus intéressante, utilisée par LOOM [McGRE91] est de profiter des caractéristiques inhérentes à une instance pour faire un algorithme plus efficace que le classifieur ; cet algorithme est appelé le "réaliseur" est sera décrit plus en détail dans § 4.5.2..

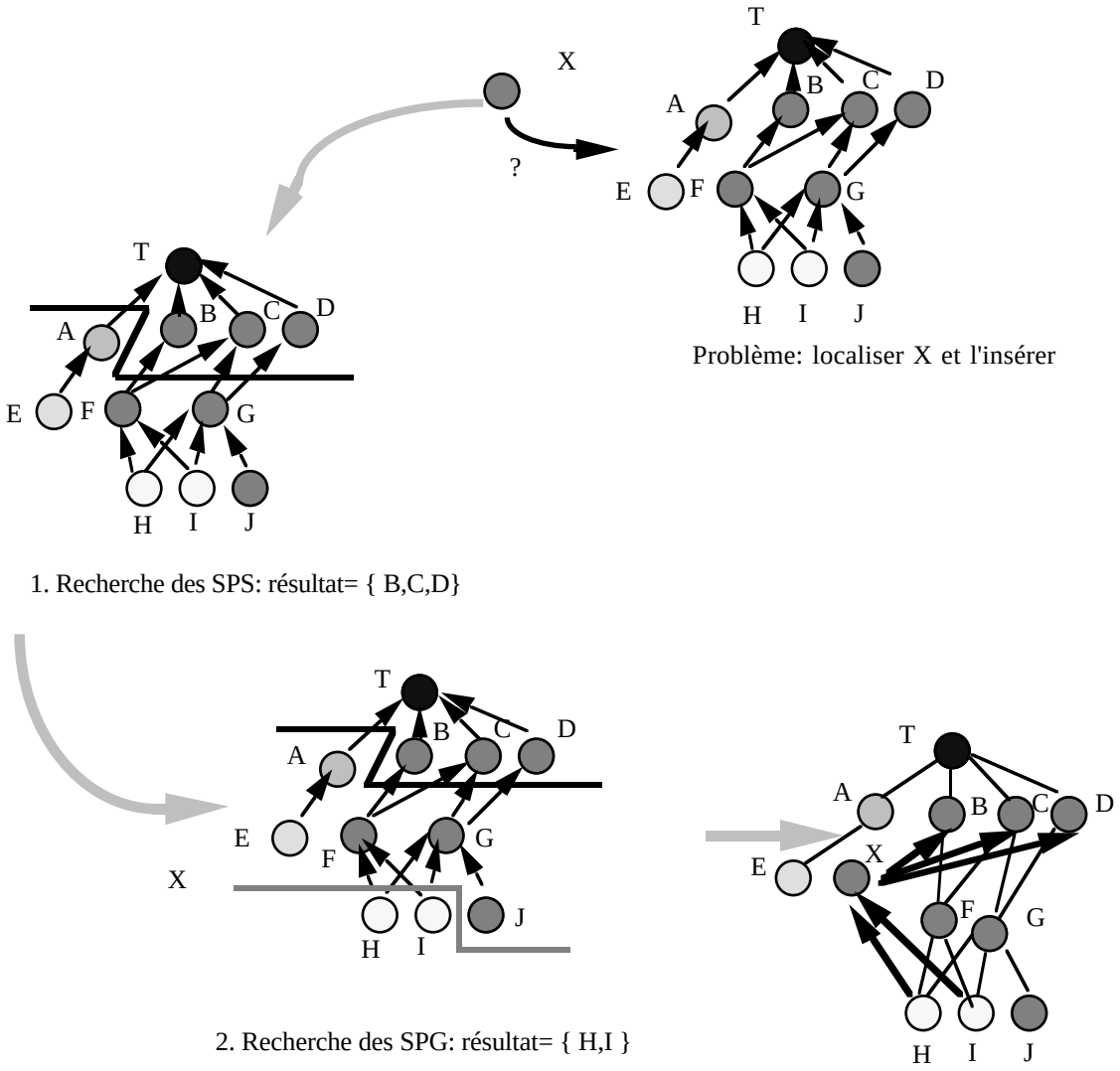


Fig. 1.13 : La classification d'un nouveau concept comporte trois étapes : la rechercher des subsumants les plus spécifiques SPS, la recherche des subsumés les plus généraux SPG, et l'insertion du concept.

Avantages

Par opposition aux réseaux sémantiques dans lesquels les liens et les noeuds sont utilisés avec différentes sémantiques, dans les logiques terminologiques les concepts et les liens ont une sémantique bien définie équivalente à un sous-ensemble de la logique du

premier ordre. Un exemple de cette équivalence est le système OMEGA [ATT&86], qui, sans être une logique terminologique, décrit des composants terminologiques à l'aide de la logique classique. Par ailleurs, la taxinomie de concepts permet une organisation adéquate de la connaissance. Enfin, l'algorithme de classification atteint dans certains systèmes comme CLASSIC une grande efficacité.

Inconvénients

Le souci de complétude du raisonnement a entraîné le développement de systèmes ayant une trop faible expressivité et ne pouvant pas représenter des problèmes intéressants. De plus, le langage de représentation des logiques terminologiques introduit des notions qui ne sont pas toujours faciles à comprendre et compliquent le développement de la base. C'est le cas des distinctions entre propriétés définitionnelles et propriétés contingents ou déduites ; entre propriétés nécessaires et propriétés suffisantes et entre subsomption extensionnelle et subsomption intensionnelle. Un aspect particulièrement difficile à comprendre et donc à expliquer, comme l'a souligné Swartout [PAT&90], est la distinction entre le raisonnement terminologique et le raisonnement assertionnel ; en effet, une affirmation du style "ce téléphone est rouge" peut être interprétée comme une assertion sur un individu particulier du concept "téléphone", indiquant qu'il est rouge, ou bien comme une assertion affirmant l'existence d'un individu du concept "téléphone rouge". Dans le premier cas, la définition du concept n'inclut pas la couleur ; la propriété couleur ne fait pas partie de la définition ; c'est une propriété contingente. Dans le deuxième cas, seuls les téléphones rouges satisfont le concept ; la couleur fait partie des conditions d'appartenance au concept.

Au niveau du raisonnement, utiliser la classification de concepts pour classer des instances est coûteux et n'est pas cohérent avec la sémantique des instances ; la classification d'instances comme un mécanisme particulier permet une relation d'appariement plus efficace que la subsomption. Ce mécanisme est encore assez peu utilisé, entre autre parce que la plupart des logiques terminologiques n'observent pas l'instance mais le concept.

1.3.6. Graphes conceptuels

Les graphes conceptuels sont des systèmes logiques développés pour la représentation les sens des phrases en langage naturel [SOW91]. Proposés par Sowa [SOW84] à partir des idées de Peirce [PEI31], les graphes conceptuels offrent une notation de la logique plus proche des propositions en langage naturel que la logique des prédicats du premier ordre. Par exemple, l'affirmation "un chat est sur le toit" , qui correspond en logique du premier ordre à la formule : $(\exists x) (\exists y) (\text{chat}(x) \Lambda \text{toit}(y) \Lambda \text{sur}(x,y))$, est représentée par le graphe conceptuel [CHAT] -> (SUR) -> [TOIT].

A partir de quelques éléments de base, Sowa développe une théorie formelle sur les graphes permettant de faire des transformations correctes et des inférences logiques et propose un opérateur ϕ pour transformer un graphe dans une formule logique équivalente.

Représentation de la connaissance

Un graphe conceptuel est un graphe fini, connexe, biparti ; il comporte deux types de nœuds : les concepts et les relations conceptuelles. Toute relation conceptuelle a un ou plusieurs arcs, chacun lié à un concept ; un concept simple peut être considéré comme un graphe. Avec cette représentation, il est possible d'apparier les phrases de langage naturel et expliciter le sens des composants de la phase.

Les **concepts** d'un graphe peuvent être **génériques** et **individuels** ; les concepts génériques correspondent aux variables en logique et représentent des individus non spécifiés, les concepts individuels, identifiés par un marqueur, correspondent aux constantes de la logique et décrivent un individu particulier du monde.

Tout concept a un **type** associé (personne, chose, entier, etc.) qui détermine une structure pour le concept ; si le concept est bien formé, il est "conforme" au type (il valide ses contraintes). Les types sont structurés dans un **treillis de types,** induit par la relation de sous-typage ; cette relation d'ordre a comme élément maximal le type universel T et comme élément minimal le type absurde $\perp$. Le treillis de types permet d'inférer des relations de conformité entre un concept et un type ; ainsi, tout individu conforme à un type t l'est aussi à tous les sur-types de t dans le treillis, y compris le type universel ; si un concept est conforme à deux types s et t, alors il l'est aussi au type intersection $s \cap t$; enfin aucun concept n'est conforme au type absurde.

Les **graphes canoniques** sont des primitifs de représentation. Ces graphes ont une sémantique associée et dite correcte pour la représentation. Pour éviter la génération des graphes absurdes, tout système de graphes conceptuels doit partir d'une base initiale de graphes canoniques et en générer d'autres à partir de quatre **règles de formation** : la copie, la restriction, le joint et la simplification. La copie consiste à faire une copie identique du graphe. La restriction est le remplacement du type d'un concept par un de ses sous-types ou par un marqueur individuel conforme au type (Fig. 1.14). Le joint permet de fusionner deux graphes ayant une partie commune en enlevant, d'un des deux, cette partie commune et en collant le reste à la partie correspondante de l'autre (Fig. 1.15). Enfin, la simplification enlève les relations dupliquées du graphe (avec les arcs correspondants).

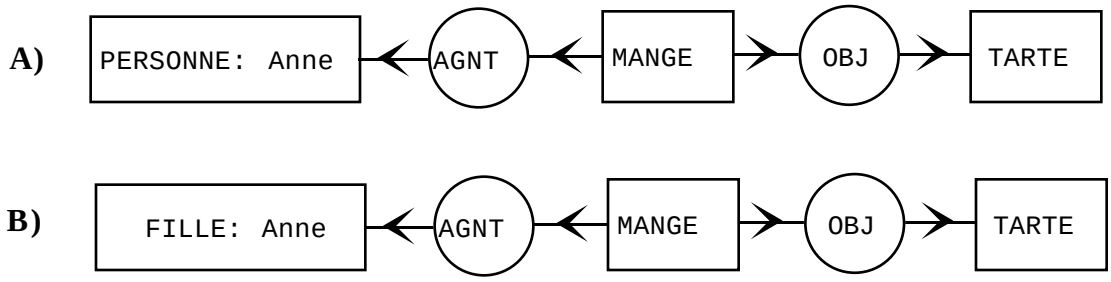


Fig. 1.14 : le graphe B est formé à partir du graphe A par restriction du type PERSONNE de Anne à son sous-type FILLE.

Les règles de formation canonique sont des règles de spécialisation. La relation inverse, la généralisation est réflexive, antisymétrique et transitive et définit un ordre partiel sur le graphe, la **hiérarchie de généralisation**. La relation de généralisation est conservée par la relation de sous-typage de concepts, la relation d'individualisation d'un concept générique et la relation de sur-graphe :

- Sous-type : Si le graphe u est identique au graphe v sauf pour quelques types de v qui ont été changés par des sous-types dans u, alors u<=v[1] (dans la figure 1.14. le graphe A est une généralisation du graphe B).
- Graphe universel : Le graphe [T] est une généralisation de tous les graphes conceptuels.
- Individus : Si u est identique à v excepté que quelques concepts génériques de v ont été remplacés par des concepts individuels (conformes aux types) dans u, alors u<=v (le graphe C est plus général que le graphe C' dans la figure 1.15.).

[1] la relation v généralise u est exprimée par : u<=v.

- Sous-graphe : Si v est un sous-graphe de u alors u<=v (le graphe C', sous-graphe de D est plus général que celui-ci, Fig 1.15.)

La correspondance entre la généralisation de graphes et l'implication logique des prédicats (si u<=v alors $\phi u \supset \phi v$), garantit que toute affirmation valide pour un graphe de la hiérarchie de généralisation, est aussi valide pour ses généralisations.

Outre les types simples, il est possible de définir des types complexes correspondant à tout un graphe conceptuel. Une telle définition consiste à donner une étiquette à un graphe en faisant ainsi une **abstraction** du graphe ; à tout moment des mécanismes de contraction et expansion permettent le passage du graphe à l'étiquette et vice versa.

D'autres éléments du modèle, comme les contextes imbriqués (pour représenter les négations) et les quantificateurs universels des concepts, permettent d'étendre l'expressivité du langage.

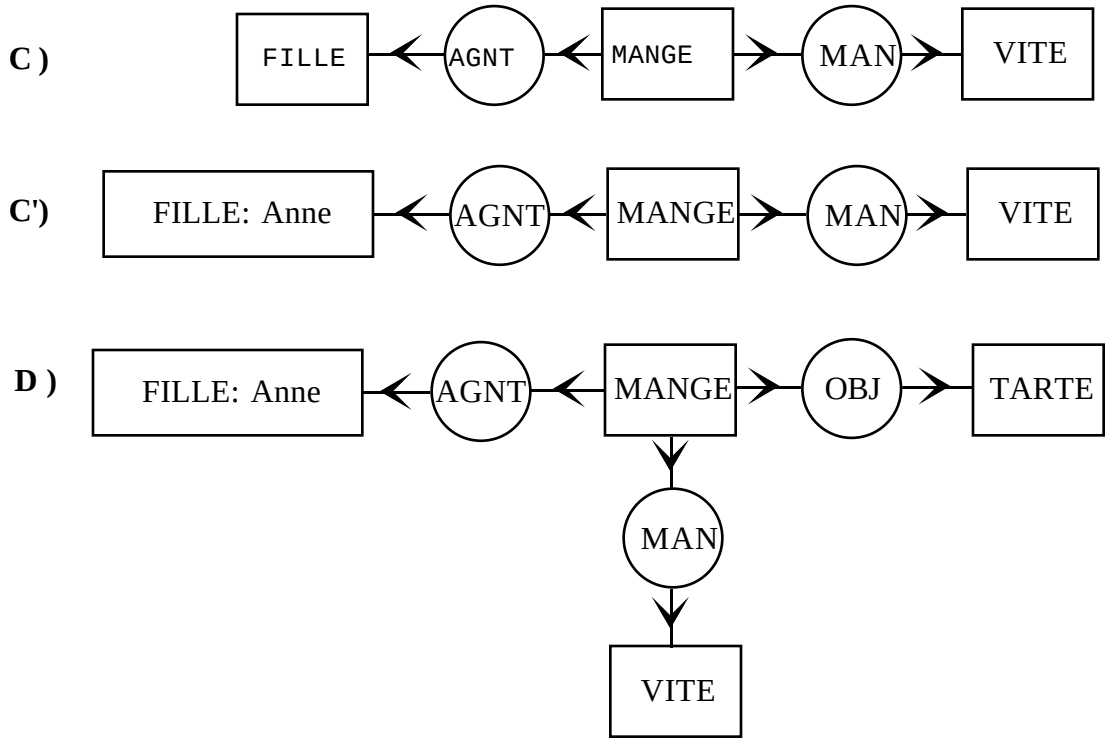


Fig. 1.15 : Par restriction du concept FILLE dans le graphe C à l'individu particulier, Anne, le graphe C' possède une partie commune avec le graphe B ; par joint ils produisent D.

Mécanismes de raisonnement

Les graphes conceptuels offrent deux types de raisonnement qui correspondent à peu près aux approches définitionnelle et prototypique des schémas, le raisonnement exact et le raisonnement plausible. Le raisonnement exact utilise l'équivalence entre la logique du premier ordre et les graphes canoniques, ainsi que la hiérarchie de généralisation pour inférer des connaissances logiquement correctes et sémantiquement convenables. Le raisonnement plausible introduit les notions de schéma et de prototype pour inclure des connaissances spécifiques du domaine dans le raisonnement. Un schéma présente les concepts et relations couramment associés à un type de concept ; les différents schémas associés à un type de concept correspondent aux différentes façons de voir l'utilisation de ce type de concept ; à la différence des définitions des types, les schémas n'établissent pas de conditions nécessaires et suffisantes pour leurs types. Si les schémas prennent en

compte la connaissance du domaine au niveau des types de concepts, les prototypes le font au niveau des individus : un prototype est un individu typique d'un concept, ayant des valeurs par défaut pour les différentes relations intervenant dans le concept. Les schémas et les prototypes permettent de raisonner sur des graphes incomplets mais le raisonnement, à la différence de la logique du premier ordre, n'est pas monotone, et les conclusions obtenues à un moment donné peuvent être remises en question plus tard.

Avantages

A la différence des réseaux sémantiques qui représentent dans un même réseau plusieurs propositions, ainsi que des définitions de concepts, des relations de sous-typages et des liens vers des perceptions du monde, les graphes conceptuels organisent les différents types d'éléments dans des structures différentes : la hiérarchie de spécialisation de graphes, le treillis de types et le "dictionnaire" d'abstractions qui comporte les graphes auxquels on a associé un nom, les définitions. Cette distinction permet la manipulation et la vérification de chaque structure à partir d'un ensemble réduit de règles de cohérence.

D'autre part, l'équivalence entre les graphes conceptuels et la logique du premier ordre garantit la correction des inférences du modèle. De plus la hiérarchie de graphes facilite la généralisation des inférences obtenues sur un graphe particulier. Par ailleurs, le treillis de types permet de contrôler la validité des contraintes de types dans les graphes. Finalement, l'abstraction des concepts offre un mécanisme utile de définition de concepts complexes.

Certains systèmes utilisent la puissance de la relation de subsomption des langages terminologiques pour structurer les graphes de connaissances dans un ordre partiel [WIL91].

Inconvénients

Les graphes conceptuels ont été conçus pour représenter formellement les phrases du langage naturel ; cette motivation explique l'intérêt porté aux problèmes d'ambiguïté de la langue comme par exemple l'interprétation du pluriel, qui ne concerne pas directement la connaissance mais son acquisition sous forme verbale et qui alourdissent la représentation. De plus, le choix d'un graphe conceptuel (équivalent à une proposition logique) comme l'unité de représentation pose les problèmes déjà soulignés pour la logique et les réseaux sémantiques, à savoir que la connaissance concernant un objet est disséminée dans les différents graphes qui en parlent ; ce problème est spécialement grave quand le domaine possède des objets complexes et quand la base de connaissances atteint une grande taille.

Conclusion

Dans les "systèmes à base de connaissances", la connaissance est séparée du contrôle. Cette séparation facilite la modification des connaissances et l'ajout de nouvelles informations à la base. Bien que nettement séparés, la représentation de la connaissance et les mécanismes d'exploitation de cette connaissance sont interdépendants : un raisonnement adapté à une représentation peut s'avérer lourd et compliqué pour une autre [NIW84].

Dans les techniques de représentation de connaissances présentées auparavant, on trouve deux paradigmes de représentation : le paradigme relationnel où les unités de

représentation sont les relations entre les éléments du monde et le paradigme objet qui structure la connaissance du monde en terme de ses objets . La logique, les règles de production et les réseaux sémantiques suivent le paradigme relationnel, tandis que les représentations par schémas et les représentations classe-instance sont des systèmes structurels. Enfin, les logiques terminologiques et les graphes conceptuels, tout en gardant le paradigme relationnel des réseaux sémantiques, ont un composant objet dans leur représentation des concepts et des types.

Bien que comportant tous un composant d'appariement et un composant de parcours de la structure de la base, les mécanismes de raisonnement associés à chaque type de représentation ont des supports conceptuels différents ; ainsi le chaînage avant des systèmes à base de règles supposent qu'un expert raisonne en créant une séquence cause-effet, tandis que pour les schémas de Minsky le raisonnement par analogie suppose l'existence de prototypes auxquels on fait appel pour compléter et comprendre une nouvelle connaissance.

Dans le chapitre suivant, nous présentons plus en détail les modèles de représentation de connaissances par objets. Ce type de représentation, qui utilise comme mécanisme de base la classification, suppose qu'on stocke la connaissance des objets du monde dans des catégories que l'on organise du plus général au plus spécifique. La classification permet d'ajouter un nouvel objet à la catégorie appropriée pour inférer des connaissances propres aux objet de cette catégorie.

Bibliographie

[AND83] ANDERSON J.R., *The Architecture of Cognition.* Harvard University, Cambridge MA, 1983.

[ATT&86] ATTARDI G., SIMI M., *A Description-Oriented Logic for Building Knowledge Bases,* in Proceedings of the IEEE, vol. 74, n°. 10, pp.1335-1344, octobre 1986.

[BAR64] BARTLETT F.C., *Remembering : a study in experimental and social psychology*, Cambridge University Press, 1964.

[BAR&81] BARR A., FEIGENBAUM E. éd. The Handbook of Artificial Intelligence, vol. 1, chapitre 3, William Kaufmann Inc CA, 1981.

[BOB&77] BOBROW D.G., WINOGRAD T., *An overview of KRL, a Knowledge Representation Language.* Cognitive Science, vol. 1, n°. 1, pp.3-45, 1977.

[BRA83] BRACHMAN R.J., *What IS_A is and isn't : An Analysis of Taxonomic Links in Semantic Networks*, IEEE Computer vol. 16, n°. 10, pp.32-36, 1983.

[BRA85] BRACHMAN R.J., "I Lied about the Trees" Or, Defaults and Definitions in Knowledge Representation. The A.I. Magazine, vol. 6, n°. 3, pp.80-93, 1985.

[BRA&85] BRACHMAN R.J., SCHMOLZE J.G.*An Overview of the KL-ONE Knowledge Representation System.* Cognitive Science, vol. 9, n°. 2, pp.171-216, 1985.

[BRA90] BRACHMAN R.J., *The Future of Knowledge Representation* (extended abstract), invited talk, 8th. AAAI, Boston, MA., USA, 1990.

[BRA91] BRACHMAN R.J., *LIVING WITH CLASSIC : When and How to Use a KL-ONE-like Language*, in Principles of Semantic Networks. Explorations in the Representation of Knowledge. J.Sowa (éd.), Morgan Kaufman Publi., chapitre 14, pp.401-456, 1991.

[BUC&78] BUCHANAN B.G., FEINBAUM E.A., *DENDRAL and MetaDENDRAL : Their Application Dimensions*, Artificial Intelligence n°.11, pp.5-24, 1978.

[CAR&73] CARBONNELL J.R., COLLINS A.M. *Natural Semantics in Artificial Intelligence*, Proceedings of the 3rd. IJCAI, pp.344-351, 1973.

[CHA&85] CHARNIAK E., McDERMOTT D., *Introduction to Artificial Intelligence*, Addison-Wesley éd., 1985.

[COH&84] COHEN B., MURPHY G.L., *Models of Concepts,* Cognitive Science, vol.8, nº.7, pp.27-58, 1984.

[COL&83] COLMERAUER A., KANOUI H., VAN CANEGHEM M., *PROLOG, bases théoriques et développements actuels*, TSI, vol. 2, n°. 4, pp.255-292, 1983.

[COR86] CORDIER, M.O., *Informations incomplètes et contraintes d'integrité : le moteur d'inférences Sherlock.* Thèse d'Etat, Université de Paris-Sud, Centre d'Orsay, 1986.

[DAV87] DAVIS H.E.,*VIEWS : Multiple Perspectives and Structured Objects in a Knowledge Representation Language.* Thèse de Bachelor and Master of science, MIT, 1987.

[DES86] DESCLE J.P., *Implications entre concepts : la notion de typicalité*. Travaux de Linguistique et de Littérature, Centre de Philologie et de Littératures Romanes de l'Université de Strasbourg, vo. XXIV, nº. 1 pp.179-202, 1986.

[FAH79] FAHLMAN S., *NETL : A System for Representing and Using Real World Knowledge*, Cambridge, MA, MIT Press, 1979.

[FER88] FERBER J., *Coreferentiality : The Key to an Intentional Theory of Object Oriented Knowledge Representation.* in Artificial Intelligence and Cognitive Science chapitre 6, J. Demongeot, T. Hervé, V. Rialle et C. Roche (éd.), Manchester University Press, 1988.

[FIK&71] FIKES R.E., NILSON N.J. *STRIPS : A New Approach to the Application of Theorem Proving to Problem Solving,* Artificial Intelligence n°.3, pp. 251-288, 1971.

[FIK&85] FIKES R., KEHLER T. *The Role of Frame-Based Representation in Reasoning.* Communications of the ACM, vol. 28, n°. 9, pp.904-920, 1985.

[GEN&87] GENESERETH M.R., NILSSON N.J. *Logical Foundations of Artificial Intelligence*, Morgan Kaufmann Pub., Los Altos C.A., 1987.

[GOO79] GOODWIN J.W., *Taxonomic Programming with KLONE,* Linköping University, Informatics Lab. R.R., février. 1979.

[HAT&91] HATON J.P., BOUZID N., CHARPILLET F., HATON M-C., LÂASRI H., MARQUIS P., MONDOT T., NAPOLI A.,, *Le raisonnement en intelligence artificielle*, InterEditions, Paris, 1991.

[HEN79] HENDRIX G., *Encoding Knowledge in Partitioned Networks,* dans Associative Networks in The Representation and Use of Knowledge by Machine, Findler N. (éd.), New York, Academic Press, 1979.

[HEW69] HEWITT C.E., *PLANNER : A Language for Proving Theorems in Robots,* in Proceedings if the 1th IJCAI, pp.295-301, Washington D.C., 1969.

[KAZ&86] KAZMAREK T., BATES R., ROBINS G., *Recent Developments in NIKL,* in Proceedings AAAI 86, Philadelphia, PA, pp.978-987, 1986.

[KOD86] KODRATOFF Y., *Leçon d'apprentissage symbolique automatique*, dans Actes Journées nationales sur l'IA, CEPADUES, Toulouse, 1986.

[LIE86] LIEBERMAN H., *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages and Applications, septembre, 1986.

[LEV&79] LEVESQUE H.J., MYLOPOULOS J., *A Procedural Semantics for Semantic Networks,* dans Associative Networks : The Representation and Use of Knowledge by Machine, Findler N. (éd.), New York, Academic Press, 1979.

[LEV&87] LEVESQUE H.J., BRACHMAN R.J., *Expressiveness and Tractability in Knowledge Representation and Reasoning,* Computer Intelligence, vol.3, pp.78-93, 1987.

[McCAR80] Mc CARTHY J. *Circumscription : A Form of Non-Monotonic Reasoning.* Artificial Intelligence Journal, vol. 13 , n°. 1-2, pp.27-39, 1980.

[McCOY85] McCOY K., *The Role of Perspective in Responding to Property Misconceptions,* in Proceedings of the 5th IJCAI, vol. 2, 1985.

[McGRE88] MAC GREGOR R.M., *A Deductive Pattern Matcher.* in Proceedings 7th. AAAI'88, St. Paul, Minnesota, pp.403-408, 1988.

[McGRE91] MAC GREGOR R.M., *Inside the LOOM Description Classifier,* SIGART bulletin, vol. 2, n°. 3, Special Issues on Implemented Knowledge Representation and Reasonong Systems, juin, 1991.

[McGRE&91] MAC GREGOR R.M., BURSTEIN M.H. *Using a Description Classifier to Enhance Knowledge Representation*, IEEE Expert Intelligent Systems and Applications, juin, 1991.

[MAS&89] MASINI G., NAPOLI A., COLNET D., LEONARD D., TOMBRE K., *Les Langages à objets*, Intereditions, Paris, 1989.

[MIN68] MINSKY M. L. (éd.) *Semantic information processing*, MIT Press, Cambridge, MA, 1968.

[MIN75] MINSKY M. *A Framework for Representing Knowledge*. in The Psychology of Comp Vision , P.H. Winston (éd.), McGrawHill, New York, chapitre 6, pp.156-189, 1975.

[NEB88] NEBEL B., *Computational Complexity of Terminological Reasoning in BACK*, Artificial Intelligence vol. 34, n°. 3, pp.371-383, 1988.

[NEW&63] NEWELL A., SIMON H.A. *GPS, a program that simulates human thought*, dans Computers and Tought, Feigenbaum & Feldman (éd.), McGraw Hill, New York, pp.279-93, 1963.

[NEW81] NEWELL A., *The Knowledge Level*, Artificial Intelligence, vol. 2 n°. 2, pp. 1-20, 1981.

[NIW&84] NIWA K., SASAKI K., IHARA H., *An Experimental Comparison of Knowledge Representation Schemes*, The AI Magazine, vol.5, nº. 2, pp.29-36, 1984.

[PAT&90] PATEL-SCHNEIDER P.F., QWSNICKI-KLEWE B., KOBSA A., GUARINO N., MAC GREGOR R., MARK W. S., MC GUINNESS D. L., NEBEL B., SCHMIEDEL A., YEN J., *Term Subsumption Languages in Knowledge Representation*, Workshop of TSL'89, AI Magazine, vol. 11, n°. 2, 1990.

[PEI31] PEIRCE C.S., *The Collected Papers of Charles Sanders Peirce*, vol.3, C. Hartsborne and P. Weiss (eds.), Harvard University Press, Cambridge, Mass., 1931-1935.

[QUI68] QUILLIAN M.R. *Semantic Memory*, in Semantic Information Processing, M Minsky (éd.), MIT Press Cambridge, MA, pp.227-270, 1968.

[RAP68] RAPHAEL B., *SIR : A Computer Program for Semantic Information Retrieval*, in Semantic Information Processing, M Minsky (éd.), MIT Press Cambridge, MA, pp.33-135, 1968.

[REC88] RECHENMANN F.*SHIRKA : système de gestion de bases de connaissances centrées-objet.* Manuel d'utilisation, Grenoble, 1988.

[REI78] REITER, R. *On Close-World Data Base*, dans Logics and Data Bases, Gallaire et Minker (éd.), Plenum Press, New York, pp.55-76, 1978.

[REI80] REITER, R. *A Logic of Default Reasoning*, Artificial Intelligence, n°.13, pp.81-131, 1980.

[ROU88] ROUSSEAU B., *Vers un environnement de résolution de problèmes en biométrie : Apport des techniques de l'intelligence artificielle et de l'interaction graphique*, Thèse de doctorat, Université Claude Bernard Lyon 1, Lyon, 1988.

[SCH73] SCHANK R.C., *Computer Models of Thought and Janguage*, San Francisco, Freeman, 1973.

[SCH&79] SCHUBERT L., GOEBEL R., CERCONE N., *The Structure and Organisation of a Semantic Network for Comprehension and Inference*, in Associative Networks : The Representation and Use of Knowledge by Machine, Findler N. (éd.), New York, Academic Press, 1979.

[SCH&83] SCHMOLZE J.G., LIPKIS T.A, *Classification in the KL-ONE Knowledge Representation System*, in Proceedings of the 8th. IJCAI, Karlsruhe, Germany, 1983.

[SHO76] SHORTLIFFE E., *Computer-Based Medical Consultation : MYCIN*, Elsevier, New York, 1976.

[SOW84] SOWA J.F., *Conceptual Structures. Information Processing in Mind and Machine*, Addison-Wesley Publishing, 1984.

[SOW91] SOWA J.F. *Toward the Expressive Power of Natural Language*, in Principles of Semantic Networks : Exploration in the Representation of Knowledge, J.F. Sowa (éd.), chapitre 5, Morgan Kaufmann Publishing, 1991.

[STE&86] STEFIK M.J., BOBROW D.G., KAHN K.M. *Integrating Acces-oriented programming into a Multi-Paradigm Environment.* IEEE Software, pp.10-18, janvier, 1986.

[STI&89] STILLINGS N.A., FEINSTEIN M.H., GARFIELD J.L. et al., *Cognitive Science : An Introduction*, MIT Press, Cambridge, MA, 1989.

[SZO&77] SZOLOVITS P., LOWELL B.H., MARTIN W., *An Overview of OWL, a Language for Knowledge Representation.* Laboratory of Computer Science TM-86 Mass, 1977.

[TUR50] TURING A.M., *Computing machinery and intelligence*, dans Minds 59, pp.433-460, oct. 1950.

[VAR89] VARELA F. *Connaître : les Sciences Cognitives. Tendances et Perspectives*, SEUIL, Paris, 1989.

[WIL91] WILLEMS M., *Subsomption in Knowledge Graphs*, in Proceedings of Declarative Knowledge, International Workshop RDK'91, Kaiserlautern, Germany, juillet 1991, LNCS 567, Springer Verlag, 1991.

[WIN72] WINOGRAD T., *Understanding Natural Language*, New York, Academic Press, 1972.

[WOO75] WOOD W.A., *What's in a Link : Foundations for Semantic Networks*, in Representation and Understanding, Bobrow D., Collins A. (éd.), 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. J.Sowa (éd.), Morgan Kaufman Publishing, chapitre 1, pp.45-94, 1991.

Chapitre 2

Représentation de connaissances à objets

Chapitre 2

Représentation de connaissances à objets

> "La philosophie, à travers de nombreuses discussions sur la nature, a toujours montré l'importance de *l'être*, notion essentielle pour la compréhension de la philosophie occidentale. A l'encontre de la logique mathématique moderne, la conception "objet" considère que la mise en relation est secondaire par rapport à la prééminence de la chose et de ses propriétés" [BAR&91].

Introduction

Les systèmes de représentation de connaissances à objets, RCO, sont des modèles déclaratifs construits autour de la notion d'objet. Apparemment simple, le terme objet a été assimilé à différentes notions, comme celles de concept, description, ensemble et prototype. Chacune de ces interprétations a des conséquences sur la sémantique des liens inter-objets et sur le raisonnement que l'on peut faire avec la représentation. Notre travail est basé sur l'approche classe / instance, dans laquelle une base de connaissances a deux types d'objets : les classes qui représentent un ensemble d'individus, et qui sont décrites par un schéma d'attributs, et les instances qui sont des membres des classes, décrites par des couples attribut-valeur.

Après une brève discussion sur l'intégration des objets dans différents domaines de l'informatique, nous présentons dans les deuxième et troisième parties du chapitre les éléments et relations des systèmes de RCO ayant l'approche classe / instance[1]. Dans la section quatre nous discutons le mécanisme d'héritage et les problèmes qu'il pose. La cinquième section est dédiée aux objets composites : leurs différentes interprétations et représentations existantes dans les représentations à objet. Enfin, dans la section six nous montrons les différences entre une représentation à objets et un langage à objets, puis nous plaçons les représentations ayant l'approche classe / instance par rapport aux logiques terminologiques et aux représentations par prototypes.

2.1. La notion d'"objet" en informatique

On trouve la notion d'objet en informatique dans des domaines qui construisent des modèles conceptuels du monde réel, tels que les bases de données, le génie logiciel et l'intelligence artificielle. Cette notion est née d'une volonté d'avoir dans des modèles

[1] Cette présentation ne prétend pas donner la structure unique d'un système de RCO ; cela est impossible dans la mesure où il n'y a pas de consensus sur les éléments et relations qui interviennent dans un système de RCO.

informatiques un élément de haut niveau reflétant directement des entités du monde réel sur lesquels raisonnent les utilisateurs humains, une abstraction des données.

Parmi les précurseurs du paradigme objet en informatique on trouve les langages orientés objet qui voient un système comme une encapsulation d'objets possédant des propriétés et sachant réaliser certaines actions. Le plus connu de ces langages, SMALLTALK [GOL&83], issu des idées de SIMULA [DAH&66] introduit les notions qui favorisent la modularité et la réutilisabilité : l'héritage, l'encapsulation et l'envoi de messages. Ces idées ont donné lieu à toute une variété de langages ; ainsi les langages acteurs comme ACT [LIE81] renforcent l'indépendance entre les objets et la communication par envoi de messages, les systèmes méta-circulaires comme OBJVLISP [COI87] se décrivent en termes d'eux-mêmes et des langages comme FLAVORS [MOO86] et LOOPS [BOB&83] traitent des aspects additionnels comme l'héritage multiple, les perspectives et les objets composites.

Par ailleurs, on voit apparaître les bases de données orientées objet comme ORION [KIM&89], qui, à la différence des bases de données relationnelles, utilisent comme structure de base les objets, structurés en hiérarchies de classes ; et les outils à objets pour le génie logiciel comme l'environnement de programmation PIE [BOB&80] qui gèrent des contextes, des versions et la documentation des programmes.

En Intelligence Artificielle, le paradigme objet donne lieu aux systèmes de **représentation de connaissances à objets**, RCO. Ces systèmes **déclaratifs** décrivent le monde en termes d'objet: entité du monde, identifiable de façon unique et séparable de son environnement. Les éléments de base d'une RCO sont les **objets** et les **liens** entre ces objets, qui modélisent les relations des entités du monde. Les mécanismes d'inférence utilisent des techniques spécialisées qui tirent profit de cette représentation.

Le terme représentation à objets a été utilisé pour décrire des systèmes aussi variés que les logiques terminologiques et les systèmes issus des schémas de Minsky. En effet, dans tous ces systèmes l'unité de représentation est l'objet, qu'on l'appelle concept, prototype, ou classe et instance. Leurs différences proviennent du fait qu'ils intègrent les objets aux diverses représentations déjà existantes. Ainsi, les logiques terminologiques comme KL_ONE [BRA&85] et LOOM [McGRE&91] intègrent des taxinomies de concepts aux réseaux sémantiques tandis que des systèmes de schémas comme FRL [ROB&77], KRL [BOB&77], KEE [FIK&85], UNITS [STE&85], YAFOOL [DUC88] et SHIRKA [REC88] reprennent des idées des schémas de Minsky en distinguant deux types de schémas, l'un générique (la classe) et l'autre individuel (l'instance).

2.2. Eléments du modèle classe / instance

Les représentations de connaissances à objets suivant l'approche classe / instance décrivent le monde du problème à l'aide de deux groupes d'objets : les classes et les instances d'une part, les relations entre ces objets : spécialisation, appartenance, attribut, composant, etc., d'autre part. Par la suite nous présentons plus en détail chacune de ces notions.

2.2.1. Classe

Une classe représente un ensemble d'individus du monde ; elle peut être décrite de façon extensionnelle ou intensionnelle. La **définition extensionnelle** d'une classe consiste à donner explicitement toutes les instances qui conforment la classe C = $\{I_1,I_2,I_3,\ldots,I_n\}$.

Par exemple, la classe des planètes du système solaire peut être définie par :

Planète = { @Mercure, @Vénus, @Terre, @Mars, @Jupiter, @Saturne, @Uranus, @Neptune, @Pluton },

où @x fait référence à l'instance décrivant la planète x. Une définition extensionnelle est possible uniquement lorsque la classe décrit un ensemble fini d'objets et que l'on connaît tous ces objets.

La définition d'une classe la plus utilisée est la **définition intensionnelle** : la classe est décrite par une liste de propriétés et conditions qui doivent être satisfaites par les instances de la classe. Une telle description de classe contient des informations à trois niveaux : l'information propre à la classe telle que son nom, les attributs présents dans toutes les instances de la classe, et des contraintes et caractéristiques de ces attributs. Ainsi, la plupart des représentations à objets utilisent une structure à trois niveaux : schéma / attribut / facette comme celle des schémas de Minsky (§ 1.3.4) [BOB&77], [ROB&77], [REC88]. Le schéma de classe est composé d'une liste d'attributs ; chaque attribut représente une propriété, relation ou composant, présente dans toutes les instances de la classe ; un attribut est décrit par son nom et une liste de facettes. Les facettes d'un attribut décrivent les contraintes imposées aux instances de la classe pour cet attribut ou bien des méthodes de calcul de la valeur de cet attribut.

$C \{ A_1 \{ Fa_{11}, Fa_{12}, \ldots, Fa_{1x_1}\}, A_2 \{ Fa_{21}, \ldots, Fa_{2x_2}\},\ldots, A_z \{ Fa_{z1}, \ldots, Fa_{zx_z}\} \}$

Pour l'exemple précédent, le schéma de la classe planète (Fig. 2.1) a les propriétés communes à toutes les planètes, telles que : distance_au_soleil, diamètre, durée_d'une_révolution, satellites, planètes_voisines, etc.

2.2.2. Attribut

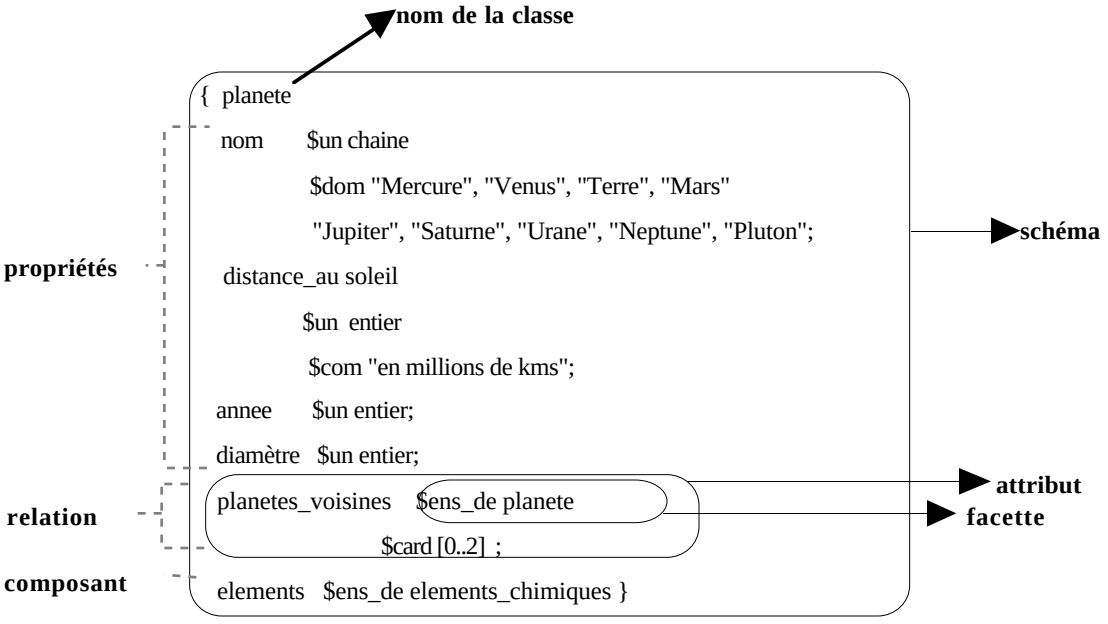


Fig. 2.1. Un schéma de classe est une liste d'attributs/facettes qui représente un ensemble d'individus, ceux qui ont ces attributs et qui satisfont leurs facettes. Un attribut représente une propriété, un composant ou une relation.

Propriété - relation - composant

Un attribut d'une classe représente une propriété ou caractéristique, une relation ou un composant des objets de la classe. Dans l'exemple des planètes, l'attribut distance_au_soleil est une propriété, tandis que l'attribut planètes_voisines décrit une relation existante entre la planète et deux autres planètes du système solaire. Enfin, l'attribut éléments peut être vu comme des composants : une planète est composée de différents éléments chimiques comme hydrogène, carbone, etc. (Fig. 2.1).

Attribut propriété

Une propriété est un attribut intrinsèque à l'objet, qui fait partie constitutive de l'objet et n'a de sens qu'avec lui. Dans l'exemple des planètes, les attributs nom, distance_au_soleil et durée d'une année n'ont pas une identité propre indépendante de l'objet. Dire 1428 n'a un sens conceptuel que lorsqu'on l'attache à la propriété distance_au_soleil de la planète Saturne. En général les propriétés représentent des attributs qui prennent leurs valeurs dans des types simples tels que entier, chaîne, etc. Cependant, définir un attribut comme étant une propriété est un choix de conception qui dépend du domaine et du problème traité.

Attribut relation

Une relation est un attribut à valeurs dans l'ensemble des objets. La relation établit des contraintes entre les objets reliés [SUS&80]. Ainsi la relation planètes_voisines décrit l'ensemble de planètes voisines à une planète donnée une planète avec ses voisines. Très souvent, une relation R liant deux objets X et Y est présente dans les deux objets (dans Y on a la relation inverse) pour pouvoir récuperer un des objets à partir de l'autre [FOR&89]. Dans l'exemple des planètes, si la planète X a comme planètes voisines Y et Z, alors Y et Z ont, entre autres comme voisine X (Fig. 2.2). Garantir la cohérence des relations entre instances est à la charge du système de maintien de la cohérence sous-jacente à la base.

Des relations peuvent être établies entre des instances de classes différentes, par exemple la relation *possède_voiture*, attribut de la classe *Personne*, relie une personne à une voiture (la classe *Voiture* aurait la relation inverse *propriétaire*)

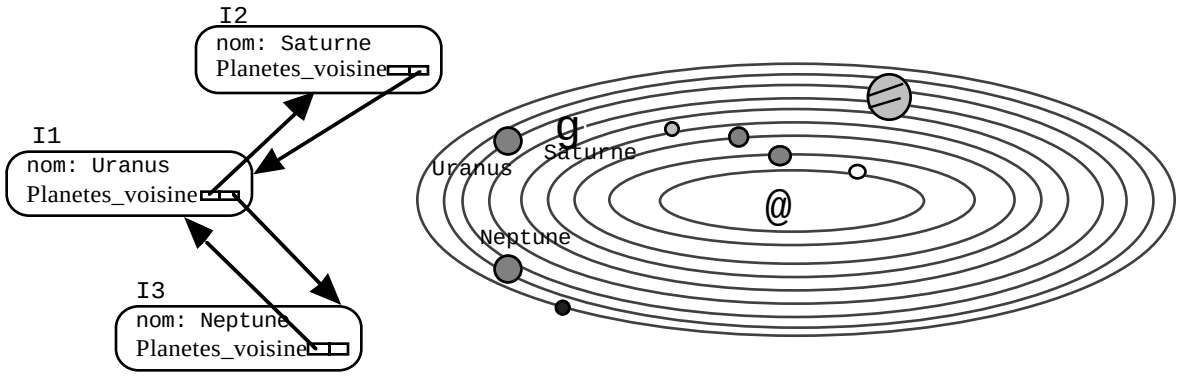


Fig. 2.2. L'attribut planètes_voisines de type planète, établit une relation entre des instances de la classe planète. Cette relation doit être cohérente dans le sens où si X est voisine de Y, alors Y est voisine de X.

Attribut composant

Un composant est un objet du monde qui décrit une partie d'un objet complexe, l'objet composite, son tout. Bien qu'intuitivement simple, la relation "partie-de"

représentée par l'attribut composant peut être interprétée de diverses façons, comme par exemple la relation *membre-collection* existante entre un bateau et une flotte, la relation *portion-masse* entre un morceau de gâteau et le gâteau, la relation *composant-tout* qui existe entre la pédale et le vélo, etc. [WIN&87], [SCH&83]. Informellement, nous pouvons dire qu'un objet composite est un objet complexe pour lequel on peut identifier des parties. Ces parties sont décrites comme des attributs composant de sa classe d'appartenance. Plusieurs systèmes de représentation de connaissances à objets manipulent des objets composites; par exemple ThingLab [BOR81], LOOPS [BOB&83], OBJLOG [DUG91], SHOOD [ESC&90], SRL [FOX&86], CONSTRAINTS [SUS&80], VIEWS [DAV87], OTHELO [FOR&89], ORION [KIM&89]. Leurs différentes façons d'interpréter et gérer la composition sont discutées dans la section (§ 2.5).

Attribut mono-valué - attribut multi-valué

Un attribut peut faire référence à une seule valeur ou bien à plusieurs valeurs. Dans ce deuxième cas, on parle d'un attribut multi-valué, par opposition à un attribut mono-valué. Lors de la définition d'un attribut multi-valué, on peut aussi définir, les sens suivants :

Cardinalité

Le plus petit et le plus grand nombre d'éléments de la collection correspondante à la valeur de l'attribut.

Sémantique

Il faut déterminer si la collection de valeurs simples qui sert comme valeur à l'attribut, représente une valeur monolithique ou bien une combinaison de plusieurs valeurs simples. Ainsi, par exemple, on peut définir l'attribut-propriété *séquence_solution* d'une classe *Problèmes_mathématiques* comme une multi-valeur de cardinalité 10 d'entiers et on peut avoir une instance *Fibonacci* de cette classe, ayant pour cet attribut la valeur [0,1,1,2,3,5,8,13,21,34]. Cette valeur est une seule unité de connaissances et elle doit être manipulée comme une valeur indivisible. Par contre, lorsqu'on définit l'attribut-propriété multi-valué *couleur* pour la classe *oiseau*, la multi-valeur ["rouge","jaune", "verte"] de l'instance *Perroquet* doit être interprétée comme une combinaison de plusieurs valeurs simples. En particulier, on doit pouvoir manipuler chacune de ces valeurs simples ou en ajouter d'autres.

Structure

Une combinaison de valeurs simples peut être vue comme un ensemble, comme une suite ou comme une collection sans éléments dupliqués. Dans le premier cas, l'ordre des éléments n'est pas important et on n'admet pas d'occurrence multiple (c'est le cas de l'attribut *couleur* des oiseaux) ; dans le cas de suites l'ordre devient important et on admet des dupliqués (comme dans l'exemple de la suite de Fibonacci) ; le troisième cas respecte l'ordre et n'admet pas de dupliqués.

La plupart des systèmes qui manipulent des attributs à valeur multiple ne définissent pas clairement la sémantique et la structure utilisées. Certains utilisent avec une même notation plusieurs sémantiques, ce qui peut générer des contradictions et des erreurs lors du raisonnement.

2.2.3. Facette

Pour un schéma de classe[1], les facettes peuvent soit établir des contraintes sur les valeurs possibles de cet attribut dans les instances, soit indiquer des mécanismes d'inférence de cette valeur, soit, enfin, déclencher certaines actions associées aux différentes manipulations de l'attribut.

Facette de contrainte

Parmi les facettes de contraintes on a le type, le domaine et des contraintes intra_attribut (concernant seulement un attribut), inter_attributs (mettant en rapport plusieurs attributs d'un objet) et inter_objets. Les deux premières sont présentes dans tous les systèmes de représentation centrés objet, la troisième est moins courante.

Facette du type

La facette du type indique le type ou classe dans lequel l'attribut peut prendre ses valeurs. Si la facette type d'un attribut *a* d'une classe *C* indique une classe *D*, alors les instances de *C* prennent comme valeur de l'attribut *a* des instances de *D*; on dit que *C* est une classe composée et ses instances sont des instances composées. Par exemple, le type de l'attribut *nom* de la classe *planète* est chaîne, l'attribut *planètes_voisines* est du type de la classe *planète*.

Facette du domaine

La facette du domaine indique, parmi toutes les valeurs du type de l'attribut, celles qui sont des valeurs valides pour cet attribut dans les instances de la classe. Pour la classe *planète* de l'exemple, l'attribut *nom* a une facette domaine qui indique que la collection de chaînes permises pour cet attribut est réduite à : "Mercure", "Vénus", "Terre", "Mars", "Jupiter", "Saturne", "Uranus", "Neptune" et "Pluton". Certains systèmes utilisent aussi la facette sauf pour exclure explicitement des éléments du domaine solution (par exemple: "sauf 13", pour un attribut de type entier, décrit le domaine composé de tous les entiers sauf la valeur 13).

Facette contrainte

```
{ planete
  nom

  planetes_voisines  $ens_de  planete
                     $sib_filtre
                     {     planete
                           planetes_voisines  $var  <-  lui
                           nom                $var  ->  planetes_voisines}

}
```

Fig. 2.3. Dans Shirka les filtres permettent de récupérer de la base de connaissances des schémas satisfaisant des conditions particulières. Par exemple, ici la valeur de l'attribut planetes_voisines pour un planete X est obtenue en récupérant tous les schémas de planete ayant X dans l'ensemble de valeurs de leur attribut planetes_voisines

D'autres contraintes de valeurs peuvent être décrites à l'aide de prédicats ou relations intra_attribut (concernant un seul attribut) ou inter_attributs (liant plusieurs attributs). Par exemple, pour indiquer dans la classe *planète* qu'aucune planète ne peut

[1] Pour un schéma d'instance, la seule facette est la valeur de cet attribut pour cette instance. Les schémas d'instance sont présentés dans § 2.2.4.

être à plus de 6000 millions de kms du soleil, on peut ajouter le prédicat intra_attribut : <= *6000* pour l'attribut *distance_au_soleil*. Peu de systèmes permettent la définition des contraintes inter_objets. En SHIRKA [REC88] par exemple, elle est implantée à l'aide des filtres (comme ceux des schémas de Minsky (§ 1.3.4)). Ainsi, pour établir la relation de voisinage entre planètes, on peut définir un filtre attaché à l'attribut_relation *planètes_voisines* du schéma de classe *planete* (Fig. 2.3)

Facette d'inférence de valeurs

Un attribut peut avoir des facettes qui permettent de calculer sa valeur pour une instance.

Valeur fixe

Si toutes les instances de la classe ont la même valeur pour l'attribut, cette valeur peut être indiquée dans la classe comme étant la valeur (fixe et unique) de l'attribut. Par exemple, on peut ajouter à la classe planète l'attribut étoile de type chaîne, qui aurait pour toutes les planètes la valeur "soleil". Parmi des systèmes permettant la définition de facettes de valeurs pour les classes, on peut citer FRL [ROB&77], MERING [FER84] .

Attachement procédural pour un calcul

Plusieurs représentations à objet [REC88] utilisent l'attachement procédural pour inférer la valeur d'un attribut à partir des autres attributs par une méthode prédéfinie de calcul. Par exemple, si la classe *planète* a deux nouveaux attributs, *masse* et *densité*, alors l'attribut *densité* peut avoir une méthode de calcul à partir de la masse et du diamètre. Ainsi, si pour une instance de planète, le système connaît la masse et le diamètre, il peut déduire la densité.

Valeur par défaut

La valeur par défaut d'un attribut est la valeur standard pour cet attribut pour les instances de la classe. A la différence de la valeur fixe, la valeur par défaut est une hypothèse, prise en l'absence de la valeur dans l'instance. En tant qu'hypothèse, elle peut être remise en question à tout moment. De même, les inférences faites avec des valeurs par défaut sont des inférences hypothétiques qui ne peuvent pas être utilisées comme des faits certains. Malgrè le caractère hypothétique du défaut, la plupart des systèmes utilisant la facette par défaut, utilisent cette valeur pour faire des inférences à caractère certain sur la base; c'est le cas de KRL [BOB&77], OBJLOG [DUG91] ,MERING [FER84] et YAFOOL [DUC88].

Facette réflexe

Un réflexe est une action associée à un attribut ; cette action se déclenche lors d'une modification ou d'une requête sur la valeur de l'attribut pour une instance. Pour garantir la cohérence de la relation de voisinage des planètes, on pourrait attacher un réflexe à l'attribut *planètes_voisines* pour que lorsque l'on modifie sa valeur, cette modification soit transmise aux voisines anciennes et nouvelles. Quelques systèmes comme LOOPS [BOB&83], YAFOOL [DUC88], KEE [FIK&85], SHIRKA [REC88] et MERING [FER84] utilisent des réflexes avant une modification de la valeur, après la modification, et pour indiquer si le calcul de la valeur a réussi, ou s'il a échoué, etc.

Les facettes de contraintes servent à faire des vérifications de valeurs des attributs des instances de la classe. Ces facettes sont utilisées lors de la classification d'une instance, pour voir si les valeurs de ses attributs satisfont les contraintes de la classe. Les facettes d'inférence de valeurs et les réflexes, par contre, ne peuvent être utilisées que lorsque l'on est sûr que l'instance appartient à la classe. En effet, ces facettes font

des inférences basées sur le fait que l'instance est bien une instance de la classe en question. En particulier, ces facettes ne servent pas lors de la classification d'une instance.

Supposons, par exemple, que l'on veut définir la classe *Carré* à partir de la classe *Rectangle* (Fig. 2.4). Une première description indique que les deux côtés doivent être connus et que leurs valeurs doivent être égaux (contrainte : côté1 = côté2). La deuxième description attache une méthode de calcul à l'attribut côté2 : la valeur de l'attribut est obtenue en copiant celle de l'attribut côté1. Dans ce deuxième cas, la contrainte côté1 = côté2 est toujours satisfaite, une fois la valeur du côté2 calculée. Si les facettes d'inférence de valeur dans une classe sont utilisées pour vérifier l'appartenance d'un individu à la classe lors d'une classification, alors pour la deuxième description de l'exemple, tout rectangle n'ayant pas une valeur connue pour l'attribut côté2, est classé comme étant un carré.

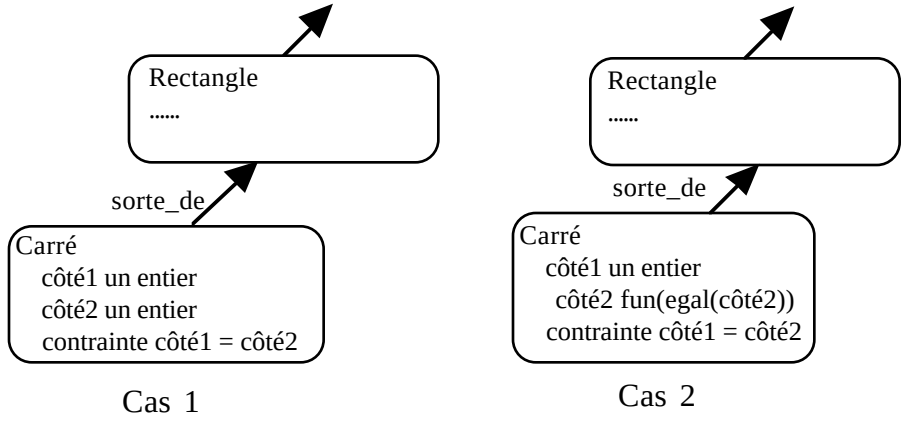


Fig. 2.4. Dans le premier cas, pour qu'un rectangle soit un carré il faut connaître les valeurs de ses deux côtés et que ces valeurs soient égales. Dans le deuxième cas, avec la valeur d'un des côtés, le système peut calculer la valeur de l'autre (la même valeur). Si ce calcul est fait lors de la classification, tout rectangle n'ayant pas de valeur pour *côté2* , est classé comme étant un carré.

2.2.4. Instance et relation d'appartenance

Une instance représente un individu particulier du monde. L'instance, de même que la classe, est décrite par une structure à trois niveaux, ici : schéma / attribut / valeur. Le premier niveau, le schéma, dispose de l'information globale de l'instance comme un identificateur [1] et sa classe d'appartenance ; le deuxième niveau établit la structure de l'instance, ses propriétés, relations et composants ; enfin le dernier niveau indique la valeur pour chaque attribut de l'instance.

Les modèles classe / instance classiques établissent qu'aucune instance ne peut exister sans sa classe d'appartenance, car c'est par cette classe qu'elle se met en rapport avec d'autres objets de la base et qu'elle acquiert les attributs communs aux individus de la même catégorie.

Une instance est liée à sa classe d'appartenance par le lien **est-un** (is-a). Ce lien peut avoir des interprétations différentes selon la signification donnée aux concepts de classe et d'instance [BRA83], [WOO75]. Dans l'interprétation ensembliste, une classe représente un ensemble, une instance représente un individu et le lien est-un la relation "membre de" d'**appartenance** ensembliste. Le lien d'appartenance indique que l'instance

[1] Toute instance est identifiée par une clé unique.

a les attributs décrits par le schéma de la classe et que la valeur que l'instance a pour chacun de ces attributs satisfait les contraintes imposées par les facettes de l'attribut dans la classe. Par exemple, l'assertion *Pluton* est-un *planète* indique que *Pluton* fait partie de l'ensemble représenté par la classe *planète* et qu'il a les attributs : *nom*, *distance_au_soleil*, *année*, etc., avec des valeurs valides, nom = "Pluton", distance_au_soleil = 5889, etc.

Lien d'appartenance et lien d'instanciation

Le lien est-un est très souvent nommé **lien d'instanciation**. Ce terme provient des langages à objets ; dans ces langages la méta-classe d'une classe possède des méthodes d'instanciation qui crée une instance de la classe. Une instance est attachée pour toujours à la classe qui la crée, sa classe d'instanciation, par un lien non modifiable, le lien d'instanciation, est-un. Ainsi, dans les langages orientés objets une instance peut être membre d'une classe seulement si celle-ci ou une de ses sous-classes est mentionnée explicitement dans l'instance [PAT90].

Dans les représentations à objets nous préférons parler de **lien d'appartenance ou d'attachement** car le lien est-un reflète une relation d'appartenance ensembliste et non pas le résultat d'une méthode de création d'instances[1]. De même que pour les langages à objets, dans les représentations à objets, une instance est créée par un mécanisme d'instanciation qui détermine une structure initiale pour l'instance et un lien est-un vers une classe ayant cette structure, sa classe d'appartenance. Mais, à la différence des langages à objet, dans les représentations à objets, le lien est-un n'est pas figé ; une instance peut migrer d'une classe à une autre, lorsque l'information qui lui est associée s'enrichit ou se modifie. Cette migration entraîne une mise à jour du lien est-un et une validation des contraintes de la nouvelle classe, pour vérifier que l'instance appartient à l'ensemble représenté par la classe. Par exemple, si une personne perd son emploi, l'instance la représentant doit migrer de la classe *Travailleur* vers la classe *Chômeur* ; son lien d'instanciation est-un doit donc se modifier (Fig. 2.5).

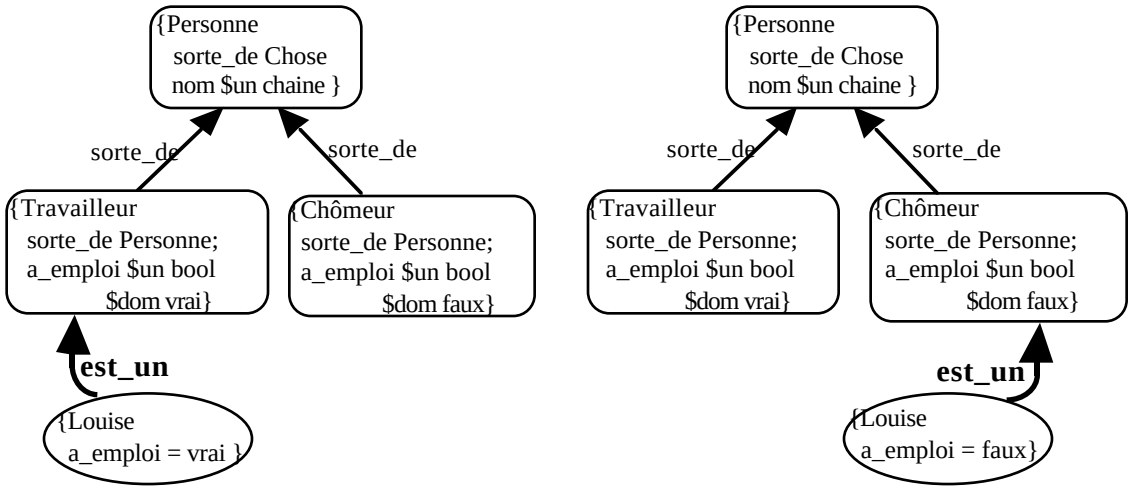


Fig. 2.5. Le lien d'appartenance est-un change lorsque les valeurs des attributs de l'instance ne satisfont plus les contraintes de la classe. Dans ce cas l'instance migre vers une nouvelle classe d'appartenance et le lien est-un est mis à jour en conséquence.

La plupart des modèles classe / instance exigent l'unicité du lien **est-un** : un individu ne peut être lié directement qu'à une seule classe, sa classe d'instanciation. Quelques

[1] Les différences entre un langage à objets et une représentation par objets sont discutées dans §2.6.1.

systèmes tels que OWL II [MAR79], permettent la multi-instanciation : un individu peut appartenir à plusieurs classes (ensembles) de la base.

Conditions nécessaires et suffisantes d'appartenance à une classe

Dans la section précédente nous avons dit que le lien est-un indique que l'instance a les attributs décrits par le schéma de la classe et que la valeur que l'instance a pour chacun de ces attributs satisfait les contraintes imposées par les facettes de l'attribut dans la classe.

Cette relation entre l'appartenance d'une instance à une classe et la satisfaction des contraintes de la classe par les valeurs des attributs de l'instance peut être interprétée de trois façons :

- si I **est-un** C (I est membre de la classe C) alors I satisfait les contraintes imposées par les attributs de C.
- si I satisfait les contraintes imposées par les attributs de la classe C, alors I **est-un** C.
- I **est-un** C si et seulement si I satisfait les contraintes imposées par les attributs de la classe C.

Conditions nécessaires

Dans la première interprétation, les conditions établies par le schéma de la classe sont des conditions nécessaires, des conditions que tout individu de la classe doit satisfaire. Des systèmes contenant seulement des conditions nécessaires, mais non pas des conditions suffisantes, peuvent utiliser le mécanisme de classification pour conclure qu'un item n'est pas un membre d'une classe, mais non pas pour conclure qu'un item EST un membre d'une classe. En effet, si une instance x ne satisfait pas toute la description de la classe[1], alors x n'appartient pas à la classe (par la contraposée de la définition). Par contre, la satisfaction de la description de la classe ne suffit pas à déterminer l'appartenance de l'instance à la classe. Dans ce type de systèmes, l'appartenance d'une instance à une classe ne peut pas être inférée, elle doit être donnée explicitement. A ce type de systèmes appartiennent les langages orientés objets, car ils ne peuvent pas décider de l'appartenance d'une instance à une classe.

Conditions suffisantes

La deuxième relation correspond à une interprétation du schéma de classe comme un ensemble de conditions suffisantes ; ainsi, pour toute instance x, si x satisfait les propriétés du schéma de la classe, alors x appartient à l'ensemble décrit par la classe. Cette condition est beaucoup plus forte que celle des propriétés nécessaires. L'établissement de conditions suffisantes pour une classe permet l'utilisation de la classe comme un filtre, comme un "reconnaisseur" pour déterminer si une instance appartient à une certaine catégorie d'objets. En effet, à partir des schémas de classe, le système peut inférer la classe d'appartenance d'une instance de la base, en regardant les conditions des différentes classes et les valeurs des attributs de l'instance. Le mécanisme par lequel le système décide de l'appartenance d'une instance à une classe, la **classification d'instances**, est un des mécanismes de raisonnement les plus puissants des représentations par classes et instances. A la différence du cas précédent, dans ce cas si une instance ne satisfait pas la description d'une classe, on ne peut pas dire qu'elle n'appartient pas à la classe.

---

[1] La description d'une classe est l'ensemble de ses attributs et leur facette, le schéma de la classe.

Conditions nécessaires et suffisantes

La dernière relation présente la double implication : pour appartenir à la classe, une instance doit satisfaire sa description et toute instance la satisfaisant est un de ses membres. De plus une instance qui ne satisfait pas une partie de la description de la classe n'en est pas un membre et aucun élément qui ne soit pas membre de la classe ne peut satisfaire sa description.

Bien que très peu de systèmes utilisent cette double implication pour toutes les classes de la base, certains systèmes des représentations à objet ayant l'approche classe/instance l'utilisent lorsqu'une première localisation de l'instance dans la base a été faite. Comme nous allons voir par la suite (§ 2.3) ces systèmes structurent les classes dans une hiérarchie de spécialisation. Dans cette hiérarchie, les sous-classes de la classe d'appartenance d'une instance établissent des conditions **nécessaires et suffisantes** d'appartenance pour cette instance. Ainsi, lorsque l'on sait qu'une instance appartient à une classe (soit parce qu'elle a été crée par instanciation de cette classe, soit parce que l'utilisateur ou un mécanisme du système assure son appartenance à la classe), il suffit qu'elle satisfasse la description d'une des sous-classes de cette classe, pour qu'elle appartienne aussi à cette sous-classe ; et de plus, si toute instance de cette sous_classe satisfait nécessairement la description de la classe. Supposons par exemple que l'on crée explicitement une instance "Louise" comme étant un membre de la classe "Personne" (Fig. 2.5) (cette instance satisfait nécessairement la description de "Personne"). Une fois que l'on sait que Louise est une personne (dans ce cas on l'a affirmé explicitement), le schéma de la classes "Travailleur" établit des conditions nécessaires et suffisantes pour l'appartenance de Louise à cette classe (de même pour "Chômeur"). En effet, toute personne ayant pour l'attribut a_emploi la valeur vrai est un travailleur et toute personne ayant la valeur faux est un chômeur.

Autrement dit, lorsque l'on sait qu'une instance I appartient à une classe C, la satisfaction de la description d'une classe C' sous-classe directe de C est **suffisant** pour que I appartienne à C' et si I appartient à C', alors I satisfait **nécessairement** la description de C'. C'est grâce à cette propriété que l'on peut faire une classification certaine d'une instance dans le sous-arbre de spécialisation de sa classe de création et que l'on peut faire des déductions de non_appartenance (savoir qu'une instance n'appartient pas à une classe).

Cette propriété devrait être vraie indépendamment du niveau de la hiérarchie dans lequel on instancie l'objet. Cependant plusieurs recherches [ROS78] [PET&88] ont montré que dans des niveaux dans lesquels on décrit des classes trop générales ou abstraites la description complète des sous-classes n'est pas toujours facile. Ce problème a été traité dans les logiques terminologiques par la distinction entre concepts primitifs et définis et dans divers représentations à objets par la partition de la base en familles [BOB&77] ou catégories de base.

D'autres interprétations de la relation d'appartenance

La présentation précédente des conditions nécessaires et suffisantes d'appartenance à une classe suppose que l'on peut toujours définir complètement une classe en termes de ses attributs et leurs contraintes. De plus, pour toute instance de la base, un appariement des valeurs des attributs de l'instance avec les contraintes des attributs de la classe doit permettre de déterminer si l'instance appartient à la classe ou pas. Ces hypothèses sont très fortes et pas toujours valides, comme l'a constaté [KAY88].

Par opposition à cette approche classique, certains systèmes ont établi des relations d'appartenance moins rigides, telles que la relation d'appartenance à trois valeurs, les

relations avec des poids d'appartenance, l'établissement d'hypothèses pour établir la relation d'appartenance et la représentation par prototypes (discutée en § 2.6.2) [KAY88].

Relation d'appartenance à trois valeurs

Une première approche est d'avoir outres les valeurs sûre et impossible pour la relation d'appartenance, une valeur possible pour indiquer les cas dans lesquels on ne peut ni nier ni assurer l'appartenance. Cette approche, toute en gardant l'hypothèse que l'on peut définir un concept en terme de ses attributs et contraintes, accepte que parfois on ne puisse pas décider de l'appartenance d'une instance à une classe, notamment lorsqu'on travaille avec des instances incomplètes (monde ouvert) [REC88].

Solutions multi-valuées

Des systèmes acceptant que des instances d'une classe aient différents degré ou niveaux d'appartenance à la classe établissent des relations d'appartenance valuées, soit par degrés qualitatifs ou quantitatifs d'appartenance par probabilité [GRA88] ou par la définition de classes floues et le calcul d'appartenance avec des logiques floues [ROS90].

Raisonnement non monotone

Les systèmes utilisant cette approche partent du fait que la connaissance est incomplète et que, pour pouvoir raisonner avec elle, il faut faire des hypothèses sur les informations manquantes, quitte à les défaire ensuite. Plusieurs théories de raisonnement hypothétique telles que la circonscription [McCAR80] et les valeurs par défaut [REI80] ont été utilisées pour faire des hypothèses, soit sur les valeurs manquantes d'une instance, soit directement sur sa relation d'appartenance à une classe [COR86].

Dans le chapitre 4 nous montrons plus en détail ces différentes interprétations.

2.2.5. Relation de spécialisation

Une classe permet de décrire un groupe d'objets semblables ; cette "ressemblance" entre les objets d'une classe est d'autant moins grande que la classe impose peu de contraintes sur les propriétés des objets qui lui appartiennent[1]. En effet, plus une classe est générale, moins l'information qui constitue le fait d'appartenir à cette classe est précise [ROU88]. Pour faire une classification des objets du monde, l'être humain part des catégories générales imposant peu de contraintes à ces objets et détermine ensuite des sous-catégories plus ou moins spécifiques en ajoutant de nouvelles contraintes aux contraintes existantes dans la catégorie générale.

De même, dans les bases de connaissances organisées en classes, on a une spécialisation des classes générales en des classes plus spécifiques ou sous-classes. La relation entre une sous-classe et la classe plus générale qui la contient, sa sur-classe, se fait par un lien **sorte-de**. Le mécanisme de spécialisation établi par le lien *sorte-de* permet la définition des concepts complexes à partir des concepts plus généraux.

Dans l'interprétation ensembliste des classes, la relation de spécialisation définie par le lien *sorte-de* traduit la relation d'inclusion d'ensembles. Ainsi, si A est sous-classe de B, alors tout élément de la classe A est aussi membre de B. Une sous-classe décrit donc un sous-ensemble de la sur-classe contenant des éléments plus "semblables". Au fur et à mesure qu'on ajoute des restrictions à la classe décrivant un ensemble, on a une description plus précise d'un ensemble plus spécifique des individus du monde.

---

[1] Par abus de langange on dira qu'une instance "appartient à une classe" ou qu'elle est un "membre de la classe" alors qu'elle appartient à (est membre de) l'extension de la classe.

Certains ystèmes utilisant le modèle classe / instance travaillent avec une **spécialisation stricte**, c'est-à-dire l'ensemble potentiel décrit par une sous-classe C1 d'une classe C est un sous-ensemble **strict** de l'ensemble potentiel d'individus de la classe C.

Affinement des attributs

Une classe D ne peut être une spécialisation (une sous-classe) d'une classe C que si D possède au moins les mêmes attributs que C, car tout élément d'une sous-classe doit rester un élément de sa sur-classe. L'affinement de la sur-classe par la sous-classe peut se faire par l'ajout des attributs qui n'apparaissent pas dans la sur-classe et/ou par la modification des attributs de la sur-classe [STE&85].

Nous nous plaçons dans le cadre des système qui interdissent le masquage, c'est-à-dire nous nous intéressons aux systèmes pour lesquels la modification des attributs de la sur-classe garde le sens d'inclusion ensembliste donné à la relation de spécialisation. Dans ce contexte, deux types de modification sont possibles : l'affinement des facettes de contraintes et l'affinement des facettes d'inférence de valeurs.

L'affinement des facettes de contraintes peut se faire par modification du type de base de l'attribut à un de ses sous-types, par réduction de l'ensemble de valeurs possibles donné par la facette domaine ou par l'ajout de contraintes (Fig. 2.6).

```
{ Rectangle_color
  sorte_de  =  figure_plane;
  cot _1       $un  entier;
  cot _2       $un  entier;
  cot _3       $un  entier;
  cot _4       $un  entier;
  contrainte   cot _1  =  cot _2;
  contrainte   cot _3  =  cot _4;
  p rim tre    $un  entier
               $si_besoin calcul(2 * cot _1 + 2 * cot _3)
  couleur      $un  cha ne
               $dom   bleu , rouge , jaune , blanc , vert , noire }

{  Carr _rouge
  sorte_de  =  Rectangle_color ;
  contrainte   cot _1  =  cot _3;
  p rim tre    $si_besoin  calcul  (  4*  cot _1);
  couleur      $dom           rouge   }
```

Fig. 2.6. La classe rectangle_coloré est spécialisée dans la classe carré_rouge par trois types d'affinement : l'ajout d'une contrainte, la réduction du domaine de valeurs possibles pour l'attribut couleur et la redéfinition de la méthode de calcul du périmètre (les deux méthodes donnent pour les carrés la même valeur).

L'affinement des facettes d'inférence de valeurs doit garantir la conservation de la relation d'inclusion ensembliste entre les classes. Ainsi, la facette valeur ne peut pas être modifiée car elle établit une valeur fixe pour toutes les instances de la classe et donc de ses sous-classes. La fonction de calcul attachée à un attribut de la sur-classe par une facette d'attachement procédural peut être redéfinie dans la sous-classe ; pour une instance donnée de la sous-classe, cette nouvelle fonction doit retourner, dans le même

contexte, la même valeur que celle de la sur-classe. La redéfinition des attachements procéduraux permet d'utiliser des méthodes de calcul plus robustes, qui doivent satisfaire moins de conditions pour se déclencher, ou bien des méthodes plus efficaces pour un ensemble spécifique d'individus (par exemple le calcul de la surface d'un *polygone* peut se simplifier pour la sous-classe des *rectangles*). Il est important de noter que, à la différence des facettes de contraintes, le système ne peut pas vérifier de façon automatique la conservation de la relation d'inclusion ensembliste pour l'attachement procédural. Enfin, la facette défaut, en tant que valeur hypothétique, peut être changée dans une sous-classe, pourvu que la nouvelle valeur par défaut soit une valeur permise pour la sous-classe.

2.3. Taxinomie de classes

La connaissance d'un univers peut être représentée par une structure hiérarchique de classes dans laquelle les classes sont liées par un lien de spécialisation. La typologie de cette structure de classes dépend des objets du monde et de la perspective à partir de laquelle on les regarde.

Dans cette partie nous allons montrer les différentes typologies de hiérarchies; nous allons décrire le mécanisme d'héritage ainsi que l'héritage simple et l'héritage multiple, et puis nous allons exposer les différentes approches prises pour traiter les conflits posés par l'héritage multiple : l'approche linéaire, l'approche graphique et l'approche circonstancielle. A la fin, nous présentons la multi-instanciation.

2.3.1. Hiérarchies de classes

La relation de spécialisation "être sous-classe de", définie par le lien *sorte-de*, établit un **ordre partiel** entre les classes, propriété qui a été identifiée par Brachman :

> "It was quickly noted that the IS-A connections formed a hierarchy (or, in some cases, a lattice) of the types being connected - that is, the IS-A relation is a partial order" [BRA83]

En effet, cette relation est **réflexive** : une classe est sous-classe d'elle-même, de la même façon qu'un ensemble est sous-ensemble de lui-même (cela est vrai lorsqu'on n'utilise pas la spécialisation stricte). Elle est aussi **antisymétrique,** car si une classe A est sous-classe d'une classe B et B est sous-classe de A, alors elles décrivent le même ensemble d'éléments, la même classe, car tout élément de A est dans B et tout élément de B est dans A. Finalement, il s'agit d'une relation **transitive** : si A est sous-classe de B et B est sous-classe de C, alors A est sous-classe de C par transitivité de la relation d'inclusion des ensembles ; dans ce cas on dit que A est une sous-classe directe de B et une sous-classe indirecte de C.

Par exemple, dans une base de connaissances sur les *Objets utiles à l'homme* (Fig. 2.7), la classe *Planche-a-voile* est sous-classe des classes *Utilisant_la_force_éolienne*, *Machine, Objet, Aquatique,* et *Moyen_de_Locomotion*. La classe *Objet* représente tous les objets de l'univers de discours ; cette classe est sur-classe directe ou indirecte de toutes les autres classes de la base. Les classes *Planche_à_voile* et *Delta_Plane* ont plusieurs sur-classes.

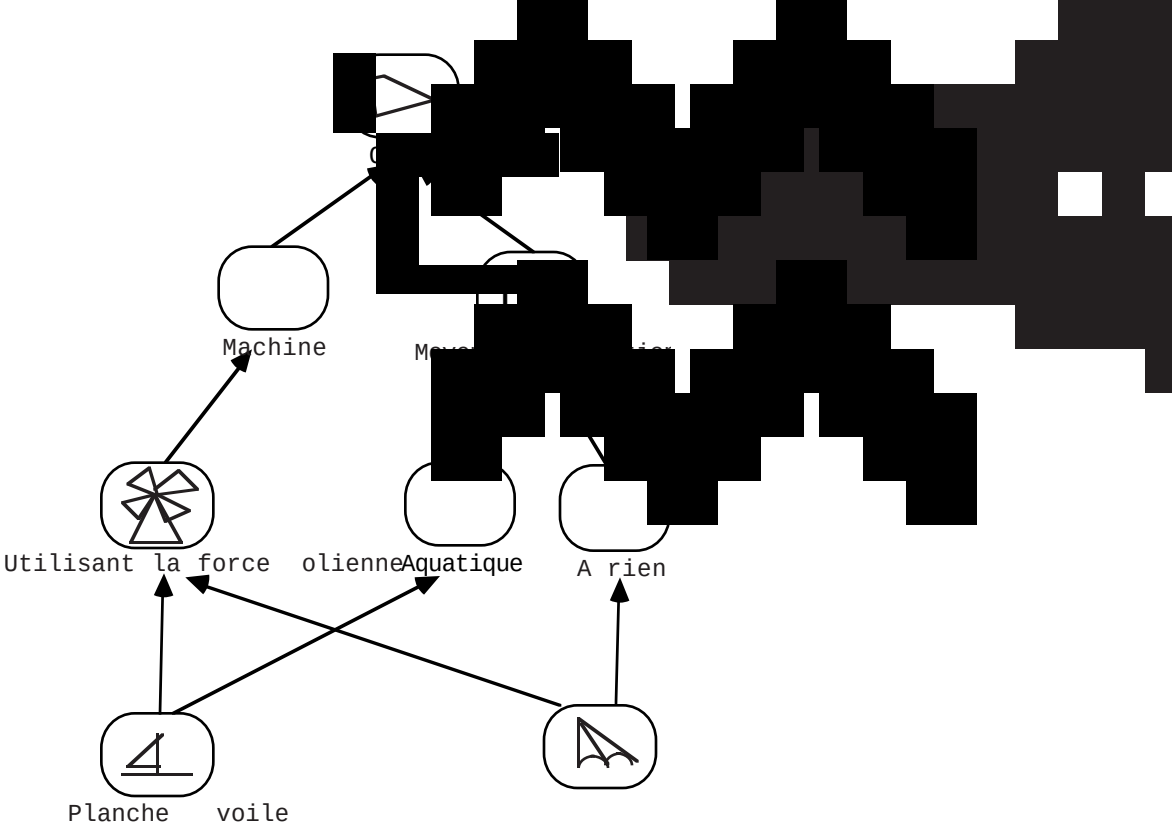


Fig. 2.7. La spécialisation établit une relation d'ordre partiel entre les classes. La classe Planche à voile est sous-classe de Aquatique ; Aquatique est sous-classe de Moyen de Locomotion qui par transitivité est sur-classe de Planche à voile.

Cette spécialisation multiple signifie que l'ensemble représenté par la sous-classe est un sous-ensemble de toutes ses sur-classes, donc que les sur-classes, dans ce cas *machine* et *moyen de locomotion*, représentent des ensembles non disjoints d'instances (Fig. 2.8).

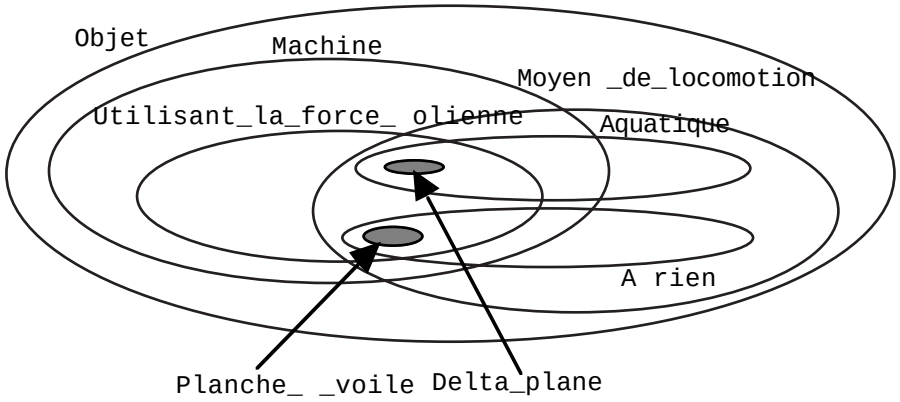


Fig. 2.8. La relation de spécialisation représente l'inclusion ensembliste. Si une classe a plusieurs sur-classes, alors l'intersection des ensembles d'instances représentés par les sur-classes est non vide ; elle a au moins toutes les instances de la sous-classe.

La hiérarchie des classes d'une base de connaissances peut avoir des structures différentes, plus ou moins restrictives ; elle peut être un ordre partiel quelconque, un treillis, un arbre, etc.

2.3.2. Treillis

Un ensemble E partiellement ordonné par une relation d'ordre ≤ est un treillis si et seulement si toute paire d'éléments a,b de E a un élément "union" a∪b et un élément "intersection" a∩b.

- L'union de a et b (a∪b) est le plus petit élément qui soit plus grand que a et b :
  a ≤ a∪b et b ≤ a∪b et pour tout d dans E, si a ≤ d et b ≤ d alors a∪b ≤ d
- L'intersection de a et b (a∩b) est le plus grand élément qui soit plus petit que a et b :
  a∩b ≤ a et a∩b ≤ b et pour tout d dans E, si d ≤ a et d ≤ b alors d ≤ a∩b [SCH86]

Une hiérarchie de classes est un treillis si pour toute paire de classes A, B il existe deux classes C(noté A∪B) et D(noté A∩B) telles que C soit la plus petite sur-classe de A et de B (qu'elle soit sous-classe de toute autre sur-classe de A et B) et D la plus grande sous-classe de A et B (qu'elle soit sur-classe de toute autre sous-classe de A et B).

La hiérarchie montrée ci-dessus (Fig. 2.7) n'est pas un treillis car les classes *planche à voile* et *delta plane* n'ont pas une classe "intersection" et elles ont deux classes "union" possibles *Utilisant_la_force_éolienne* et *Moyen_de_Locomotion* qui ne sont pas comparables (aucune n'est plus grande que l'autre).

La hiérarchie suivante (Fig. 2.9) est un treillis car, toute paire de classes, y possède une classe "union" et une classe "intersection" :

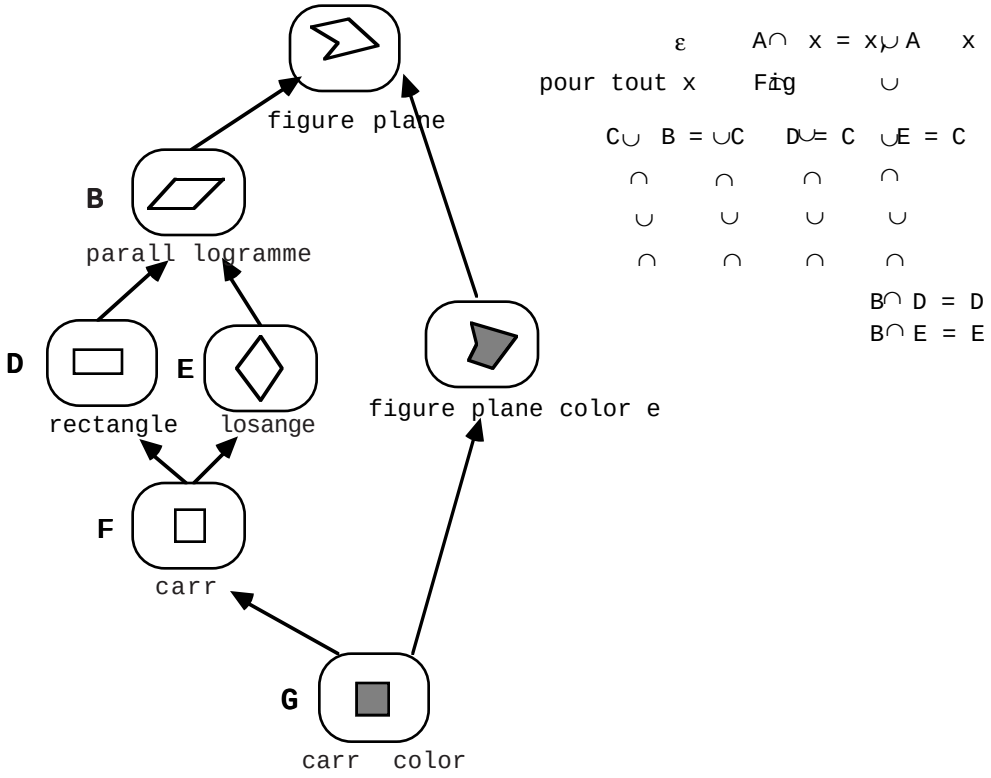


Fig. 2.9. Cette base de Figures planes est un treillis car toute paire de classes a une classe union et une classe intersection.

Tout treillis fini étant complet, le treillis de l'exemple précédent, et en général, les treillis qui nous interessent sont des treillis complets, car finis. Un ensemble E partiellement ordonné est un treillis complet si toute partie P de l'ensemble admet une borne supérieure, **sup** et une borne inférieure, **inf**. Une partie P admet une borne supérieure

si et seulement si l'ensemble des majorants de P (éléments de E plus grands ou égaux que tout élément de P) a un plus petit élément ; ce plus petit élément est la borne supérieure. De même, une partie P admet une borne inférieure si et seulement si l'ensemble des minorants de P (éléments de E plus petits ou égaux que tout élément de P) a un plus grand élément ; ce plus grand élément est la borne inférieure [SCH86].

Une hiérarchie de classes est un treillis complet si, pour tout ensemble de classes de la base, il existe une plus petite classe englobant toutes les classes de l'ensemble et il existe une plus grande classe contenue dans toutes les classes de l'ensemble. En général, on n'a pas de treillis complet, car la plupart des classes englobantes et des intersections des classes n'ont pas d'intérêt conceptuel pour justifier leur création. Dans le treillis des figures planes (Fig. 2.7) il est clair que le concepteur de la base n'est pas concerné par la classe des rectangles colorés, sous-classe des trapèzes colorés, mais seulement par celle des carrés colorés.

Dans des ordres partiels, une classe C peut avoir plusieurs sur-classes, D1, D2,...,Dn. Dans ce cas, l'ensemble d'éléments représenté par la classe C est inclus dans l'intersection non vide des ensembles d'individus représentés par ses sur-classes. Dans le cas où la sous-classe n'a pas de contraintes propres, la classe C inclut tous les éléments de l'intersection de ses sur-classes : pour appartenir à la sous-classe il est nécessaire et suffisant d'appartenir à toutes ses sur-classes. Dans le treillis précédent (Fig. 2.9), tout élément de *figure plane colorée* qui est un *carré*, appartient à la classe *carré coloré*. Dans d'autres systèmes, la sous-classe peut avoir des contraintes supplémentaires, étant donc un sous-ensemble de l'intersection des sur-classes ; c'est le cas de la hiérarchie des *objets utiles à l'homme* (Fig. 2.7), où la classe *delta_plane* décrit un sous-ensemble propre des *moyens de locomotion aériens utilisant la force éolienne*, cette sous-classe n'inclut pas des *ballons*, qui utilisent aussi la force éolienne pour se déplacer dans l'air.

La principale avantage d'utiliser une structure de treillis complet pour représenter une base de connaissances est la prise en compte des propriétés mathématiques d'une telle structure lors du raisonnement. Ainsi, par exemple, dans un treillis complet de classes, une instance a une seule plus petite classe d'appartenance. Les structures de treillis sont particulièrement utiles lorsque l'on veut faire explicites tous les éléments implicites dans une hiérarchie, par exemple pour la construction d'un graphe de types [MIS&88].

2.3.3. Arbre

Une structure d'arbre (orienté) est un graphe de classes dans lequel chaque classe a une seule sur-classe directe (sauf la classe *Objet* , appelée classe racine, qui n'a pas de sur-classe). Cette structure restreint les relations possibles entre deux classes : soit l'une est sous-classe directe ou indirecte de l'autre et l'ensemble d'individus qu'elle représente est inclus dans l'ensemble de la sur-classe, soit elles ne sont pas comparables par la relation de spécialisation.

Conceptuellement, cette structure de classes est utilisée pour représenter des domaines dans lesquels à chaque niveau de spécialisation d'une classe, les sous-classes intéressantes représentent des sous-ensembles mutuellement exclusifs. Par exemple, on peut spécialiser des animaux en deux classes : les vertébrés et les invertébrés ; ces deux classes sont exclusives, aucun animal ne peut être vertébré et invertébré en même temps, l'ensemble des vertébrés et celui des invertébrés sont disjoints (de même pour la division des vertébrés en mammifères, oiseaux et reptiles) (Fig. 2.10).

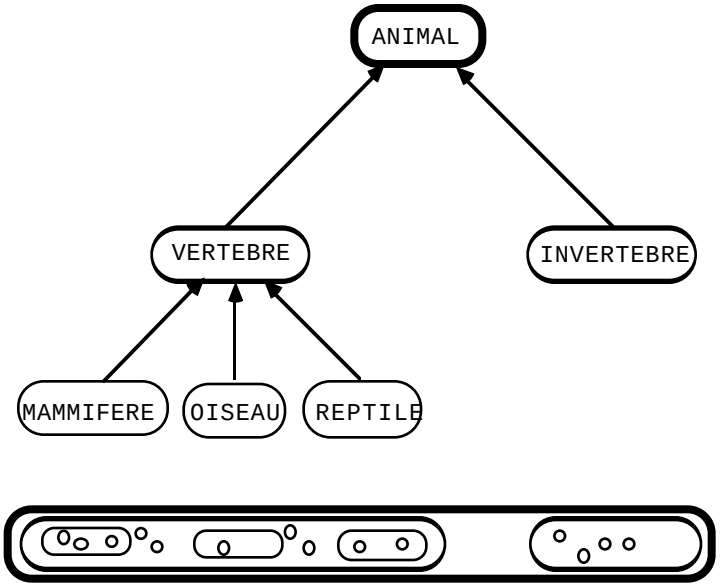


Fig. 2.10. Lorsque chaque classe a une seule sur-classe directe, le graphe de classe a la structure d'un arbre ; les ensembles d'instances représentés par des classes au même niveau sont disjoints.

Plusieurs langages orientés objets (p.ex. SIMULA et les premières versions de SMALLTALK) offrent seulement le mécanisme d'héritage simple. Cependant, les structures d'arbre générées par ces systèmes ne décrivent pas nécessairement des classes sœurs non disjointes.

2.3.4. Taxinomie

Le terme taxinomie a été utilisé pour décrire la classification des animaux ou des plantes dans une organisation hiérarchique. Une taxinomie est un arbre où dans chaque niveau on a une partition de l'ensemble des éléments, c'est-à-dire les sous-classes directes d'une classe de la taxinomie sont mutuellement exclusives et collectivement exhaustives : tout élément d'une classe C trouve sa place dans une et seulement une des sous-classes directes de C. Ainsi, par exemple, si dans l'arbre des animaux présenté ci-dessus on ajoute la classe *Poisson* et la classe *Amphibie* comme des sous-classes de la classe *Vertébré*, on a une taxinomie, car tout animal est un vertébré ou un invertébré et tout vertébré est un mammifère, un oiseau, un reptile, un poisson ou un amphibie (Fig. 2.11).

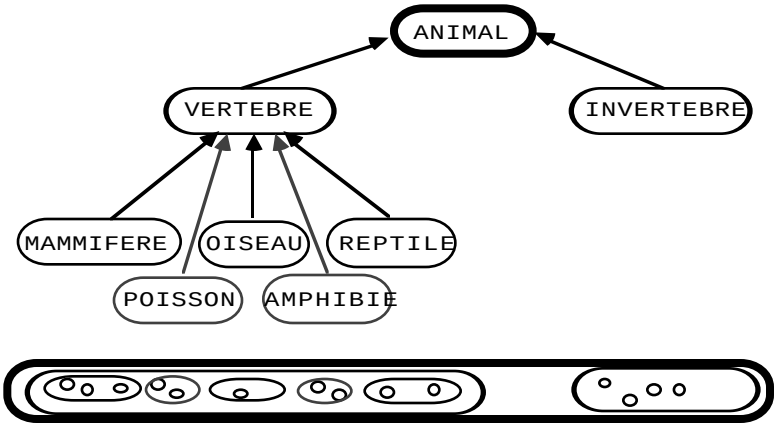


Fig. 2.11. Dans une structure taxinomique, chaque niveau établit une partition des individus. Les instances d'une telle structure se trouvent dans les classes du plus bas niveau, les feuilles de l'arbre.

2.4. Héritage

L'organisation des classes dans des hiérarchies de spécialisation permet de créer des classes complexes à partir de classes plus générales, en affinant la description générale. Une sous-classe se construit à partir d'une autre classe par ajout des attributs ou par restriction des attributs existant dans l'autre classe. Le mécanisme par lequel une classe récupère l'information héritée de ses sur-classes s'appelle **héritage**. L'héritage est donc un mécanisme de partage d'information par factorisation des attributs, qui facilite la création des objets qui sont presque égaux aux autres objets. Il réduit la nécessité de spécifier des informations redondantes et simplifie l'actualisation et la modification car l'information est localisée dans un seul endroit.

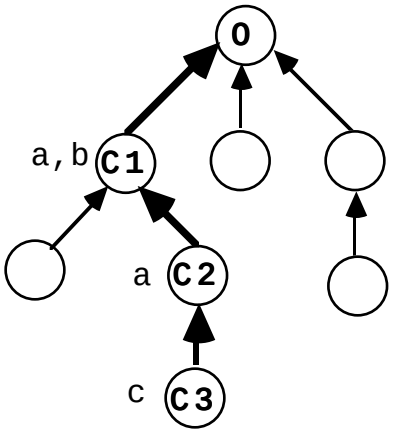
L'héritage permet d'inférer toutes les attributs d'une classe qui n'y sont pas données explicitement, en les cherchant dans les classes supérieures (ancêtres) selon l'ordre du plus affiné au plus général établi par les liens *sorte-de*. Ce mécanisme d'inférence revient à un algorithme de parcours du graphe des classes, appelé généralement **lookup**. [CAR89]. Pour trouver les attributs d'une classe C dans la base, l'héritage parcourt le sous-graphe composé par les classes qui sont sur-classes directes ou indirectes de la classe C ; l'ordre de parcours du graphe doit garantir qu'on commence par la classe C et qu'on respecte la **règle du plus spécifique** qui dit que pour un attribut donné, c'est toujours sa définition la plus spécifique qui est prise en compte. Il doit être clair que les attributs de la classe *Objet* sont toujours les derniers à être pris en compte.

Lorsque l'on parle d'héritage, il faut distinguer deux types d'héritage : l'**héritage de nom** et l'**héritage de valeur** [DUC&89], [DUC93]. Le premier est un héritage ontologique, structurel : la sous-classe hérite d'un attribut pour indiquer que les éléments de cette sous-classe ont cet attribut. L'héritage de valeur, de son côté concerne l'héritage de la valeur d'un attribut qui est déjà présente dans la sous-classe. Si le premier héritage concerne toute la structure de facettes de l'attribut, le deuxième concerne seulement les facettes de valeur fixe et défaut. Par la suite nous allons présenter le mécanisme d'héritage dans une structure arborescente, l'héritage simple, puis nous allons discuter les divers mécanisme d'héritage multiple qui permettent à une classes ayant plusieurs sur-classes d'hériter de leurs attributs. Pour cette discussion, nous n'allons pas faire la différence explicitement entre héritage de nom et héritage de valeur, car les parcours du graphe de classes proposées par ces mécanisme servent aussi bien à la récupération du nom que à celle de la valeur d'un attribut.

2.4.1. Héritage simple

Dans le cas où la structure de la base de connaissances est un arbre ou une taxinomie, une classe a une seule sur-classe directe et on parle d'**héritage simple**. Pour inférer des attributs d'une classe C, on suit un parcours linéaire des liens *sorte-de* à partir de la classe C : les sur-classes de la classe sont totalement ordonnées de la plus affinée, C, à la plus générale, *Objet.*

Par exemple, soit la hiérarchie suivante où C3 est sous-classe de C2, qui est sous-classe de C1, une sous-classe de la classe Objet. Dans C1 on définit les attributs “a” et “b” , dans C2 on affine “a” et dans C3 on définit “c”. Alors, les instances de la classe C3 sont représentées par trois attributs: l’attribut “c” défini dans leur classe d’appartenance C3, l’attribut “a” hérité de la sur-classe C2 et l’attribut “b” hérité de C1 (aucun attribut n’est hérité de la classe Objet).



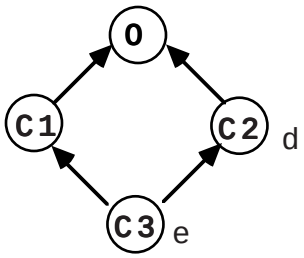
Plusieurs LOO, tels que SIMULA et les premières versions de SMALLTALK [GOL&83], n'offrent que l'héritage simple.

2.4.2. Héritage multiple

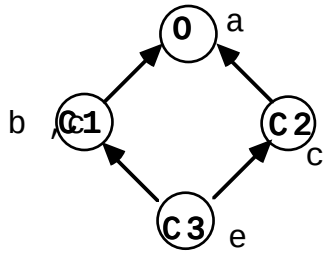
Lorsque dans la hiérarchie des classes une classe peut avoir plusieurs sur-classes, on parle de spécialisation multiple. Le mécanisme qui permet d’hériter des attributs de ces sur-classes dans la sous-classe est dit **héritage multiple**. De même que pour l’héritage simple, le but ici est de partager l’information en réduisant la redondance ; une sous-classe, en tant que spécialisation de plusieurs classes, hérite de l’union des attributs de ses sur-classes et peut ainsi partager des informations avec elles [STE&85].

Dans ce type de structure, les ancêtres d’une classe ne forment pas un ordre total, mais seulement un ordre partiel. Le parcours des sur-classes avec cet ordre partiel ne pose pas de problème si, pour aucune classe C, on n’a d’attributs de même nom définis ou modifiés dans deux sur-classes différentes.

Dans l’exemple suivant, on a l’ordre partiel défini par: C3 < C1 < O et C3 < C2 < O; les classes C1 et C2 ne sont pas comparables. Néanmoins, pour déterminer les attributs hérités par une instance de C3, il suffit de voir que le seul attribut commun à C1 et C2 est l’attribut “a”, hérité, dans les deux cas, de la même classe O.
On a donc l’ordre total C3 < (C1 et C2) < O et une instance de C3 possède l’attribut “a” défini dans O, les attributs “b” et “c” de C1, l’attribut “d” hérité de C2 et l’attribut “e” défini dans C3.



Le problème se pose lorsque deux sur-classes directes d’une même classe ont des attributs de même nom; on parle ici d’un conflit d’héritage. Si dans l’exemple précédent on ajoute à la classe C1 un affinement de l’attribut “a” défini dans O et on change le nom de l’ attribut de C2 par “c”, il y a deux conflits à résoudre pour l’héritage d’attributs dans la classe C3. D’une part doit-elle hériter de la définition de l’attribut “a” donnée initialement dans O et héritée par C2 ou bien de l’affinement de cet attribut fait dans C1 ? D’autre part, doit-elle hériter de l’attribut “c” défini dans C1, doit-elle prendre la définition donnée dans C2 ou garder les deux ?



Pour la première question, les algorithmes d'héritage multiple existants respectent tous la règle du plus affiné énoncée précédemment et ils garantissent que la classe C3 hérite de l'attribut "a" affiné dans la classe C1 et non pas la définition plus générale donnée dans O.

La deuxième question a été traitée de différentes façons, guidées souvent par des aspects algorithmiques plutôt que par une analyse conceptuelle de la signification de ce type de conflits. Le parcours du graphe d'héritage pour résoudre ce problème a été le sujet de plusieurs études [DUC&89], [FER88], [CHO&88]. Par la suite nous allons présenter certaines de ces propositions : l'approche linéaire, l'approche graphique [CAR89] et l'approche circonstancielle. Pour un étude plus détaillé et des solutions plus élaborées, voir [DUC93].

L'approche linéaire

Les algorithmes suivant cette approche font un parcours du graphe d'héritage d'une classe et linéarisent ses sur-classes en une chaîne en construisant un ordre total à partir de l'ordre partiel établi par la relation de spécialisation *sorte-de*. Ainsi, la classe n'hérite que d'un seul des attributs conflictuels, celui de la plus petite classe selon l'ordre total construit.

Dans cette catégorie certains systèmes comme SHIRKA [REC88] font un parcours du graphe en profondeur d'abord, tandis que d'autres comme MERING [FER88] le font en largeur d'abord.

Les algorithmes basés sur le parcours en profondeur d'abord créent une liste linéaire des sur-classes de la classe en commençant avec la première sur-classe directe, la plus à gauche, et en continuant en profondeur ; ils suppriment ensuite toutes les occurrences dupliquées, en vérifiant la règle du plus affiné. Dans le graphe suivant (Fig. 2.12), cet algorithme linéarise les sur-classes de C4 en la liste : C4, C3, C1, O, C2, O ; puis il élimine la première occurrence de la classe O ; ainsi C4 hérite de l'attribut "a" de la classe C3 et l'attribut "b" de la classe C1.

Le parcours en largeur utilisé par MERING [FER84] produit pour l'exemple (Fig. 2.12) la liste d'héritage C4, C3, C2, C1, O, selon laquelle la classe C4 hérite de l'attribut "a" de la classe C3 et l'attribut "b" de C2.

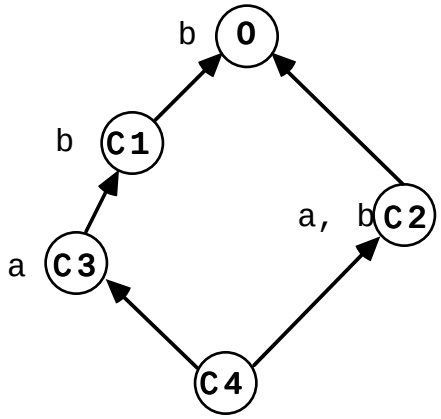


Fig. 2.12. L'approche linéaire pour résoudre le conflit d'héritage d'un attribut, construit un ordre total, soit par profondeur : C4,C3,C1,O,C2,O, soit par largeur : C4,C3,C2,C1,O.

L'approche linéaire fait dépendre l'héritage des attributs d'une classe de l'algorithme utilisé, c'est-à-dire qu'il résout un conflit conceptuel avec une solution

purement calculatoire, produisant parfois des comportements inattendus. Les contraintes que pose cette approche, qui a été néanmoins la plus utilisée, vont encore plus loin, car un même algorithme peut donner des résultats différents si on change l'ordre dans lequel sont définies les sur-classes directes d'une classe (fait qui, ne devrait pas avoir de conséquences sur la façon d'hériter des attributs).

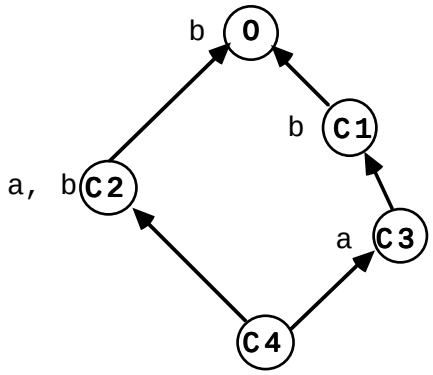


Fig. 2.13 : La linéarisation du chemin d'héritage d'une classe fait dépendre le résultat de l'ordre dans lequel les classes sœurs (du même niveau) sont déclarées. Ici l'héritage par profondeur pour C4 donne la séquence : C4,C2,O,C3,C1,O, qui est différente de celle présentée dans la Fig. 2.10.

Par exemple, pour le graphe précédent, on peut avoir un graphe qui représente la même connaissance, en inversant l'ordre de définition de C1 et C2, sous-classes de O (Fig. 2.13). Les algorithmes d'héritage indiquent que la classe C4 doit hériter les attributs "a" et "b" de la classe C2 au lieu d'hériter "a" de C3 et "b" de C1 en profondeur, ou bien l'attribut "a" de C3 et "b" de la classe C2 en largeur.

L'approche graphique

Dans l'approche graphique, une classe hérite de tous les attributs de ses sur-classes. Pour résoudre les conflits de noms, les attributs conflictuels sont hérités avec l'information du chemin des sur-classes suivi à partir de la classe où ils ont été créés. Cette approche est prise par COMMON-OBJECTS et par d'autres langages orientés objet.

Une variante à cette approche est proposée par [CHO&88]. Ils proposent deux façons de traiter les conflits de noms : soit la classe hérite de tous les attributs conflictuels avec l'information de leurs chemins d'héritage, soit elle n'hérite que d'un attribut. La deuxième possibilité sert dans les cas où on peut vérifier, en regardant le chemin d'héritage, que, même s'ils viennent de sur-classes différentes, les attributs ayant le même nom sont déclarés initialement dans un ancêtre commun. Quoique dans cette approche la solution des conflits ne dépende pas de la façon de parcourir le graphe, la source sémantique du conflit n'est pas traitée non plus.

L'approche circonstancielle

Dans cette dernière approche on peut classer tous les algorithmes qui, en acceptant que le traitement des conflits dans l'héritage multiple est circonstanciel et non généralisable, essaient de trouver une solution moins rigide que les précédents.

D'autre part, Wegner [WEG87] affirme que, dans le cas de conflits, il doit y avoir une interdépendance comportementale entre les attributs du même nom et il propose donc de définir dans la sous-classe l'attribut conflictuel avec une fonction reliant tous les attributs de même nom hérités. Le cas extrême se trouve dans les dernières versions de SMALLTALTK [GOL&83] où c'est l'utilisateur qui doit résoudre les conflits en disant explicitement quel est l'attribut dominant.

2.4.3. Multi-instanciation

Les sections précédentes montrent les différentes façons de résoudre le conflit d'héritage multiple. L'héritage multiple est le mécanisme d'héritage des attributs d'une classe ayant plusieurs sur-classes. Une classe peut avoir plusieurs sur-classes seulement si dans un niveau supérieur du graphe il existe deux classes sœurs (C1 et C2) non disjointes, c'est-à-dire dont les ensembles potentiels d'individus ont une intersection non vide.

Ainsi, le problème conceptuel qu'on essaie de résoudre avec l'héritage multiple est le traitement des éléments de l'ensemble intersection de deux classes non disjointes (C1 et C2) (Fig. 2.14).

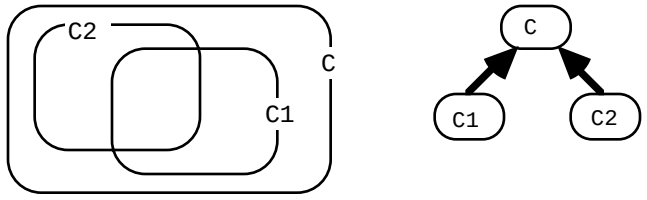


Fig. 2.14. Le problème de l'héritage multiple ne peut se présenter que lorsque deux classes sœurs décrivent des ensembles non disjoints d'individus.

La solution de ce problème par héritage multiple revient à créer une classe, C3, sous-classe de C1 et C2. Cette sous-classe représente l'intersection de ses sur-classes, ou bien un sous-ensemble de cette intersection (Fig. 2.15).

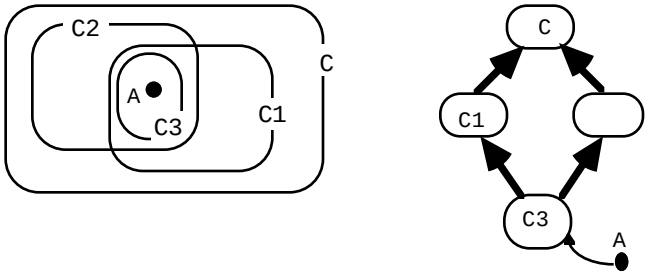


Fig. 2.15. La multi-spécialisation, c'est-à-dire la création d'une classe ayant plus d'une sur-classe demande des algorithmes d'héritage multiple, qui peuvent avoir à traiter des conflits d'héritage d'attributs du même nom.

Néanmoins, il peut y avoir des cas où la sous-classe (C3 dans l'exemple Fig. 2.15) n'a pas de raison d'être en tant que telle, par exemple lorsqu'elle représente l'ensemble complet des éléments de l'intersection, car sa création n'ajoute aucune information importante au graphe. Dans ces cas, on voudrait pouvoir exprimer le fait qu'une instance appartient aux deux sur-classes (C1 et C2) sans avoir à créer la classe intersection. On parle donc de **l'instanciation multiple** : une instance peut être attachée (par le lien est_un) à plusieurs classes qui ne sont pas liées entre elles par la relation de spécialisation**.** Dans l'exemple précédant, au lieu de créer la classe C3, l'objet A peut être instancié par la classe C1 et par la classe C2 ; il hérite de tous les attributs de ces deux classes (Fig. 2.16).

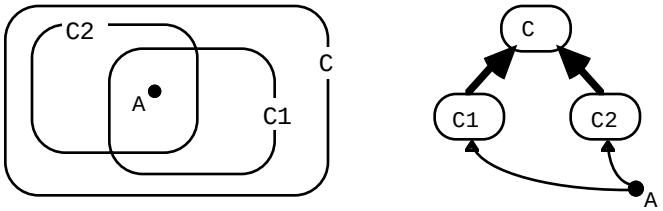


Fig. 2.16. La multi-instanciation consiste à attacher une instance à plusieurs classes (qui décrivent des ensembles non disjoints d'instances).

La solution par multi-instanciation, utilisée par OWL II [MAR79] est beaucoup moins utilisée que celle de l'héritage multiple ; elle présente aussi des problèmes. D'une part, elle exige des mécanismes puissants de contrôle de cohérence pour interdire qu'une instance soit attachée à deux classes incompatibles (deux classes disjointes ou l'une sous-classe de l'autre). D'autre part, le problème du traitement des attributs ayant le même nom n'est pas résolu, mais transporté à l'instance au moment de sa création.

Quoique la discussion présentée ci-dessus montre l'importance donnée au problème de l'héritage multiple et de la multi-instanciation, la question conceptuelle de base n'est presque jamais posée :

Quand a-t-on des classes non disjointes dans une hiérarchie de classes ?

Cette question est abordée dans le chapitre suivant où nous allons essayer de la résoudre avec la notion de points de vue.

2.5. Objets composites

Comme nous l'avons présenté dans la section (§ 2.2.2), une classe (et ses instances) peut avoir des attributs décrivant une propriété, une relation ou un composant. Dans cette partie, nous présentons les différentes interprétations possibles de la relation de composition et nous montrons les diverses approches prises par les systèmes centrés objets qui manipulent des objets composites. Nous avons dit, informellement, qu'un objet composite est un objet complexe pour lequel on peut identifier des parties ; ces parties sont décrites dans la classe de l'objet comme des attributs de type *composant*. La relation de composition "partie-de" peut être interprétée de différentes façons. [WIN&87] ont identifié six catégories différentes de relations de compositions (Fig. 2.17) :

Relation	Exemple	Fonctionnalité	Homogénéité	Séparabilité
Composant / tout	moteur- voiture	+	-	+
Membre / collection	arbre - forêt	-	-	+
Portion / masse	grain - sel	-	+	+
Matière / objet	bois - armoire	-	-	-
Trait / activité	payer - faire des courses	+	-	-
Lieu / Aire	oasis - désert	-	+	-

Fig. 2.17. Les 6 types de relations de compositions se distinguent par trois aspects : si la relation entre le tout et les parties est fonctionnelle (+) ou pas (-); si les parties sont du même type que le tout (+) ou pas (-) et s'ils sont séparables du tout ou pas.

la relation composant (partie) - tout, par exemple entre le moteur et la voiture ;

la relation membre-collection, entre un arbre et la forêt ;

la relation portion-masse, entre un grain de sel et le sel ;

la relation matière-objet existante entre bois et armoire ;

la relation étape-activité, qui lie le fait de payer à l'activité complète de faire des courses ;

la relation lieu-aire entre, par exemple, une oasis et le désert.

Ces relations de compositions diffèrent en trois aspects :

fonctionnalité : si la relation entre le tout et les parties est fonctionnelle ou pas, c'est-à-dire si chaque partie réalise ou sert à réaliser une sous-tâche de la tâche réalisée par le tout.

homogénéité : si les parties sont du même type que le tout ;

séparabilité : si les parties sont séparables du tout.

La relation de composition est une relation transitive, non réflexive et antisymétrique :

- non réflexive car un objet n'est pas partie de lui-même[1],
- asymétrique car si a est-partie de b, alors b n'est pas partie de a
- transitive car si a est-partie de b et b est-partie de c, alors a est-partie de c.

Cette dernière propriété de transitivité n'est pas toujours valide. Par exemple, on peut dire [WIN&87] :

Le carburateur fait partie du moteur (relation composant - objet entier)
Le moteur fait partie de la voiture (relation composant - objet entier)
---
alors, le carburateur fait partie de la voiture (relation composant - objet entier)

Mais, on ne peut pas dire :

Le bras de Paul fait partie de Paul (relation composant - objet entier)
Paul fait partie du département de philosophie (relation membre - collection)
---
alors, le bras de Paul fait partie du département de philosophie.

En général, la propriété de transitivité est garantie lorsque l'on travaille avec un seul et même type de relation ; dans les autres cas elle peut poser des problèmes.

La plupart des systèmes de représentation à objets manipulant des objets composites ne font pas la distinction explicite entre ces différents types de relation de composition. Ces systèmes peuvent se classer en trois groupes. Le premier groupe utilise la définition structurelle d'objet composite : un tout et des composants qui sont des objets à part entière [BOB&83], [MAS&89]. Au deuxième groupe appartiennent les systèmes qui offrent des mécanismes généraux de représentation, où le concepteur de la base donne une sémantique à la relation [DUG91], [FOX&89], [FOR&86], [ESC&90]. Enfin, le dernier groupe, tout en restant proche de l'approche composant - objet entier, favorise les relations et connexions entre les composants et avec le tout ; ils centrent leur travaux sur le maintien de la cohérence par propagation de contraintes entre objets [BOR81], [SUS&80], [DAV87]. Une approche un peu différente est celle proposé par [NAP92] qui définit la relation de composition à partir de la relation plus générale de subsomption.

---

[1] Certains systèmes [MAR&92] considèrent que la composition est une relation réflexive, un objet pouvant être composant de lui-même. Nous prenons ici l'approche plus courante selon laquelle un objet n'est pas composant de lui-même.

2.5.1. Relation de composition fixe et prédéfinie

Certains systèmes donnent une sémantique particulière et unique à la relation de composition (correspondant à la décomposition physique d'un objet) qui entraîne les contraintes suivantes :

° La décomposition d'un objet composite forme une structure arborescente, c'est-à-dire qu'à un même niveau de décomposition tous les composants sont disjoints.

° L'existence des composants dépend de celle de l'objet composite, **dépendance existentielle.**

° Un objet ne peut faire partie que d'un seul objet composite, **dépendance exclusive.**

A ce groupe appartiennent, entre autres, LOOPS [BOB&83], un langage hybride d'objets et de règles, et YAFOOL [DUC88], système fondé sur la théorie des prototypes. Ils décrivent l'objet composite par des moyens spécifiques : dans LOOPS un objet composite est modélisé par une classe, instance de la méta-classe "Composite object", et ses composants sont représentés par les variables d'instance de cette classe ; YAFOOL utilise deux attributs dans l'objet composite pour décrire ses composants et leurs types (Fig. 2.18). De plus, ils fournissent des outils particuliers de création et de manipulation d'objets composites. Ainsi, l'instanciation des composants d'un objet composite se fait automatiquement en même temps que celle de l'objet.

```
(defmodele Automobile
    (compose-de (value Carrosserie Moteur (4 Roue
   (lien-de-composition
         (value la-Carrosserie le-Moteur les-Roues
```

```
(defmodele Carrosserie
    (compose-de (value Capot (3 Porte) Toit))
   (lien-de-composition
       (value le-Capot  les-Portes le-Toit)))
```

```
(defmodele Capot
    (couleur (valeur rouge-sang)))
```

Fig. 2.18. Définition de l'objet composite Automobile, d'un de ses composants, Carrosserie, et d'un composant de celui-ci, Capot, en YAFOOL [MAS&89].

Le système ORION [KIM&87], [KIM&89] est à la frontière entre ce premier groupe et les systèmes offrant une relation générique. En effet, il pourvoit des primitives de manipulation et de gestion de la relation "partie-de", mais cette relation est moins restrictive que dans les systèmes précédents : les dépendances exclusives et existentielles sont optionnelles et doivent être établies ou enlevées explicitement dans chaque relation entre un composant et l'objet composite.

2.5.2. Systèmes avec une relation générique

Des systèmes comme OBJLOG [DUG88], [DUG91], SRL [FOX&86], OTHELO [FOR&89] et SHOOD [ESC&90] offrent des outils pour définir différents types de relations entre les objets de la base ; il s'agit en général de relations binaires qui sont établies au niveau des classes mais qui peuvent être modifiées au niveau des instances pour traiter des cas d'exception.

Dans OBJLOG, un langage à objets écrit en Prolog, une relation est décrite par un attribut dans lequel on spécifie le domaine des valeurs de la relation ainsi que les propriétés partagées par les éléments participant à la relation. Dans le cas de la relation de composition, dans l'attribut "partie-de" du composant, le concepteur indique la classe d'appartenance de l'objet composite et il établit la liste des "slots" qui vont être partagés entre ce composant et l'objet composite (héritage sélectif d'attributs) (Fig. 2.19).

SRL et SHOOD représentent les relations comme des objets à part entière, groupant ainsi toute l'information propre à la relation dans un même schéma. SRL attache aux relations des propriétés mathématiques (réflexivité, symétrie, etc.). SHOOD définit, de plus, des opérations logiques telles que l'union, l'intersection, la composition et associe une dépendance (exclusive, partagée, nulle, existentielle ou spécifique) entre les objets intervenant dans la relation.

Dans OTHELO, les relations sont décrites par une classe qui contient, entre autres, les méthodes pour maintenir la cohérence de la base ; à la différence des autres systèmes de ce groupe, OTHELO modélise uniquement des relations maître-esclave, c'est-à-dire celles ayant un objet influant et un objet dépendant (par exemple : la relation de composition et l'héritage).

```
Automobile
    sorte-de objet.nil
   <compose-de,nil>
     statut structurel
     domaine-ref <Carrosserie.Moteur.Roue.nil
     heritage
        <<Carrosserie,<couleur,Carrosserie.ni
```

```
Carrosserie
    sorte-de objet.nil
    <compose-de,nil>
       statut structurel
       domaine-ref <Capot.Porte.Toit.nil>
   <couleur,nil>
       domaine-ref <ident>
```

Fig. 2.19. Description en OBJLOG des classes Automobile (d'objets composites) et Carrosserie (un de ses composants). Ils partagent l'attribut couleur [MAS&89].

2.5.3. Approche parties-contraintes-relations

Les systèmes de ce groupe, tels que THINGLAB [BOR81], CONSTRAINTS [SUS&80] et VIEWS [DAV87] utilisent une représentation proche des réseaux sémantiques, dans laquelle les objets (parties) sont liés par des relations génériques ; ces objets et ces relations sont restreints par un ensemble de contraintes qui peuvent servir pour inférer des connaissances par propagation de valeurs connues (Constraints et ThingLab (Fig. 2.20)) ou pour limiter la structure d'un groupe de parties et de relations (Views).

En ce qui concerne la relation de composition, c'est au concepteur de définir la sémantique de la relation et de lui attacher des contraintes appropriées.

Bien que les deux derniers groupes de systèmes permettent la modélisation de toute une variété de relations, ils n'ont pas de méthodes prédéfinies pour une relation particulière, telle que la composition.

```
Classe MidPointLine
  Superclasses  Geometric object
  Part Descriptions Line:  a line
                              midpoint: a po
  Constraints  midpoint = (line point1 + li
     midpoint <- (line point1 + line point2)
     line point1 <- midpoint * 2 - line poin
     line point2 <- midpoint * 2 - line poin
```

Fig. 2.20. Représentation d'une classe en ThingLab. Dans la classe MidpointLine le point du milieu doit être à mi-chemin des deux extrémités. Cette contrainte peut se satisfaire de trois façons, décrites par les méthodes énoncées sous la règle ; l'ordre des méthodes indique l'ordre des préférences pour leur déclenchement [BOR81]

2.5.4. La co-subsomption : subsomption + composition

Comme nous avons présenté dans § 1.3.5., la relation de subsomption est une relation d'ordre partiel entre deux concepts : un concept A subsume un concept B si la description de A est plus générale que celle de B (l'ensemble d'objets décrits par A inclut celui d'objets décrits par B).

La relation de subsomption permet de mettre en correspondance des concepts décrits en terme de conjonctions ou de disjonctions de littéraux. Dans les représentations à objets, les objets sont définis par une conjonction de littéraux (propriétés). Une restriction de la subsomption pour prendre en compte les relations d'ordre entre ces objets est proposée par Napoli [NAP&92]. Cette relation, appelée **O-subsomption** est définie à partir de

- un ensemble d'objets, O
- un ensemble de propriétés, P
- un ensemble de relations d'ordre partiels $\geq_{pi}$ dépendant de chaque propriété p_i et de son type, $\{\geq_{pi}, pi \in P\}$
- un ensemble de propriétés déductibles Q. La valeur d'une propriété $q \in Q$ est calculée à partir des règles de partages spécifiques, S.

Soit $p_i(O_j)$ la propriété p_i associée à l'objet O_j. Alors un objet O_1 défini par une conjonction de propriétés $(p_1(O_1) \wedge p_2(O_1) \wedge \ldots \wedge p_n(O_1))$ **O-subsume** un objet O_2 contenant la conjonction $(p1(O2) \wedge p2(O2) \wedge \ldots \wedge pn(O2))$ si et seulement si $p_1(O_1) \geq_{p1} p_1(O_2)$, $p_2(O_1) \geq_{p2} p_2(O_2)$, …, $p_n(O_1) \geq_{pn} p_n(O_2)$.

De plus, si O_1 O-subsume O_2, alors pour tout $q \in Q$ la valeur $q(O_2)$ est déduite de la valeur $q(O_1)$ en utilisant la règle S.

Cette relation de O-subsomption, peut être étendue pour modéliser la relation de composition. En effet, la O-subsomption permet la modélisation des relations d'ordre entre objets, et la relation de composition "composé de" (ainsi que la relation inverse "partie-de") est une relation d'ordre partiel entre un composant et le tout si l'on accepte qu'un objet fasse partie de lui-même. La relation de composition par subsomption **co-subsomption** proposée par Napoli [NAP92] est obtenue à partir de la O-subsomption en ajoutant à l'ensemble de propriétés P la propriété "composé de" qui mémorise les parties de l'objet et en modifiant les règles de partage de valeurs S pour faire une diffusion adéquate des valeurs d'attributs entre le tout et ses composants. La sémantique accordée à la co-subsomption est la suivante :

Un objet O_1 **co-subsume** un objet O_2 si O_1 est ou décrit une partie de O_2.

La relation de co-subsomption est spécialement conçue pour l'élaboration d'objets composites par assemblage de parties plus simples ; pour le moment elle est exclusivement appliquée à la synthèse de structures moléculaires (système YCHEM). Cette approche "constructive" explique le choix de définir la co-subsomption comme la relation "est-partie-de" et non pas comme la relation "est-composé-de".

La description de la relation de composition à partir de la relation de subsomption favorise l'utilisation des mécanismes de raisonnement liés à la subsomption, tels que la classification, pour raisonner avec des objets composites.

Le système TROPES, que nous allons décrire dans le chapitre 5, se place dans l'approche stricte qui établit une relation de composition fixe, prédéfinie, différente de la relation de spécialisation. Pour ce type de systèmes, on peut développer des mécanismes de raisonnement spécialisés, faisant des inférences à partir de la sémantique bien définie des relations de composition. Dans le chapitre 3 nous proposons une relation de composition définie en termes de points de vue.

2.6. Comparaison avec d'autres approches

La représentation à objets a été très souvent confondue avec la programmation orientée objet, bien que cette dernière concerne les langages d'implémentation et la représentation à objets la modélisation déclarative du monde. Dans cette dernière partie du chapitre nous détaillons les différences entre ces deux termes, ainsi que les différences, moins nettes mais aussi importantes, entre les représentations à objets ayant l'approche classe / instance et les représentations à objets ayant l'approche par prototypes. Enfin, nous faisons une comparaison avec les logiques terminologiques, qui tout en restant des successeurs des réseaux sémantiques, ont évolués vers une représentation centrée concepts qui ressemble à la représentation centrée classes - instances de notre approche.

2.6.1. Représentation à objets - langage orienté objets

Un système à base de connaissances prétend résoudre un problème en effectuant des inférences sur des connaissances spécifiques à un domaine. Les connaissances de ce domaine sont décrites sous forme déclarative, à l'aide d'objets structurés. La sémantique donnée à ces objets est guidé par la sémantique du monde à modéliser et non pas par des considérations opérationnelles d'implémentation - on parle ici d'une sémantique externe qui fait le lien entre le monde réel et la syntaxe décrivant les éléments du modèle. C'est grâce à cette sémantique, formalisée dans le modèle que le système peut faire des inférences sur le modèle, puis les transférer aux objets du monde.

Les langages de programmation orientée objets sont des outils de construction de programmes fondés sur un modèle procédural de la connaissance. Bien qu'ils puissent être utilisés pour "construire" des systèmes de représentation de connaissances, ils ne sont pas adéquats pour représenter directement la connaissance, car ils n'offrent pas une sémantique formelle, pour les objets et les classes ; cette manque de sémantique limite l'expressivité du système (on ne peut pas représenter certains aspects des objets) et permet aux procédures utilisateur de modifier les structures de données rendant difficile la description et la compréhension du système. Dans la programmation orientée objet la sémantique est subordonnée à la définition opérationnelle du système [PAT90].

Les classes dans la programmation orientée objets ont un pouvoir d'expression limité : un objet peut être membre d'une classe seulement si cette classe (ou une de ses sous-classes) est mentionnée explicitement dans l'objet ; on ne peut pas utiliser une classe comme un "pattern" pour reconnaître. Dans certains langages comme SMALLTALK [GOL&83] et LOOPS [STE&85], la classe peut avoir une structure tout à fait différente de celle des instances : elle peut avoir, par exemple, des propriétés et contraintes qui concernent la classe comme un tout. Dans ces cas il est impossible non seulement de dire si une instance appartient à une classe ou pas, mais de vérifier qu'un lien d'appartenance déjà établi est correct.

De plus, la plupart des langages orientés objets gèrent seulement des contraintes existentielles et du typage. Certains permettent la définition de valeurs par défaut mais en général la sémantique du défaut n'est pas claire. Par exemple une valeur par défaut peut être utilisée pour représenter une valeur typique ou bien pour propager aux instances des modifications faites à une classe. D'autre part, plusieurs langages orientés objet permettent de masquer des contraintes d'une classe dans ses instances ou dans ses sous-classes pour traiter des exceptions ; cette liberté rend plus difficile la production de conclusions sur les instances d'une classe à partir de l'information de la classe. Bien que les problèmes des défauts et des exceptions soient aussi présents dans certains systèmes de représentations (§ 2.6.3), la sémantique de ces systèmes devrait permettre d'identifier les cas conflictuels ou ambigus.

Les problèmes de l'héritage multiple d'attributs de même nom, présentés dans (§ 2.4.2), sont d'autant plus graves dans les langages orientés objet que l'héritage y est une notion opérationnelle qui n'a pas une justification sémantique de base ; il est contrôlé par des informations qui ne sont pas du niveau de la représentation. Dans un système de représentation centré objet, l'héritage n'est qu'une description dérivée de la façon dont l'information circule entre des classes liées ; ce n'est pas une primitive du système spécifiée en termes opérationnels. Les problèmes conceptuels de conflit d'héritage peuvent (et doivent) ici être traités au niveau de la sémantique accordée à la spécialisation des classes et aux objets de la base, les détails d'implémentation restant complètement cachés. C'est l'approche que nous allons prendre dans le chapitre suivant pour discuter le problème de l'héritage multiple.

En ce qui concerne le raisonnement et les explications, la sémantique d'une représentation de connaissances détermine les déductions valides du système (c'est-à-dire une requête satisfaite dans tous les mondes possibles). Les inférences sont faites par des mécanismes spécialisés tels que la classification et le filtrage, qui ont des étapes de déductions valides et explicables. Dans les langages orientés objet, par contre, le raisonnement est câblé dans des programmes ou bien contrôlé par l'envoi de messages entre des objets. Ces objets sont des boîtes noires, qui encapsulent leur propriétés et leur comportement interne ce qui rend difficile la génération d'explications du raisonnement suivi. En effet, le système global ne peut pas voir les structures et méthodes qui interviennent dans un raisonnement.

Enfin, dans les représentations à objets, le modèle conceptuel que peut créer l'utilisateur est plus proche de sa propre représentation du problème que dans les langages orientés objet où le développeur doit définir aussi bien des objets du monde représenté que ceux concernant l'implémentation (fenêtres de dialogue, objets événements, etc). L'équivalence entre le modèle "réel" et sa représentation présente dans les RCO facilite la communication entre le système et l'utilisateur, tant pour que l'utilisateur donne la spécification des objets du monde que pour expliquer le raisonnement suivi par le système.

2.6.2. Approche classe /instance - approche par prototype

Comme nous l'avons présenté auparavant, les système de représentation à objets ayant l'approche classe / instance représentent le monde avec deux types d'objets : les classes qui décrivent des s communes à une catégorie d'objets et les instances qui représentent des individus particuliers d'une classe. Dans la plupart de systèmes utilisant cette approche, le schéma d'une classe établit les conditions nécessaires et suffisantes d'appartenance à la classe (§ 2.2.4.)

Par opposition à cette approche, les représentations par prototypes décrivent les entités du monde à l'aide d'un seul type d'objet : le prototype. Un prototype est un individu représentatif d'une catégorie d'objets, qu'il décrit implicitement ; ainsi, le prototype possède les propriétés les plus remarquables chez les membres de la catégorie ; ces propriétés sont des informations par défaut et non pas des informations définitionnelles dans la mesure où un autre membre particulier de la même catégorie peut ne pas les partager.

L'approche par prototypes prend ses sources dans des études en psychologie qui affirment que l'être humain raisonne mieux à partir des exemples spécifiques qu'il garde dans sa mémoire ; pour ajouter une nouvelle connaissance, un nouveau concept, il cherche dans sa mémoire un prototype semblable et en fait une copie en modifiant ou annulant les propriétés que le nouveau concept ne partage pas avec le prototype copié [MAS&89]. A la différence de l'approche classe / instance qui exige d'avoir la définition de la classe avant de créer ses instances, l'approche par prototype permet de créer les concepts individuels indépendamment de toute abstraction (qui serait la classe) [LIE86]. Cette propriété fait des prototypes des bons candidats pour représenter les connaissances des domaines dans lesquels l'identification a priori de tous les attributs essentiels d'un concept n'est pas facile.

Cependant, la nature par défaut des propriétés des prototypes empêche toute capacité définitionnelle et toute déduction à caractère sûr dans la base [BRA85]. Dans l'approche classe / instance, une classe établit des conditions nécessaires pour ses instances ; ainsi, **toute** instance I de la classe C possède **toutes** les propriétés présentes dans C et satisfait **toutes** leurs conditions. De même, toute sous-classe de la classe C hérite de toutes ses propriétés et contraintes. Un prototype, par contre, possède des propriétés qui sont vraies seulement pour les individus de la catégorie (toute copie du prototype) pour lesquels elles ne sont pas explicitement annulées. Cette facilité d'annulation, offerte pour

traiter les exceptions, peut être utilisée sur toutes les propriétés du prototype : aucune propriété n'a un caractère nécessaire ; aucun ensemble de propriétés n'est définitionnel (nécessaire et suffisant).

Ainsi, par exemple on peut dire qu'une pierre est un éléphant excepté qu'il n'a pas de trompe, il n'est pas vivant, il n'a pas de pattes, … [BRA85]. Une pierre peut donc être construite à partir d'une **copie** du prototype éléphant et une annulation de toutes ses propriétés ; cette construction est reflétée dans la base de connaissances par un lien "copie-de" liant la pierre avec l'éléphant. A la différence des liens sorte-de et est-un de l'approche classes / instance, le lien "copie-de" des prototypes n'a pas une sémantique précise ; il prétend représenter une sorte de ressemblance, mais cette ressemblance n'est pas bien définie et elle peut être aussi faible que celle existant entre une pierre et un éléphant. Aucune connaissance déduite pour un prototype ne peut être "héritée" par ses copies (les prototypes voisins).

Malgré la structure apparente, les prototypes d'une base doivent être traités comme des concepts primitifs, individuels. Pour cette raison, une représentation par prototypes ne permet d'exprimer aucun fait universel, aucune vérité valide pour tous les individus d'une catégorie [BRA85]. D'autre part, aucune connaissance déduite pour un prototype ne pouvant être "héritée" par ses copies, cette représentation ne peut utiliser des mécanismes de raisonnement aussi puissants que le filtrage et la classification .

En plus, la description d'un groupe d'objets dans la base dépend de l'ordre d'insertion de ces objets, car la localisation d'un individu se fait par comparaison avec d'autres individus déjà stockés. Voyons un exemple un peu caricatural : supposons qu'un individu connaisse un seul animal, un *aigle*. Un jour, il fait connaissance d'un *dinosaure* ; en cherchant dans sa mémoire il trouve des attributs communs à l'aigle et au dinosaure : les deux sont des êtres vivants qui se reproduisent et qui se déplacent ; il trouve aussi des différences : leur façon de se déplacer, leur nourriture, leur durée de vie, etc. Pour représenter le dinosaure, il fait donc une copie de l'aigle et il modifie cette copie pour prendre en compte les différences remarquées. Si après on lui montre un *canard*, il fera une nouvelle copie de l'aigle, cette fois en masquant moins de attributs car il y a plusieurs vraisemblances entre les deux oiseaux. Le modèle final (Fig. 2.21) dépend de l'ordre d'insertion des instances. Si notre individu avait appris le dinosaure en premier, puis le canard puis l'aigle, le modèle final (Fig. 2.22) aurait présenté une structure linaire suggérant que le dinosaure était un prototype du canard et celui-ci un prototype d'aigle.

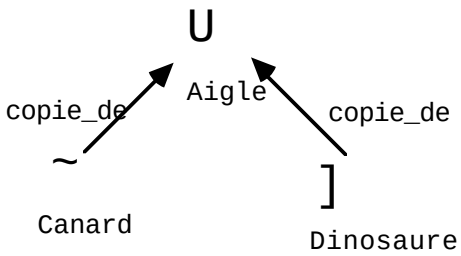


Fig. 2.21 : Représentation prototypique obtenue par l'acquisition des prototypes : aigle, puis dinosaure, puis canard.

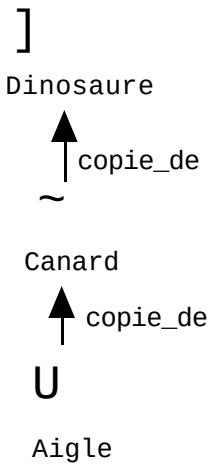


Fig. 2.22 : Représentation obtenue par l'insertion du dinosaure, puis du canard, et enfin de l'aigle.

Dans l'approche classe / instance, l'individu commence par créer une classe générique, *animal*, ayant les attributs communs à toutes les animaux, telles que est_vivant et peut_se_déplacer. L'*aigle* peut ensuite être créé comme instance de la classe *animal* ou bien d'une nouvelle classe, *oiseau*, sous-classe d'*animal*, pour laquelle il doit donner les attributs essentiels (Fig. 2.23).

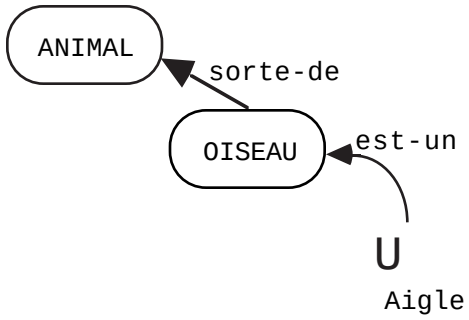


Fig. 2.23. Dans l'approche classe / instance, l'instance *aigle* est rattachée à sa classe d'appartenance, *oiseau*, sous-classe de la classe racine, *animal*.

Après, en regardant le *dinosaure*, il se rend compte des différences de base entre cet animal et les oiseaux (par exemple la température du sang) et il construit une nouvelle sous-classe d'animal, la classe *reptile*, dans laquelle il peut classer le *dinosaure*. Finalement, il classe le *canard* comme étant un membre de la classe *oiseau*. Le modèle produit (Fig. 2.24) ne dépend pas de l'ordre d'insertion des trois animaux. En effet, la place d'une instance de la base dépend de ses propriétés et des descriptions des classes de la base et non pas des instances déjà existantes dans la base.

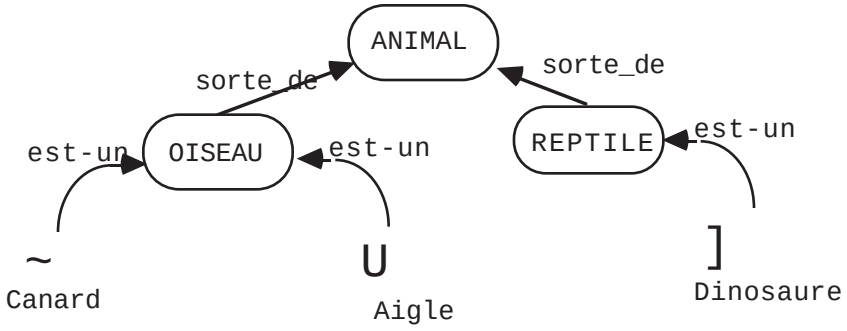


Fig. 2.24. Dans l'approche classe / instance toute instance est rattachée à une classe. La structure de la base de connaissances ne dépend pas de l'ordre d'insertion de ses instances.

D'autre part, l'interdiction de masquer les attributs d'une classe par une sous-classe garantit l'interprétation ensembliste des liens des classes et la cohérence de l'information de la base et permet de raisonner à partir de l'information stockée ; notamment ça permet de faire une classification correcte des individus de l'univers du discours. Cependant, l'interdiction du masquage réduit le pouvoir expressif du langage et s'avère une grande limitation dans des domaines ayant un grand nombre d'exceptions.

2.6.3. Modèle classe / instance - logique terminologique

Les logiques terminologiques et les représentations à objets (ayant l'approche classe / instance) ont plusieurs attributs en commun ; en particulier, le mécanisme principal de raisonnement dans les deux cas est la classification. Malgré leurs origines

différentes - les logiques terminologiques proviennent des réseaux sémantiques et les représentations à objets des schémas de Minsky - les deux types de systèmes représentent le monde en termes de ses éléments : des concepts dans le cas des logiques terminologiques, des classes et des instances dans les représentations à objets. De plus, ils organisent ces objets dans une structure induite par une relation d'ordre partiel. Les logiques terminologiques structurent les concepts à l'aide de la relation de subsomption qui détermine qu'un concept subsume un autre si l'ensemble d'objets satisfaisant la description du premier inclut l'ensemble d'objets satisfaisant celle du deuxième (§ 1.3.5.). Dans les représentations à objets on trouve deux types de liens : le lien de spécialisation entre classes qui correspond à la subsomption intensionnelle, et le lien d'appartenance qui lie un individu particulier à sa classe d'appartenance. Ce deuxième type de lien n'est pas explicite dans les logiques terminologiques (bien qu'il puisse être simulé par un concept décrivant un seul élément ou retrouvé dans les assertions). Dans les deux types de systèmes, les mécanismes de raisonnement de base : la classification, la vérification de cohérence, le filtrage, etc., tirent parti de cette structuration des objets.

Malgré ces similitudes, il y a certaines différences entre ces deux paradigmes. Les logiques terminologiques permettent la description de concepts définis comme des conjonctions ou des disjonctions de littéraux alors que les représentations par classes et instances ne permettent que les conjonctions de littéraux. Par contre, les logiques terminologiques ne manipulent pas des connaissances incomplètes[1] ou incertaines tandis que les représentations par classes et instances permettent la création et manipulation d'instances incomplètes et d'instances hypothétiques.

De plus, les logiques terminologiques offrent deux langages : la logique terminologique permettant de définir des concepts complexes à l'aide de concepts de la base et le langage assertionnel pour l'établissement d'assertions. Les représentations à objets offrent un seul langage de description de la base de connaissances, de ses concepts et de ses instances ; cette homogénéité d'expression rend plus facile la mise a jour de la connaissance ainsi que la compréhension du système et la génération d'explications (Swartout dans [PAT&90]).

Puis, les logiques terminologiques permettent de décrire un individu de deux façons : par une assertion décrivant l'individu ou par un concept individuel. Dans le premier cas, la validité de l'assertion est dépendante du contexte et n'est pas traité comme un définition d'un élément persistant de la base. Dans le deuxième cas, le concept individuel n'est pas distingué des concepts décrivant des classes d'individus. Tous les concepts de la base de connaissances sont au même niveau et les mécanismes d'inférences, comme la classification, ne peuvent pas tirer parti de la différence conceptuelle existant entre un individu et un ensemble d'individus. Dans les représentations par classes et instances, la nette différence entre une classe et une instance permet de faire des mécanismes spécialisés de raisonnement sur des instances, comme la classification d'instances et le filtrage et des mécanismes, moins fréquents de manipulation de classes comme l'insertion d'une nouvelle classe dans la base ou la conjonction de deux classes.

Enfin, les classes sont définies, en général, en fonction de leurs sur-classes alors que la définition d'un concept est donnée par l'ensemble exhaustive de ses attributs.

[1] Un certain degré d'incomplétude est pourtant toléré, dans le cas où plusieurs valeurs sont disponibles

Conclusion

Dans ce chapitre nous avons décrit les attributs et éléments des représentations à objets. Ces représentations modélisent le monde à l'aide de classes et d'instances. Une classe représente un ensemble d'individus et une instance un individu particulier de l'ensemble. Une classe est décrite par une structure schéma / attribut / facette. Les attributs représentent les propriétés, relations et composants présentes dans toutes les instances de la classe. Une instance est décrite par une structure schéma / attribut / valeur ayant, pour les attributs de sa classe, des valeurs qui valident leurs contraintes (facettes). La plupart des représentations par objets définissent des conditions nécessaires et suffisantes d'appartenance d'une instance à une classe.

Des systèmes ayant des attributs composants décrivent des objets composites ; ces systèmes doivent définir la sémantique de la composition et gérer la diffusion de valeurs et les relations de dépendance entre les composants et le tout.

Dans la base de connaissances les classes sont organisées en une structure induite par la relation de spécialisation ou inclusion ensembliste. Une classe récupère les attributs de ses sur-classes par le mécanisme d'héritage. Plusieurs algorithmes de gestion du partage des propriétés ont été développés pour résoudre le conflit engendré lorsqu'une classe spécialise plusieurs classes ayant des attributs du même nom. Dans le chapitre suivant nous allons montrer comment la notion de point de vue permet, d'une part de structurer la base de connaissances selon les perspectives des différents experts, et d'autre part, de résoudre d'une façon propre le problème de la spécialisation multiple.

Bibliographie

[BAR&91] BARTHES J.P., FERBER J., GLOESS P.Y., NICOLLE A., VOLLE P., *Objets et intelligence artificielle,* PRC-IA, pp.272-325, 1991.

[BOB&77] BOBROW D.G., WINOGRAD T., *An overview of KRL, a Knowledge Representation Language.* Cognitive Science, vol. 1, n°. 1, pp.3-45, 1977.

[BOB&80] BOBROW G., GOLDSTEIN P., *Descriptions for a Programming Environment*, AAAI'80, Standford University, CA, pp.187-189, 1980.

[BOB&83] BOBROW D.G., STEFIK M.S., *The Loops Manual. Programming in LOOPS*, Xerox Corporation, 1983.

[BOR81] BORNING, A. *The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory*, ACM Transactions on Programming Languages and Systems, vol. 3, n°. 4, pp. 353-387, 1981.

[BRA83] BRACHMAN R.J., *What IS_A is and isn't : An Analysis of Taxonomic Links in Semantic Networks*, IEEE Computer vol. 16, n°. 10, pp.32-36, 1983.

[BRA85] BRACHMAN R.J., "I Lied about the Trees" Or, Defaults and Definitions in Knowledge Representation. The A.I. Magazine, vol. 6, n°. 3, pp.80-93, 1985.

[BRA&85] BRACHMAN R.J., SCHMOLZE J.G.*An Overview of the KL-ONE Knowledge Representation System.* Cognitive Science, vol. 9, n°2, pp.171-216, 1985.

[CAR89] CARRE B., *Méthodologie orientée objet pour la représentation des connaissances*, Thèse Laboratoire d'Informatique Fondamentale de Lille, 1989.

[CHO&88] CHOURAQUI E., DUGERDIL Ph., *Conflict Solving in a Frame-like Multiple Inheritance System*, ECAI, Munich, pp.226-232, 1988.

[COI87] COINTE, P. *Meta-classes are First Classes : The ObjVlisp Model.* In Proceedings of the 2nd. OOPSLA, Orlando, Florida, pp.156-167, 1987.

[COR86] CORDIER, M.O., *Informations incomplètes et contraintes d'intégrité : le moteur d'inférences Sherlock.* Thèse d'Etat, Université de Paris-Sud, Centre d'Orsay, 1986.

[DAH&66] DAHL O.J., NYGAARD K., *SIMULA - An ALGOL-Based Simulation Language*. Communication of the ACM, vol. 9, nº. 9, pp.671-678, 1966.

[DAV87] DAVIS H.E., *VIEWS : Multiple Perspectives and Structured Objects in a Knowledge Representation Language*, Bachelor and Master of Science Thesis, MIT, 1987.

[DUC88] DUCOURNAU R. *YAFOOL*. Version 3.22. Manuel de référence, SEMA.METRA, Montrouge, 1988.

[DUC&89] DUCOURNAU R., HABIB M., *La Multiplicité de l'héritage dans Les Langages à Objets.* TSI, vol. 8, n°.1, janvier, pp. 41-62, 1989.

[DUC93] DUCOURNAU R., *Héritages et représentations*, Mémoire, Diplôme d'Habilitation à diriger des recherches, spécialité : Informatique, Université Montpellier II, 1993.

[DUG88] DUGERDIL P., *Contribution à l'étude de la représentation des connaissances fondée sur les objets. Le langage OBJLOG.* Thèse de l'Université d'Aix-Marseille II, 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*, M.Lenzerini, D.Nardi and M.Simi (éd.), John Wiley & Sons Ltd., pp. 245-256, 1991.

[ESC&90] ESCAMILLA J., JEAN P., *Relationships in an Object Knowledge Representation Model*, Proceedings IEEE. 2nd Conference on Tools for Artificial Intelligence, Washington D.C. USA, pp.632-638, 1990.

[FER84] FERBER J., *MERING : Un langage d'acteurs pour la représentation des connaissances et la compréhension du langage naturel*. Actes du 4ème Congrès AFCET / INRIA, Paris, pp. 179-189, 1984.

[FER88] FERBER G.J., *Coreferentiality : The Key to an Intentional Theory of Object Oriented Knowledge Representation.* in Artificial Intelligence and Cognitive Science, chapitre 6, J. Demongeot, T. Hervé, V. Rialle et C. Roche (éd.), Manchester University Press, 1988.

[FIK&85] FIKES R., KEHLER T. *The Role of Frame-Based Representation in Reasoning.* Communications of the ACM, vol. 28, n°. 9, pp.904-920, 1985.

[FOR&89] FORNARINO M., PINNA A-M., TROUSSE B., *Approche orientée objet pour la mise en œuvre des relations dans un langage de schémas*, 7-ième Congrès AFCET / INRIA, novembre, pp.886-895, 1989.

[FOX&86] FOX M.S., WRIGHT J.M., ADAM D., *Experiences with SRL : An analysis of a frame-based knowledge Representation*, Expert Database Systems, pp.161-172, 1986.

[GOL&83] GOLDBERG A., ROBSON D., *SMALLTALK-80, The Language and its Implementation.* Addison-Wesley, 1983.

[GRA88] GRANGER C., *CLASSIC - générateur de systèmes experts en classification et en diagnostic*, Manuel de l'Utilisateur Ver 2.2, ILOG, 1988.

[KAY88] KAYSER D., *What Kind of Thing is a Concept*, Computer Intelligence, vol. 4, pp.158-165, 1988.

[KIM&89] KIM W. , BANERJEE J., CHOU H.T., GARZA J.F., WOELK D.,*Composite Object Support in an Object-Oriented Database System.* In Proceedings of the 2nd. OOPSLA, Orlando, Florida, ACM SIGPLAN Notices vol. 22, n°.12, pp. 118-125, 1987.

[KIM&89] KIM W., BERTINO E., GARZA J.F., *Composite Objects Revisited*, ACM SIGMOD, Proceedings of the International Conference on the Management of Data, Portland, vol. 2, pp.337-347, 1989.

[LIE81] LIEBERMAN H., *A Preview of ACT 1*, AI Memo 625, AI Lab., MIT, Cambridge, MA, 1987.

[McCAR80] Mc CARTHY J. *Circumscription : A Form of Non-Monotonic Reasoning.* Artificial Intelligence Journal, vol. 13 , n°. 1-2, pp.27-39, 1980.

[McGRE&91] MAC GREGOR R.M., BURSTEIN M.H. *Using a Description Classifier to Enhance Knowledge Representation*, IEEE Expert Intelligent Systems and Applications, juin, 1991.

[MAR79] MARTIN W.A, *Descriptions and Specialization of Concepts* in Artificial Intelligence. An MIT Perspective vol. 1, pp.377-419, P.H.Winston, R.H. Brown. (éd.), The MIT Press, 1979.

[MAR&92] MARKOWITZ J.A., NUTTER J.T., EVENS M.W., *Beyond IS-A and Part-Whole : More Semantic Network Links*, Computers Math. Applic. vol. 23, nº6-9, pp.377-390, 1992.

[MAS&89] MASINI G., NAPOLI A. COLNET D. LEONARD D., TOMBRE K., *Les langages à objets*. Intereditions, Paris, 1989.

[MIS&88] MISSIKOFF M., SCHOLL M., *An Algorithm for Insertion into a Lattice : Application to Type Classification*, First Draft, août, 1988.

[MOO86] MOON D., *Object-Oriented Programming with Flavors*, in Proceedings of the 1st OOPSLA, pp.1-8, Portland, Oregon, 1986.

[NAP92] NAPOLI A., *Représentations à objets et raisonnement par classification en intelligence artificielle*, Thèse d'état, Université de Nancy 1, Nancy (FR), 1992.

[NAP&92] NAPOLI A., DUCOURNAU R., *Subsumption in Object-Based Representations*,Proceedings ERCIM Workshop on theoretical and practical aspects of knowledge representation, (rapport ERCIM 92-W001) pp1-9, Pisa (IT), 1992.

[PAT&90] PATEL-SCHNEIDER P.F., QWSNICKI-KLEWE B., KOBSA A., GUARINO N., MAC GREGOR R., MARK W. S., MC GUINNESS D. L., NEBEL B., SCHMIEDEL A., YEN J., *Term Subsumption Languages in Knowledge Representation*, Workshop of TSL'89, AI Magazine, vol. 11, n°. 2, 1990.

[PAT90] PATEL-SCHNEIDER P.F., *Practical, Object-based Knowledge Representation for Knowledge-based Systems*, Information Systems, vol. 15, n°. 1, pp.9-19, 1990.

[PET&88] PETERS S.L., SHAPIRO S.C., *A Representation for Natural Category Systems*, Object Oriented Computing Systems, B.Randel (Ed.), University of Newcastle Upon Tyne, 1988.

[REC88] RECHENMANN F., *SHIRKA : système de gestion de bases de connaissances centrées-objet*, Manuel d'utilisation 1988.

[REI80] REITER, R. *A Logic of Default Reasoning*, Artificial Intelligence, n°.13, pp.81-131, 1980.

[ROB&77] ROBERTS R.B., GOLDSTEIN I.P., *The FRL Manual*, AI Memo 409. AI Lab. MIT. Cambridge, MA, 1977.

[ROS78] ROSCH E., *Principles of Categorization.* In E.Rosch & B.B. Lloyd (Eds.), Cognition and Categorization, Hillsdale, NJ : Erlbaum, pp. 27-48, 1978.

[ROS90] ROSSAZZA J.P., *Utilisation des hiérarchies de classes floues pour le représentation de connaissances imprécises et sujettes à exceptions : le système "SORCIER"*, Thèse d'Informatique, Université Paul Sabatier de Toulouse, 1990.

[ROU88] ROUSSEAU B., *Vers un environnement de résolution de problèmes en biométrie : Apport des techniques de l'intelligence artificielle et de l'interaction graphique*, Thèse de doctorat, Université Claude Bernard Lyon 1, Lyon, 1988.

[SCH&83] SCHUBERT L., PAPALASKARIS M.A., THAUGHER J., *Determining Type, Part, Color, and Time Relationships*, IEEE Computer, vol. 16, n°. 10, pp.53-60, octobre, 1983.

[SCH86] SCHMIDT D., *Denotational Semantics*, Allyn and Bacon Inc., 1986.

[SMI&81] SMITH E., MEDIN D.L., *Categories and Concepts*, Harvard University Press, Cambridge, MA, USA, 1981.

[STE&85] STEFIK M., BOBROW D.G., *Object-Oriented Programming : Themes and Variations*, The A.I. Magazine, vol. 6, n°. 4, pp. 40-62, 1985.

[SUS&80] SUSSMAN G.J., STEELE G.S. Jr., *CONSTRAINTS— A Language for Expressing Almost-Hierarchical Descriptions*, Artificial Intelligence, vol. 14, pp.1 - 39, 1980.

[WEG87] WEGNER,P.,*The Object-Oriented Classification Paradigm*, dans Research Directions in Object-Oriented Programming, Bruce Shiver, Peter Wegner (éd.), The MIT Press, Cambridge, MA, 1987.

[WIN&87] WINSTON M.E., CHAFFIN R., HERRMANN D., *A Taxonomy of Part-Whole Relations*, Cognitive Science, vol. 11, pp. 417 - 444, 1987.

[WOO75] WOODS W.A., *What's in a Link : Foundations for Semantic Networks*, in Representation and Understanding, Bobrow D., Collins A. (éd.), Academic Press, New York, 1975.

Chapitre 3

Perspectives

Chapitre 3

Perspectives

"When there are many meanings in a network, you can turn things around in your mind and look at them from different perspectives ; when you get stuck, you can try another view. That's what we mean by thinking" [MIN83].

Fig. 3.1. L'abstraction se concentre sur les caractéristiques essentielles d'un objet, selon le point de vue de l'observateur [BOO90]

Introduction

Représenter consiste à faire une abstraction d'une réalité riche en information. Cette abstraction dépend de l'agent qui la réalise, de son domaine d'intérêt et de ses connaissances préalables. Ainsi, lorsque plusieurs observateurs (agents) travaillent sur un même univers de connaissance, ils observent des éléments et relations différents. La plupart de systèmes de représentation de connaissances négligent cette variété de perceptions et ils offrent des outils pour créer un modèle unique du monde, une représentation étroite où chaque utilisateur doit filtrer ce qui l'intéresse, un modèle mono-utilisateur, conçu pour un seul utilisateur du système, une seule vision du monde.

Par opposition à cette approche mono-point de vue mono-utilisateur (mono-disciplinaire) nous allons présenter dans ce chapitre l'approche multi-utilisateurs (multi-disciplinaire) qui permet de modéliser une même réalité selon des points de vue différents, un point de vue étant la perception qu'une personne a du monde observé.

La notion de point de vue ou perspective a été utilisée avec des sens divers dans différents domaines de l'informatique tels que le génie logiciel [BOB&80], les systèmes de raisonnement hypothétique [ATT&86] et les représentations de connaissances. Dans la première partie de ce chapitre nous discutons les différents sens donnés aux termes perspective et point de vue. Dans la deuxième partie nous montrons différentes façons de représenter les perspectives dans les représentations à objets. De même que des observateurs différents perçoivent des propriétés différentes des objets du monde, ils

peuvent identifier différentes parties dans la décomposition d'un objet complexe. La dernière partie de cette chapitre traite des décompositions multiples d'objets complexes.

3.1. Le concept de perspective

Le concept de **perspective**[1] a été utilisé avec des sens divers dans différents domaines de l'informatique. Cette différence reflète l'ambiguïté existante dans la langue commune quand on utilise les termes "perspective" ou "point de vue". En général on peut définir une perspective comme étant une position "conceptuelle" de laquelle un observateur regarde un objet. Ainsi la notion de perspective met en correspondance un agent et un monde. Comme nous l'avons présenté dans la section § 1.2.3 il existe quatre types de relations possibles entre un groupe d'agents et un monde observé :

1. Le monde est unique et les agents le perçoivent tous de la même façon : ce cas correspond à la représentation mono-disciplinaire. On a une seule perspective, une seule façon de voir le monde. Ce cas est le cas simple où le système ne manipule pas la notion de perspective.

2. Le monde est unique et les agents le perçoivent de façons complémentaires : dans ce cas, il y a plusieurs agents qui observent le même monde ; chaque agent en voit une partie. Une perspective est ici une représentation partielle mais correcte du monde ; l'union des différentes perspectives représentées est une représentation cohérente [ZEI84], [DAV87], [FER88], [SHA91]. Nous allons centrer notre discussion sur ce cas car il concerne la plupart des représentations par objets qui manipulent les perspectives. Par ailleurs, la notion de "vue" dans les systèmes de gestion de bases de données corresponde aussi à cette approche, car les vue sont des visions partielles d'une base unique [ULL88].

3. Le monde est unique et les agents en ont une perception incomplète et collectivement inconsistante : ce cas se base sur l'idée que le monde est trop complexe pour être connu complètement. Pour raisonner avec cette représentation incomplète du monde, chaque agent crée un contexte hypothétique, c'est-à-dire, un ensemble d'hypothèses, cohérentes entre elles et avec la connaissance certaine du monde. Une perspective est ici un contexte hypothétique, c'est-à-dire une représentation complète mais hypothétique du monde. Les perspectives de deux agents peuvent être contradictoires car elles peuvent inclure des hypothèses opposées. A ce groupe appartiennent des systèmes comme OMEGA[2] [ATT&86] où les perspectives (points de vues) sont des ensembles de règles individuellement monotones et collectivement non-monotones, permettant de suivre un raisonnement hypothétiques avec des ensembles différents d'hypothèses.

4. Le monde dépend de l'observateur : chaque agent a une réalité qui inclut son être et le monde qui l'entoure. Ainsi le monde est subjectif et le point de vue de chaque observateur est une vision complète mais subjective du monde, qui est indissociable de l'observateur [VAR89].

Par la suite nous allons centrer la discussion sur la deuxième interprétation de la relation agent-monde qui suppose un **monde unique**, figé et objectif ; **les point de vue sont des visions partielles mais complémentaires** de ce monde ; leurs unions produit une représentation cohérente du monde.

---

[1] Par la suite nous allons traiter les termes point de vue et perspective comme des synonymes.

[2] OMEGA organise la connaissance dans un graphe de descriptions formalisables en termes de la logique des prédicats. Son raisonnement est basé sur la structure taxinomique de la connaissance et sur des stratégies de déduction logique.

Une des premières références au terme perspective se trouve dans le travail de Minsky [MIN75]. Minsky introduit le concept de perspective avec une connotation **spatiale**. Pour lui, un objet peut être vu par des observateurs différents à partir de différents points de vue ; ces observateurs regardent tous les mêmes attributs mais chacun peut les voir avec des valeurs différentes selon son propre point de vue. Bien que différentes, ces valeurs sont unifiables par des opérations de transformation spatiale. Un objet est représenté sous la forme d'un tableau de schémas ; chaque schéma représente la perception qu'un observateur a de l'objet représenté, en fonction de la localisation spatiale de cet observateur. Un point de vue est lié aux autres points de vue par des opérations de transformation de l'objet. Ainsi, par exemple, les observateurs d'un cube regardent tous les faces du cube mais ils les voient de façons différentes selon leurs points de vue (Fig. 3.2). L'approche "spatiale" de cette vision vient du fait que l'intérêt principal de Minsky était le raisonnement dans un système de perception ou de vision par ordinateur.

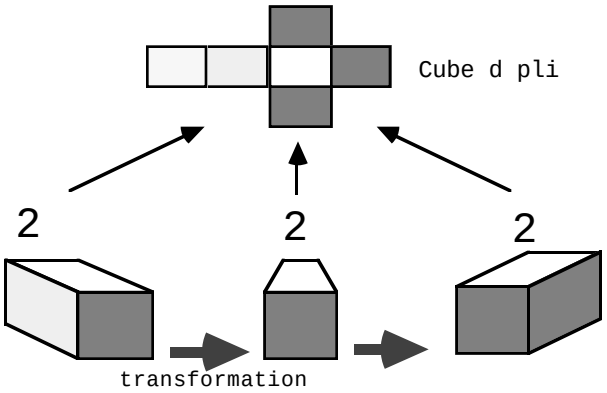


Fig. 3.2. Les perspectives de Minsky représentent les différentes perceptions qu'un observateur peut avoir de l'environnement, en fonction de sa position dans l'espace. Ces perspectives sont unifiables par des transformations spatiales.

Cette interprétation spatiale des perspectives est très utilisée dans les systèmes de perception [MIN75]. Dans d'autres domaine de l'I.A qui nécessitent de la représentation de grandes bases de connaissances, les perspectives servent a modéliser les divers domaines de connaissances présents dans la base. En effet, la construction d'une telle base doit prendre en compte le point de vue d'observateurs de domaines divers, sur les objets du monde : un objet de l'univers étant une entité très complexe, l'utilisateur doit faire un choix sur les attributs et relations qu'il va modéliser dans sa représentation. Les perspectives apparaissent ici comme les points de vue des observateurs des différentes disciplines regardant un même objet. Par opposition à l'approche précédente, les observateurs regardant un même objet peuvent voir des attributs différents, selon leurs intérêts ; pour un attribut commun ils vont tous voir la même valeur.

Ainsi, le cube de l'exemple peut être regardé par un mathématicien et par un peintre ; le premier regardera les propriétés géométriques tandis que le deuxième peut être plus intéressé par la combinaison des couleurs des faces (Fig. 3.3).

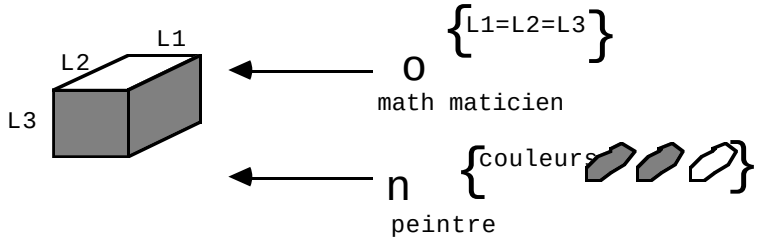


Fig. 3.3. Chaque utilisateur, en observant un objet, voit les attributs relevant de son domaine d'intérêt. Pour un même attribut, deux observateurs voient la même valeur.

L'approche "multi-disciplinaire" (plusieurs façons de voir un même monde) est utile pour modéliser des objets des domaines aussi divers que l'électronique et la biologie. En effet, les composants d'un circuit peuvent être vus du point de vue fonctionnel, technologique ou spatial ; les taxinomies d'animaux peuvent être construites en prenant en compte des propriétés morphologiques, évolutives ou géographiques des espèces, les attributs visibles dans chacune des taxinomies étant différents.

La plupart des systèmes de représentation par objets qui utilisent la notion de perspective, le font pour modéliser des systèmes "multi-disciplinaire" ; ils sont exposés par la suite.

3.2. Les perspectives dans les R.C.O.

Bien que les applications en I.A. soient en souvent multi-disciplinaires, il existe très peu de systèmes traitant explicitement le concept de perspectives dans la représentation. Par la suite, nous allons analyser les différentes façons de représenter les perspectives dans les modèles par objets à l'aide d'un exemple simple, une base de connaissances des employés enseignants d'une université que l'on veut regarder à partir de deux point de vue : domaine et emploi. Le point de vue domaine établit une classification des employés selon leurs unités de travail : faculté et département de l'université (Fig. 3.4).

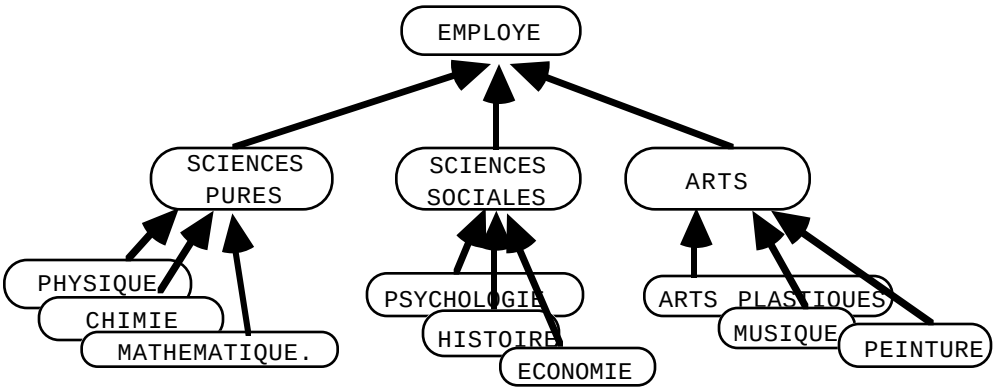


Fig. 3.4. Un employé de l'université dépend d'un département d'une faculté. Par exemple, l'employé Paul Ricaud donne des cours dans la division peinture de la faculté d'arts.

Du point de vue emploi, un enseignant peut être : maître de conférences, professeur ou assistant (Fig. 3.5). Un employé, par exemple "Paul Ricaud, assistant d'Arts", est décrit, d'une part, par ses attributs en tant que assistant : "assistant de classe A" et d'autre part, par ses caractéristiques en tant que membre de la faculté d'Arts : "donne des cours de peinture aux élèves de première année". Ces deux groupes d'attributs doivent pouvoir être manipulés indépendamment.

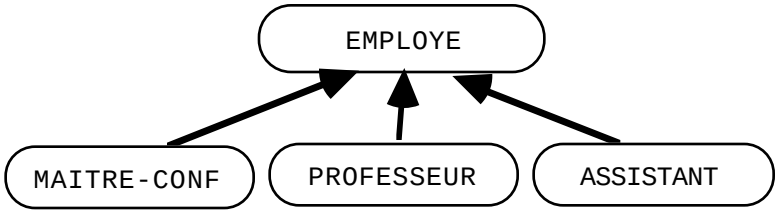


Fig. 3.5. Un enseignant de l'université peut être un assistant, un professeur ou bien un maître de conférence. Paul Ricaud, par exemple, est un assistant.

3.2.1. Représentation des perspectives par héritage multiple

Les systèmes ne traitant pas les perspectives explicitement modélisent ce concept par l'héritage multiple. Ainsi, si toutes les facultés de l'université ont les trois types d'enseignants, le modèle aura une sous-classe pour chaque faculté et chaque type de travail (Fig. 3.6). Si, en plus, les facultés sont organisées en département (Fig. 3.4), le graphe de classe s'agrandit énormément.

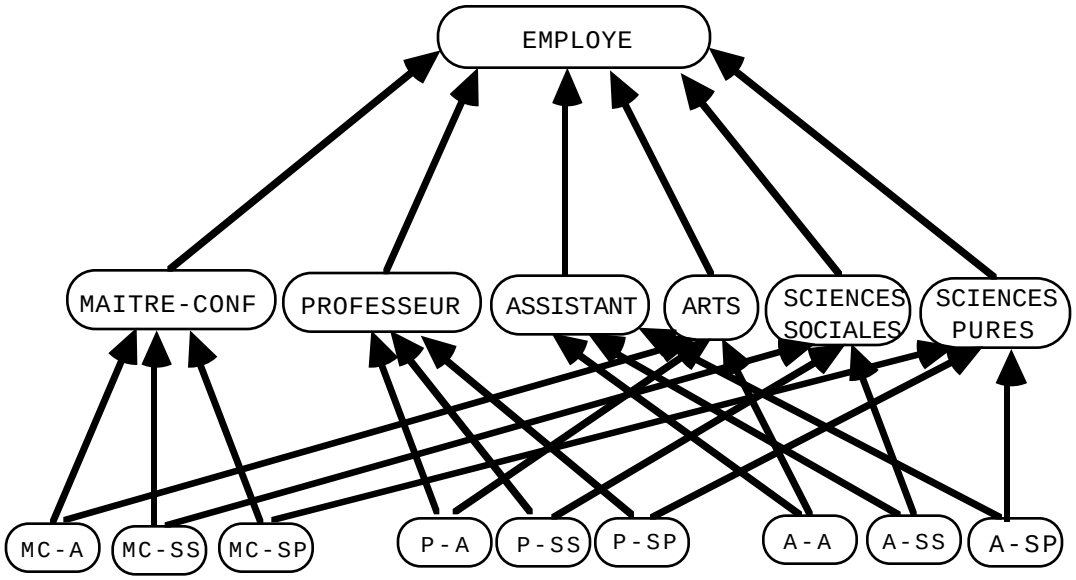


Fig. 3.6. Modéliser les perspectives par l'héritage multiple revient à créer un graphe où les deux classifications de départ sont confondues (toutes les classes sont au même niveau).

Une telle représentation peut présenter les problèmes liés à l'héritage multiple, tels que le conflit d'attributs ayant le même nom (§ 2.4.2). D'autre part, dans la structure obtenue, les points de vue de départ sont mélangés ; les mécanisme d'inférence de la base ne peuvent pas faire la distinction entre les classes appartenant à des perspectives différentes. De plus, rien n'empêche la création d'une classe "Assistant-Professeur", classe qui n'aura jamais d'éléments car aucune personne ne peut être assistant et professeur en même temps. Enfin, dans une instance de la base, les attributs correspondants aux différents points de vue sont tous mélangés dans un même schéma. Pour raisonner avec une telle instance, une personne est obligé de manipuler tous ses attributs, et non pas seulement ceux de son domaine d'intérêt. Par exemple, dans l'instance "Paul Ricaud" les attributs concernant des aspects administratifs (en tant qu'assistant), tels que le salaire, l'horaire de travail, etc. sont mélangés avec des attributs qui concernent son profil professionnel : domaine de recherche, cours donnés, etc[1].

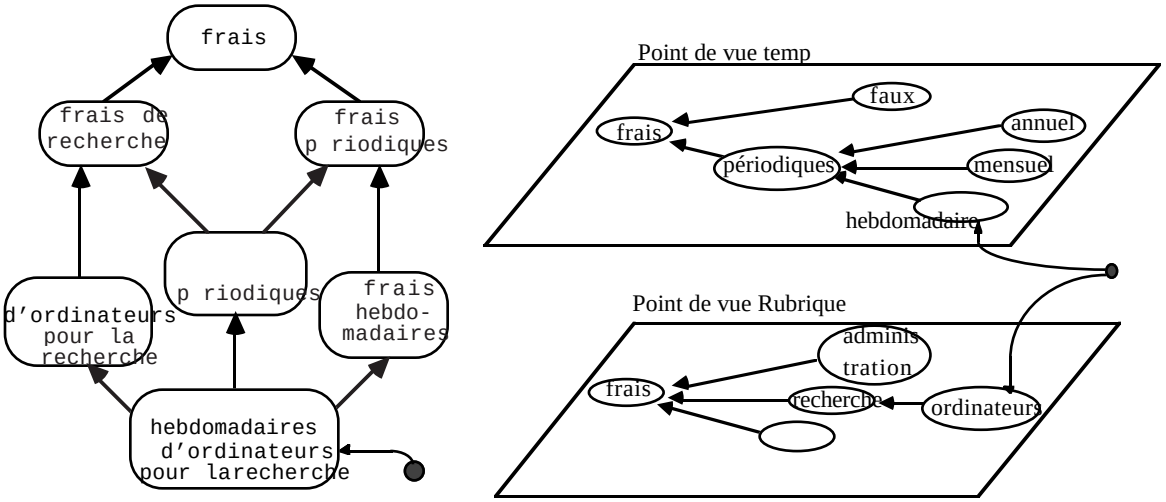
L'exemple précédent indique que lorsque l'on représente plusieurs points de vue dans une même hiérarchie de classes, le graphe résultant a plusieurs classes sœurs non disjointes ; une telle structure donne lieu aux spécialisations multiples et aux problèmes d'héritage multiple d'attributs soulignés auparavant (§ 2.4.2.).

L'analyse inverse montre que dans la plupart de cas, le problème de l'héritage multiple se présente lorsque l'on mélange dans une même hiérarchie des points de vue différents. En effet, dans les exemples des graphes de classes ayant des spécialisations multiples [FIK&85], [STE&85], etc., deux classes sœurs ayant des sous-classes communes correspondent à deux points de vues différents.

---

[1] [COR86] présente une solution alternative au multi-héritage qui conserve la structure arborescente de la hiérarchie mais qui crée plusieurs occurrences d'une même classe. Ainsi par exemple en dessous de CHAQUE feuille du graphe du domaine (Fig.3.4) il y aurait les trois classes "maître-conf", "assistant" et "professeur".

Ainsi par exemple, la hiérarchie des frais d'entreprises (Fig. 3.7.a) présentée dans [GOO79] comme un exemple d'héritage multiple du système KLONE, décrit deux points de vues de ces frais, le point de vue **temps** et le point de vue **rubrique**. Si l'on sépare ces points de vue, les graphes d'héritage deviennent des arbres (Fig. 3.7.b) ; une instance de la classe *frais hebdomadaires d'ordinateurs pour la recherche* du graphe original correspond, dans la représentation multi-points de vue, du point de vue rubrique à un frais d'ordinateur et du point de vue temporel, à un frais hebdomadaire.



a) b)

Fig. 3.7. a) représentation sans point de vue des frais d'une entreprise (pris de [GOO79]) ; plusieurs classes sont des spécialisations multiples. b) représentation avec deux points de vue : temps et rubrique.

Dans les sections suivantes nous discutons quatre systèmes qui exemplifient assez bien les différentes façons de représenter explicitement les perspectives dans les représentations par objets : KRL, LOOPS, ROME et VIEWS.

3.2.2. KRL

Développé comme un outil de construction de systèmes pour la compréhension du langage naturel, KRL : Knowledge Representation Language [BOB&77] est un modèle mixte qui inclut de la connaissance procédurale et une base déclarative structurée. Dans la description des unités déclaratives, KRL est le premier système à reconnaître qu'un objet peut être vu de plusieurs façons, selon le point de vue de l'observateur.

Une unité de KRL est un ensemble de descriptions qui sert comme référence mentale unique des entités et catégories du monde. Il y a des modes différents pour décrire un objet : appartenance à une classe, relation dont l'objet est un membre, identification unique, rôle dans un événement, etc. Chaque description est une liste de descripteurs ayant des facettes associées ; chaque descripteur est une caractérisation indépendante de l'objet associé. Ainsi, par exemple pour parler de Paul Ricaud on peut dire : *"Un employé de l'université", "Un enseignant de peinture" ,"Un assistant"*, etc.

Dans KRL il y a sept types différents d'unités : *Basic, Specialization, Individuals, Abstract, Manifestation, Relation* et *Proposition*. Les trois premières permettent le traitement des perspectives. Les classes *Basic* sont des racines des graphes des différents familles d'objets (tels que *Personne* et *planète*) qui établissent une première partition de l'univers du discours; les classes *Specialization* sont des sous-classes d'une classe *Basic* ou d'une autre classe *Specialization* et les classes *Individuals* décrivent des entités uniques du monde : des individus.

Pour traiter les perspectives, KRL utilise le descripteur *perspective*. Un individu a une première perspective qui est la classe la plus générale à laquelle il appartient, une unité de type *Basic* et il peut avoir d'autres perspectives parmi les unités de spécialisation de sa classe de base. Ainsi, dans l'exemple, la classe de base est la classe *Employé* qui peut être spécialisée en divers unités, telles que *Maître-conférences, Professeur, Assistant, Art, Science Pures, etc.* L'unité individuelle, Paul, est un employé qui, selon la perspective *Assistant* est un assistant de classe A et, selon la perspective *Art* est un enseignant de Peinture (Fig. 3.8) :

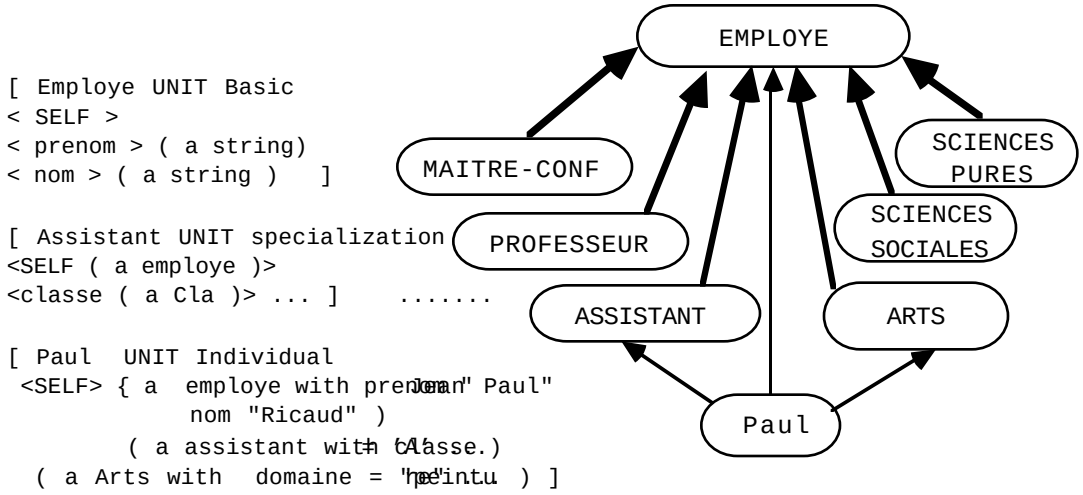


Fig. 3.8. Les perspectives dans KRL sont représentées au niveau des instances. Une instance est liée à son unité de base et à ses perspectives. Les attributs hérités de chacune de ces unités sont organisés dans des groupes d'attributs.

Dans l'unité individuelle qui représente Paul, les attributs sont groupés selon la perspective à partir de laquelle ils décrivent l'objet. Cette idée de grouper les attributs d'une instance a été étendue à des domaines bien différents tels que le développement de programmes dans PIE [BOB&80] et la linguistique dans le système OWL II [MAR79]. OWL ajoute au modèle de KRL la possibilité de changer la perspective principale à partir de laquelle on voit un objet ; ainsi, dans l'exemple précédent, *Assistant* peut être décrit comme un "exemple" de *Employé* ou bien on peut voir *Employé* comme étant une "caractérisation" d'*Assistant*.

3.2.3. LOOPS

Contrairement à KRL, OWL II et PIE, où tous les attributs sont stockés dans le même objet décrivant l'individu, LOOPS [STE&85] divise l'instance même dans des objets différents, ses différentes perspectives.

Dans LOOPS, un objet et ses perspectives sont une sorte d'objet composite, les composants étant les différentes perspectives. Chaque perspective est un objet indépendant auquel on peut s'adresser directement pour lui envoyer des messages de modification, d'élimination, etc. Cette indépendance permet de définir des attributs de même nom ayant des sens différents dans plusieurs perspectives.

Les perspectives sont toutes liées à l'objet ; le lien entre perspectives est un lien conceptuel, dû au fait qu'elles nomment toutes le même objet. En reprenant l'exemple précédent, l'individu Paul est lié à ses deux perspectives : *Paul comme un assistant de classe A* et *Paul comme un enseignant de Peinture de la faculté d'Arts.* A la différence des objets composites, les perspectives sont créées dynamiquement, à la demande. Ainsi, on pourrait ajouter par la suite une perspective *Sexe*, divisant les employés en deux classes et voir *Paul comme homme*.

A part le lien existant entre une instance et ses perspectives, chaque perspective, en tant qu'instance indépendante, appartient à une classe à laquelle elle est liée par le lien *est-un*. *Paul* est-un *Employé* tandis que sa perspective *Assistant de classe A* est-un *Assistant* et sa perspective *Peinture* est une instance de la classe *Art*. Pour traiter les perspectives, LOOPS utilise deux classes abstraites (appelées mixins) : *Node* et *Perspective*. Une classe décrivant des objets qui sont des perspectives d'autres objets doit être définie comme sous-classe de la classe *Perspective* et une classe qui décrit des objets ayant des perspectives est sous-classe du mixin *Node* .

Dans l'exemple précédent les classes *Assistant* et *Arts* sont sous-classes du mixin *Perspective* et la classe *Employé* est sous-classe du mixin *Node* (Fig. 3.9).

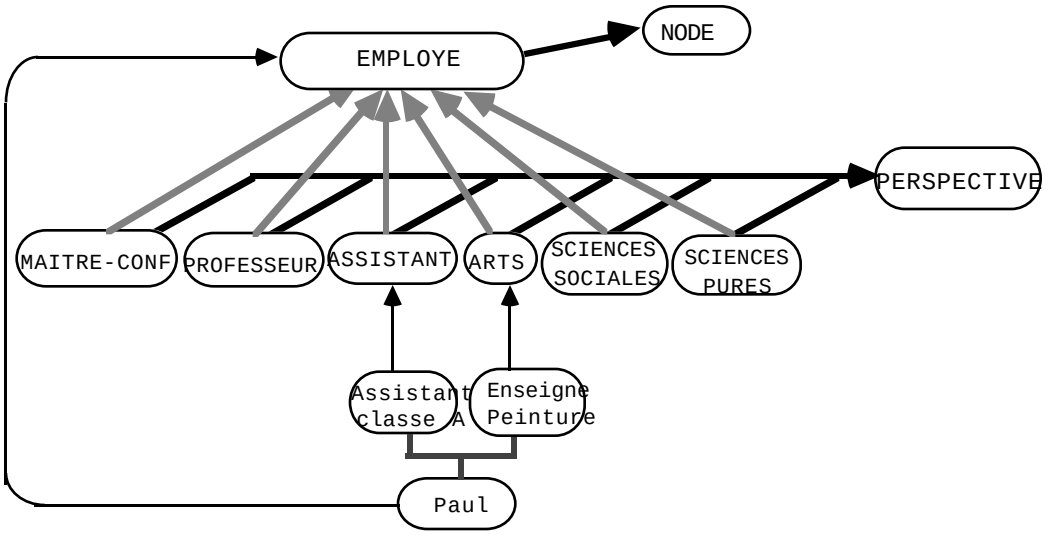


Fig. 3.9. Les perspectives d'un objet dans LOOPS sont des objets indépendants, instances des sous-classes du mixin "Perspective". L'objet même est membre d'une sous-classe du mixin "Node".

La modélisation des "perspectives" dans LOOPS apporte de nouvelles idées sur le sujet. D'une part, le traitement indépendant de perspectives permet de regarder un objet selon un point de vue sans être saturé par l'information des autres points de vue, fait qui reflète la façon de travailler d'un groupe interdisciplinaire : il regarde le modèle d'un point de vue et il en tire des conclusions, puis il change de perspective pour compléter ou vérifier l'information acquise. D'autre part, la dynamique permise par cette indépendance au niveau de la création et de la modification des perspectives permet de stocker une instance dans la base et de la manipuler sans avoir l'information complète de toutes les perspectives.

Malgré ses apports, la représentation de perspectives dans LOOPS exige l'ajout d'éléments artificiels, comme les liens spéciaux et les classes abstraites. Notamment, le fait de mélanger les mixins *Perspectives* et *Node* avec le graphe de classes de la base obscurcit le modèle[1].

3.2.4. ROME

Le système ROME [CAR89] est un langage hybride qui combine les principes des représentations par objets et des langages à objets. Des langages à objets il prend la notion d'instanciation : une classe a des méthodes pour créer des instances, ces instances étant définitivement attachées à cette classe par le lien d'instanciation. Des classes ROME ayant cette fonctionnalité d'instanciation sont appelées **classes d'instanciation**.

[1]L'option de perspectives de LOOPS n'est pas encore implémentée dans le langage de règles de LOOPS version 2.2.

Vu au niveau représentation, un objet (instance) peut être incomplet, il peut être vu selon différents aspects et il peut évoluer pendant sa vie ; ainsi, pour traiter le côté représentation des instances, ROME introduit la notion de **classe de représentation**. Une classe de représentation, par opposition à une classe d'instanciation, n'a pas la fonctionnalité d'instanciation. Les classes de représentation sont des spécialisations d'une classe d'instanciation ; elles décrivent des sous-ensembles d'individus de la classe d'instanciation, ayant certaines propriétés spécifiques. Une instance appartient à une seule classe d'instanciation (mono-instanciation) mais elle peut avoir plusieurs classes de représentation. De plus, pendant sa vie, une instance qui évolue peut changer de classes de représentation, mais elle reste toujours rattachée à sa classe d'instanciation.

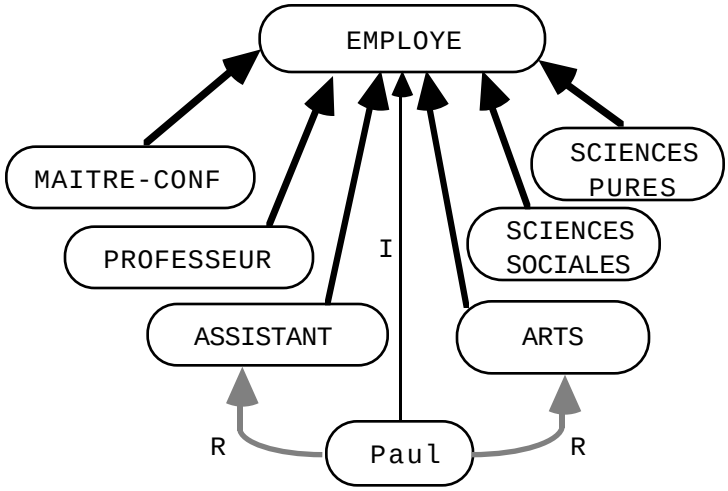


Fig. 310. Une instance de ROME a une classe fixe d'instanciation, mais elle peut avoir plusieurs classes de représentation, sous-classe de sa classe d'instanciation. Les liens de représentation peuvent changer lors de l'évolution de l'instance.

Dans l'exemple de l'université, "Paul Ricaud" est, avant tout, une **instance** de "Employé" ; il est aussi **représentant** des classes "Assistant" et "Arts" (Fig. 3.10). Cette idée d'avoir une classe d'instanciation pour l'information de base de l'objet et plusieurs classes de représentation pour ses perspectives est proposée aussi par le système PINOL [NGU&91] qui fait une distinction entre le type d'une instance contentant son information structurelle de base et ses classes qui représentent les différentes perspectives.

Cette distinction entre classe et lien d'instanciation, d'un côté et classe et lien de représentation de l'autre côté, permet, au niveau langage, de distinguer les différents liens intervenant dans le graphe, et au niveau représentation, de représenter des connaissances variées et évolutives.

A part la possibilité de lier une instance à plusieurs classes de représentation, ROME offre une autre facilité pour la manipulation des **points de vue** : la possibilité de parcourir un sous-graphe du graphe incluant certaines classes spécifiques. Supposons dans l'exemple précédent que l'on veut créer la classe "Assistant-Art" pour décrire certaines propriétés propres à tous les assistants de cette faculté. "Paul Ricaud" est maintenant une instance de cette nouvelle classe. Pour pouvoir regarder Paul en tant que membre de la faculté d'Art, sans avoir à manipuler son information en tant que assistant, ROME définit le concept de *point de vue d'une classe sur un objet.*

Le point de vue d'une classe C sur un objet O est l'ensemble de sur-classes et de sous-classes de C qui sont sur-classes de la classe d'instanciation de O. Ainsi, le point de vue de la classe "Art" sur l'objet "Paul Ricaud" inclut les classes "Employé", "Arts" et

"Assistant-Arts" (de ce dernier on ne regarde que les attributs hérités d "Arts" et d'"Employé") (Fig. 3.11).

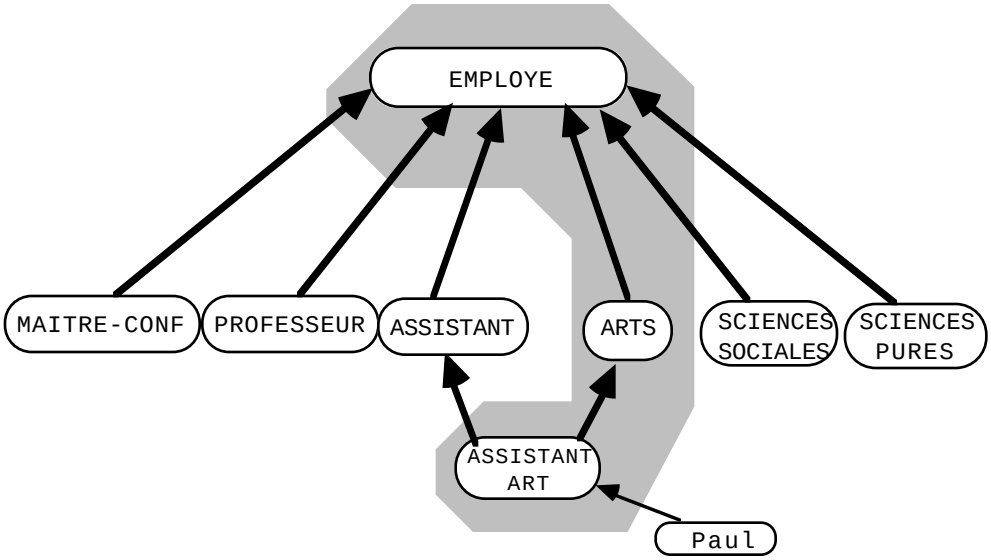


Fig. 3.11. Un point de vue dans ROME est définit en termes d'une classe et d'une instance et il détermine un sous-graphe de la base. Le point de vue de la classe "Art" sur l'instance "Paul" inclut les classes "Employé","Art" et "Assistant-Art".

Le modèle ROME offre plusieurs avantages. D'une part, il établit la différence conceptuelle existant entre la classe qui définit un concept et les classes qui ajoutent des contraintes pour décrire des sous-catégories de ce concept[1]. Un individu ne peut faire partie que d'un seul concept, fait qui justifie la mono-instanciation. D'autre part, par la représentation multiple, il permet la liaison d'une instance à plusieurs classes. Finalement, dans un graphe ayant des classes sous-classes de plusieurs classes, on peut faire le parcours du graphe de façon à ne regarder que des classes liées à une certaine classe.

Il est important d'établir ici la différence conceptuelle entre la notion de point de vue introduit par ROME et celle définie dans l'introduction de ce chapitre. Dans le cas de ROME le point de vue est déterminé par une classe et une instance ; ainsi on parle du point de vue "Assistant" sur "Paul Ricaud" ou du point de vue "Art" sur "Paul Ricaud". Cette définition sert au parcours du graphe mais cache, comme dans les modèles précédents, le fait que "Assistant" et "Maître-conférences" décrivent l'employé du côté "emploi" tandis que'"Art" et "Sciences sociales" traitent le côté "domaine". Notamment, le fait qu'une classe ne peut pas avoir comme sur_classes une classe du côté "emploi" et une autre du côté "domaine" doit être décrit explicitement à l'aide des relations spéciales comme la disjonction de classes.

3.2.5. VIEWS

Le système Views [DAV87] est un langage de représentation de connaissances qui combine des idées des schémas et des réseaux sémantiques. L'unité de description est la vue (view) qui comporte trois types d'objets dans une organisation de réseau : les parties, les relations et les contraintes. Une partie peut être vue comme un nœud d'un réseau qui peut être une valeur simple ou bien une autre vue. La partie n'a pas a priori une sémantique, celle-ci est donnée par les relations entre les parties. Les relations

[1] Les notions de classe d'instanciation et classe de représentation de ROME ressemblent respectivement aux unités Basic et Specialization de KRL.

fournissent les éléments structuraux de base d'une vue : elles changent une collection de parties en une structure significative. Les relations sont comme les attributs des schémas; elles peuvent avoir différentes sémantiques (par exemple : PartOf, Subclass, Subtask, etc). Enfin, les contraintes limitent la portée de la structure d'une vue (les valeurs possibles d'une partie et des relations et même d'autres contraintes).

Dans Views une perspective est représentée par une vue, c'est-à-dire un ensemble d'éléments liés entre eux. Cette interprétation structurelle des perspectives rejoint la vision spatiale de Minsky : deux observateurs qui regardent une chambre de deux points de vue différents, voient ses meubles (éléments) avec des relations spatiales (structures) différentes.

Les perspectives dans le sens multi-disciplinaire qui permettent plusieurs organisations des classes des objets du monde et différents groupements de ses attributs, peuvent être décrites dans Views par des vues ayant des graphes de classes et d'instances. L'exemple de l'université peut être décrit pas trois vues : une vue pour chacune des perspectives (emploi et domaine) et une vue globale qui groupe ces perspectives (Fig. 3.12).

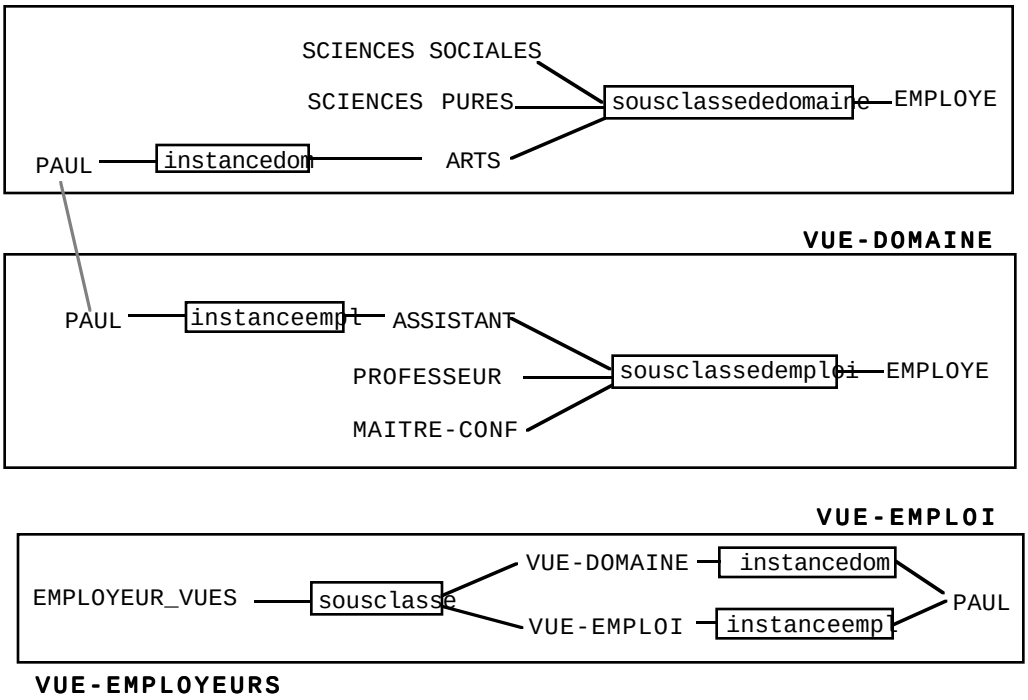


Fig. 3.12. Dans Views chaque perspective est décrite par une vue. Les perspectives sont liées dans une troisième vue globale. Les relations de spécialisation des différentes vues doivent être liées entre elles par une quatrième vue "Relations", créée par l'utilisateur.

A la différence des systèmes précédents, Views permet de faire un graphe de classes pour chaque perspective et de le stocker dans une unité de représentation. Ainsi, chaque agent voit la structuration du monde qui lui convient et il peut ignorer les autres vues. Deux vues, a priori indépendantes, peuvent être liées par des contraintes ou par des éléments communs (par exemple lorsqu'une classe est présente dans les deux taxinomies). De plus, dans tous les systèmes présentés auparavant, l'instance est attachée à une classe principale (Basic dans KRL, classe d'instanciation dans ROME, une sous-classe de la classe Node dans LOOPS) et à plusieurs classes secondaires représentant les points de vue. Dans Views il n'y a pas de classe ou vue principale servant de référent aux perspectives ; toutes les vues ont la même importance, ce qui permet à chaque agent de voir sa vue comme étant la vue principale de la base.

En plus la représentation de taxinomies multiples, la vue dans Views sert à représenter des décompositions multiples d'objets complexes (§ 3.3.) et des modélisation de contextes hypothétiques à la OMEGA, car la sémantique des relations et contraintes est complètement à la charge du concepteur de la base. Cette flexibilité s'avère un des principaux inconvénients du système, car celui-ci ne possède pas de mécanismes d'inférence spécialisés pour la manipulation multi-disciplinaire de la base, tels que l'interaction entre les agents pour résoudre un problème, la classification d'instances dans les taxinomies multiples, etc.

Les systèmes présentés auparavant montrent l'évolution de la représentation explicite des perspectives dans les RCO. KRL (§ 3.3.2) est le premier système à faire la distinction entre les classes d'instanciation et les classes de spécialisation d'une instance, ces dernières correspondent aux perspectives. Toute l'information d'une instance est stockée dans une même structure mais elle commence à être organisée dans des groupes d'attributs selon les diverses perspectives. Dans LOOPS (§ 3.2.3) la distinction entre les classes d'instanciation et les classes de perspectives est prolongée des liens de l'instance aux liens du graphe de classes. Le système ROME (§ 3.3.4), (de même que KRL) a deux types de liens pour l'instance : le lien d'instanciation, fixe et unique et les liens de représentation qui peuvent évoluer. La notion de point de vue dans ROME sert a délimiter la partie active du graphe de classes, lors des inférences et des recherches d'attributs hérités par une instance. Enfin, VIEWS (§ 3.3.5) propose la représentation de perspectives à l'aide de vues, "views", unités structurées de représentation de connaissances composées de parties, relations et contraintes ; ainsi, l'information d'une perspective peut être représentée par une vue dont les seules relations sont la spécialisation et l'instanciation.

3.3. Les perspectives et les objets composites

Dans la section § 2.5 nous avons présenté les différentes relations de compositions existant entre un objet composite et ses parties et nous avons discuté différents types de systèmes qui manipulent les objets composites. Dans cette partie nous allons voir comment l'ajout de la notion de points de vues dans une représentation entraîne des conséquences sur la décomposition d'un objet composite. En effet, la décomposition en parties d'un objet n'est pas unique ; elle dépend des objectifs du concepteur de la base ; "things are often described in terms of parts and wholes ; the way the division into parts is made depends on the purpose of the analysis"[1]. Ainsi, deux observateurs regardant le même objet vont le décomposer différemment selon leurs points de vue. Comme nous avons dit auparavant (§ 3.1 par.2) nous travaillons sur la base qu'un objet a une réalité unique ; ainsi deux points de vue sur le même objet doivent être cohérents tant au niveau des propriétés qu'au niveau des composants.

Très peu de systèmes permettent la représentation de décompositions multiples [DAV87], [BLA&87], [SUS&80], [WOL&91]. Dans CONSTRAINTS [SUS&80], un langage de réseaux hiérarchiques de contraintes pour la modélisation de circuits électriques, un circuit est décomposé en différents circuits équivalents ; chaque décomposition permet de déduire des paramètres de certaines équations ; la mise en correspondance de ces paramètres permet de résoudre le système algébrique associé au

[1]Dans [BLA&87] : " Les objets sont souvent décrits en terme de parties et du tout ; la façon dans laquelle cette division en parties est faite dépend du but de l'analyse".

circuit. VIEWS [DAV87] offre la notion de vues : une vue représente la structure d'une unité complexe d'information ; des vues décrivant un même objet avec des structures différentes représentent des décompositions multiples.

Les composants d'un objet composite peuvent, eux-aussi, être des objets composites ; on peut avoir donc un graphe de décomposition décrivant une décomposition imbriquée d'un objet. Il faut noter que pour garantir la cohérence des inférences, la sémantique donné à la relation "partie de" doit être la même dans tous les niveaux de la décomposition (§ 2.5). Des décompositions multiples imbriquées établissent un ensemble de graphes de composition. Lorsque la sémantique de la relation de composition se base sur la métaphore de l'objet physique (§ 2.5.1), les graphes de décomposition sont des arbres et la décomposition multiple correspond à une forêt dont tous les arbres ont la même racine, l'objet complexe tout entier. Ainsi, par exemple une voiture, vue du point de vue mécanique est composée de moteur, transmission, suspension, direction et freinage, tandis que d'un point de vue physique elle a une carrosserie et 4 roues, la carrosserie étant composée des sièges, portes, instruments de bord et essuie-glaces (Fig. 3.13).

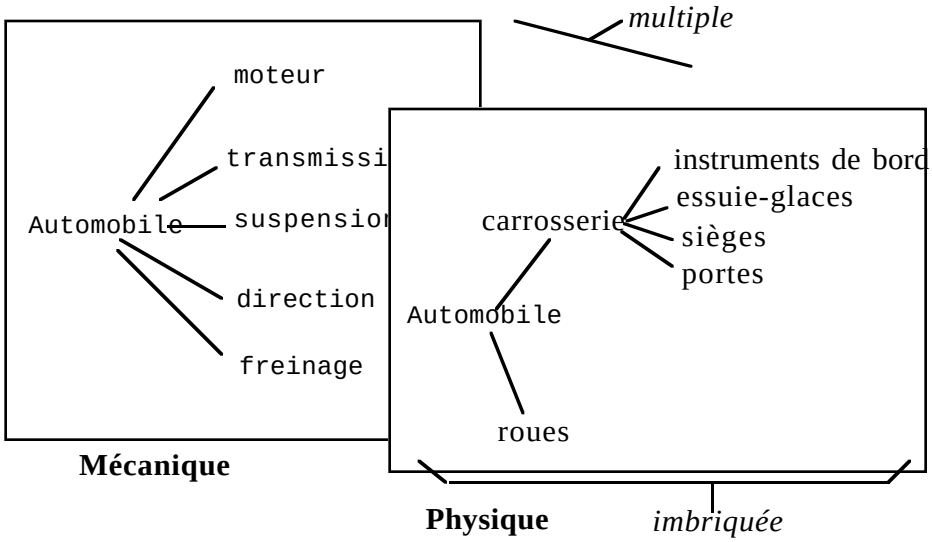


Fig. 3.13. Décomposition multiple imbriquée d'une voiture

Dans la plupart de systèmes physiques on peut identifier au moins deux points de vue pour la décomposition des objets : le point de vue structurel et le point de vue fonctionnel[1] (Fig. 3.13).

Les diverses décompositions d'un objet dans les différents points de vue peuvent être plus ou moins indépendantes, selon la relation existant entre leurs composants. On peut identifier quatre cas :

- Chaque composant n'est présent que dans une seule des décompositions (dans un seul des points de vue) (Fig. 3.13).
- Un même composant est présent dans deux décompositions ; c'est le cas du cœur dans l'exemple du corps humain qui est un composant structurel et fonctionnel du corps (Fig. 3.14.). Il est intéressant de noter que pour ce même composant, le cœur, on peut regarder des propriétés et composants différents dans chaque point de vue.

[1]Ces deux points de vue pourraient correspondre, dans l'analyse de [WIN&87] aux relations de composition séparables et fonctionnelles respectivement. Dans l'exemple de la voiture (Fig.3.12), le point de vue physique correspond au point de vue structurel et le point de vue mécanique correspond au point de vue fonctionnel (bien qu'on puisse aussi bien penser à un deuxième point de vue fonctionnel : le point de vue électrique).

- Un composant d'une décomposition groupe plusieurs composants de l'autre. Par exemple le composant "intestins" du point de vue structurel correspond aux composants "gros intestin" et "intestin grêle" du point de vue fonctionnel (Fig. 3.14).
- Un composant d'un point de vue n'est pas forcement disjoint avec un composant d'un autre, ce qui veut dire que ces deux composants peuvent encore se décomposer dans des éléments plus simples et qu'un groupe de ces éléments est commun aux deux composant présentés.

Dans le premier cas, les décompositions sont indépendantes et chacune peut être traitée comme une décomposition d'une représentation mono-perspective. Les autres trois cas peuvent poser des problèmes de maintien de la cohérence entre perspectives et ils rendent difficile l'identification des relations de compositions entre des objets.

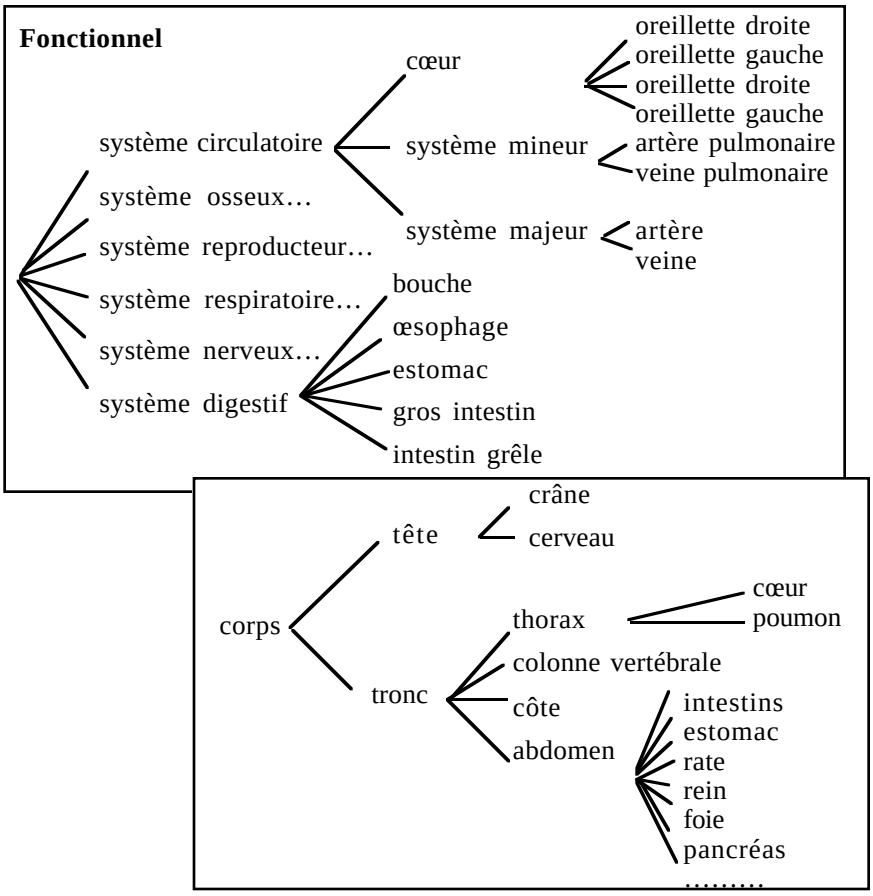


Fig. 3.14. En anatomie, le corps humain est décomposé du point de vue fonctionnel selon les systèmes qui font les différentes fonctions vitales et du point de vue structurel selon la localisation des différentes organes et parties. Un même composant peut être présent dans les deux décompositions, comme c'est le cas du cœur.

Schubert et al. [SCH79], [PAP&81], [SCH&83] montrent que, lorsqu'on complète les graphes de décompositions pour avoir, au niveau des feuilles des différentes perspectives, un ensemble de composants disjoints deux à deux, on peut établir des méthodes simples et complètes d'inférence pour résoudre le problème de savoir si un objet a est partie d'un objet b, c'est-à-dire, étant donnés deux objets du monde, sire si l'un fait partie du graphe de décomposition de l'autre. Ce problème est particulièrement important pour la propagation des valeurs dans les systèmes qui parcourent le graphe de composition de bas vers le haut, par exemple pour l'assemblage de pièces.

Les graphes de décomposition fermés et demi-fermés proposés par Schubert et al. concernent des décompositions inter-dépendantes d'un objet particulier dont les composants sont des unités (mono-valués). Les décompositions alternatives

(complètement indépendantes) décrites au niveau des classes d'objets (Fig. 3.14) et permettant des composants multiples (comme les "portes" de la voiture Fig. 3.13) ne peuvent pas être traités par ces méthodes.

Conclusion

Les perspectives dans les systèmes de représentation de connaissances peuvent représenter des notions diverses telles que des mondes hypothétiques différents, des positions spatiales complémentaires ou bien l'aspect multi-disciplinaire d'une base de connaissances.

L'approche multi-disciplinaire est traitée dans la plupart des représentations à objets par la multi-spécialisation de classes dans une seule hiérarchie, solution qui complique la structure des classes et donne lieu aux problèmes d'héritage multiple d'attributs. Outre cette représentation implicite des perspectives, quelques systèmes rendent explicites certains des aspects d'une perception multi-points de vue du monde tels que le groupement des attributs dans les instances selon la perspective caractérisée (KRL), le traitement indépendant des différentes perspectives d'un objet (LOOPS), la possibilité de parcourir une partie du graphe des classes selon l'intérêt de l'utilisateur (ROME) et enfin la possibilité de créer des taxinomies différentes de classes pour chaque perspective et de les mettre en correspondance par des points communs pour faire un raisonnement multi-disciplinaire (VIEWS).

Le point de vue à partir duquel on regarde un objet peut aussi déterminer sa décomposition : deux observateurs peuvent voir deux ensembles différents de composants, donnant lieu aux décompositions multiples (multi-points de vue). De plus un composant peut être composé par d'autres objets, ce qui donne lieu à une décomposition multiple imbriquée. Pour pouvoir faire des inférences correctes sur une décomposition imbriquée, la sémantique de la relation de décomposition doit être la même dans tous les niveaux de décomposition. De plus, les décompositions différentes d'un objet peuvent être complètement indépendantes et ne partager aucun composant ou bien avoir des composants ou parties de composants communs ; dans ce dernier cas, les inconsistances doivent pouvoir être détectées dès la création de la base de connaissances.

Il est important de noter que les notions de point de vue et de décomposition sont des notions orthogonales, bien que certains systèmes, comme LOOPS et VIEWS, utilisent un même mécanisme ou des mécanismes similaires pour les représenter.

Dans le chapitre 5 nous allons décrire TROPES, un modèle de représentation de connaissances par objets multi-points de vue. TROPES partitionne le monde en des concepts ou familles ; chaque famille peut être décrite selon différents points de vue. Les points de vue de TROPES intègrent les divers apports de systèmes décrits dans (§ 3.2). Dans TROPES un objet peut avoir différentes décompositions dans les différents points de vue. Ces décompositions sont décrites au niveau des classes d'objets. La sémantique donnée à la relation de composition est celle de composant - tout (§ 2.5.1) qui donne lieu à une structure de décomposition arborescente. Dans Tropes les points de vue et les décompositions sont deux notions différentes, orthogonales et décrite par des mécanismes différents.

Bibliographie

[ATT&86] ATTARDI G., SIMI M., *A Description-Oriented Logic for Building Knowledge Bases*, in Proceedings of the IEEE, vol. 74, n°. 10, pp.1335-1344, octobre 1986.

[BLA&87] BLAKE, E., COOK S., *On including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk*, ECOOP 87, AFCET, Paris, juin, pp.45-54, 1987.

[BOB&77] BOBROW D.G., WINOGRAD T., *An overview of KRL, a Knowledge Representation Language.* Cognitive Science, vol. 1, n°. 1, pp.3-45, 1977.

[BOB&80] BOBROW D.G., GOLDSTEIN P., *Description for a Programming Environment*, in Proceedings of the AAAI, Stanford University, CA, pp.187-189, 1980.

[BOO90] BOOCH G., *Object Oriented Design with Applications.* Benjamin/Cummings Readings, MA, 1990.

[CAR&89] CARRE B., *Méthodologie orientée objet pour la représentation des connaissances* Thèse d'informatique, Laboratoire d'Informatique Fondamentale de Lille, 1989.

[COR86] CORELLA F., *Sémantique retrieval and Levels of Abstraction*, Expert Databasie Systems, Larry Kerschberg (ed.), The Bejamin / Cummings Publishing Company Inc., pp.91-114, 1986.

[DAV87] DAVIS H.E., *VIEWS : Multiple Perspectives and Structured Objects in a Knowledge Representation Language*, Bachelor and Master of Science Thesis, MIT, 1987.

[FER88] FERBER G.J., *Coreferentiality : The Key to an Intentional Theory of Object Oriented Knowledge Representation.* in Artificial Intelligence and Cognitive Science chapitre 6, J. Demongeot, T. Hervé, V. Rialle et C. Roche (éd.), Manchester University Press, 1988.

[FIK&85] FIKES R., KEHLER T. *The Role of Frame-Based Representation in Reasoning.* Communications of the ACM, vol. 28, n°. 9, pp.904-920, 1985.

[GOO79] GOODWIN J.W., *Taxonomic Programming with KLONE*, Linköping University, Informatics Lab. R.R., février. 1979.

[MAR79] MARTIN W.A, *Descriptions and Specialization of Concepts* in Artificial Intelligence. An MIT Perspective vol. 1, pp.377-419, P.H.Winston, R.H. Brown. (éd.), The MIT Press, 1979.

[MIN75] MINSKY M. *A Framework for Representing Knowledge.* in The Psychology of Comp Vision , P.H. Winston (éd.), McGrawHill, New York, chapitre 6, pp.156-189, 1975.

[MIN83] MINSKY, M. , *Why People Think Computers Can't*, Technology Review, décembre 1983.

[NGU&91] NGU A., WONG L., WIDJOJO S., *On Canonical and Non-canonical Classifications*, 2° International Conference on Deductive and Object-oriented Databases, Munich, Germany, décembre 1991, LNCS 566, pp.371-390, 1991.

[PAP&81] PAPALASKARIS M.A., SCHUBERT L., *Parts Inference : Closed and Semi-closed Partitionings Graphs*, Proceedings 7th. IJCAI, pp.304-309, 1981.

[SCH79] SCHUBERT L., *Problems with Parts*, Proceedings 6th. IJCAI, pp.778-784, 1979.

[SCH&83] SCHUBERT L., PAPALASKARIS M.A., THAUGHER J., *Determining Type, Part, Color, and Time Relationships*, IEEE Computer, vol. 16, n°. 10, pp.53-60, octobre, 1983.

[SHA91] SHASTRI L., *Why Semantic Networks*, in Principles of Semantic Networks. Explorations in the Representation of Knowledge. J.Sowa (éd.), Morgan Kaufman Publishing, chapitre 3, pp.109-136, 1991.

[STE&85] STEFIK M., BOBROW D.G., *Object-Oriented Programming : Themes and Variations.* The A.I. Magazine, vol. 6, n°. 4, pp.40-62, 1985.

[SUS&80] SUSSMAN G.J., STEELE G.S. Jr., *CONSTRAINTS— A Language for Expressing Almost-Hierarchical Descriptions*, Artificial Intelligence, vol. 14, pp. 1-39, 1980.

[ULL88] ULLMAN J., *Principles of Data based and Knowledge based Systems*, vol. 1, Computer Science Press, Maryland, 1988.

[VAR89] VARELA F. *Connaître : les Sciences Cognitives. Tendances et Perspectives*, SEUIL, Paris, 1989.

[WIN&87] WINSTON M.E., CHAFFIN R., HERRMANN D., *A Taxonomy of Part-Whole Relations*, Cognitive Science, vol. 11, pp. 417 - 444, 1987.

[WOL&91] WOLINSKI F., PERROT J.F., Representation of Complex Objects : Multiple facets with Part-Whole Hierarchies, ECOOP'91, Springer Verlag, Geneve, Suisse, juillet, pp.288-306, 1991.

[ZEI84] ZIEGLER B.P., Multifaceted Modeling Methodology : grappling with the irreductible Complexity of Systems, Behavioral Science, vol. 29, pp.167-178, 1984.

Chapitre 4

Raisonnement par classification

Chapitre 4

Raisonnement par classification

> "Le savoir naturel est d'abord classificatoire. […] . Du point de vue de la relation avec l'environnement, classer signifie reconnaître un certain nombre de discontinuités vitales : au moins être capable de distinguer ce qui est comestible de ce qui ne l'est pas. L'activité classificatoire représente la condition minimale de l'adaptation" [VOG88].

Introduction

Nous avons vu auparavant que la connaissance peut être représentée avec différentes techniques telles que les règles de production, les schémas ou les réseaux sémantiques, et que pour chacun de ces types de représentation il existe des mécanismes de raisonnement spécialement adaptés : la déduction par *modus ponens* pour la logique, le raisonnement hypothético-déductif pour les règles de production, l'analogie pour les représentations par prototypes, etc. Pour les représentations à objets, le mécanisme d'inférence le plus adapté est la classification.

La classification n'est pas seulement une des activités les plus puissantes du raisonnement humain, mais aussi un mécanisme fondamental d'inférence [CLA85]. Ce mécanisme est spécialement adapté aux représentations à objets. En effet, la structuration de la connaissance en classes, sous-classes et instances favorise l'utilisation de la classification pour récupérer les connaissances implicites, des relations entre une nouvelle situation et des situations déjà connues.

Le terme *classification* a été utilisé pour désigner trois types de mécanismes :

- la catégorisation c'est-à-dire le regroupement d'objets en classes,
- la classification de classes ou insertion d'une nouvelle catégorie ou classe dans un graphe de classes,
- et enfin la classification d'instances qui consiste à trouver, dans le graphe de classes, la classe d'appartenance la plus appropriée pour une instance. Notre travail concerne ce troisième type de classification.

Dans ce chapitre nous présentons le raisonnement par classification. Dans la première section nous montrons l'importance de la classification comme mécanisme de raisonnement de l'être humain. Dans la section 2, nous décrivons les trois types de classification présents dans une représentation par objets : la catégorisation, la classification de classes et la classification d'instances. La classification dans une base de connaissances dépend de la sémantique donnée aux éléments de la représentation (catégories et individus) ainsi qu'aux relations qui existent entre ces éléments (spécialisation, subsomption, appartenance, etc.). Ces éléments ainsi que les composants principaux de l'algorithme de classification, le parcours du graphe et l'appariement d'objets, sont décrits dans la section 3 du chapitre. La section 4 étudie la classification de classes, plus particulièrement la classification de concepts dans les logiques terminologiques et la classification par insertion de types dans des systèmes associant un

type à chaque classe. Enfin, dans la section 5, nous montrons les idées principales de la classification d'instances, et nous décrivons quelques systèmes qui utilisent ce mécanisme de raisonnement. Nous concluons par un résumé des étapes et caractéristiques principales des algorithmes de classification dans les représentations à objets.

4.1. La classification : un mécanisme de raisonnement

La classification provient du besoin universel de décrire, dans tout domaine de discours, les régularités de collections d'instances [WEG87]. Dès que l'enfant commence à manipuler les opérations concrètes, il groupe des objets réels dans des collections différentes selon certaines caractéristiques visuelles. La création de ces catégories d'objets et leur groupement dans une structure allant des catégories générales aux catégories plus spécifiques, permet à l'être humain d'organiser sa connaissance des objets du monde.

Dans les différents domaines de la science, l'être humain structure les objets en classes :

- ainsi, par exemple, en sciences de la nature on trouve des taxinomies animales, minérales et végétales ;
- en mathématiques, le concept de classes d'équivalence permet de construire des structures hiérarchiques de classes d'objets abstraits (par exemple la classe "équivalent mod 10" est sous-classe de la classe "équivalent mod 2") ;
- en programmation, les différentes versions d'un programme peuvent être organisées dans une hiérarchie d'affinement, idée prise par PIE [BOB&80], etc.

Bien que la création de classes à partir d'un groupe d'exemples soit une activité primordiale pour la compréhension d'un domaine de connaissance particulier tel que la botanique et la minéralogie, la puissance principale du raisonnement par classification concerne la classification d'un individu dans une structure de classes déjà créée. Selon Clancey [CLA85], la résolution de problèmes inclut des étapes de classification. Clancey explique que pour résoudre un problème, un système expert commence par le classer dans une hiérarchie de problèmes connus pour trouver un problème connu semblable. Par une association heuristique, il trouve la solution de ce problème connu dans une hiérarchie de solutions, puis il affine cette solution, par une autre procédure de classification, pour la faire correspondre au problème à résoudre (Fig. 4.1.).

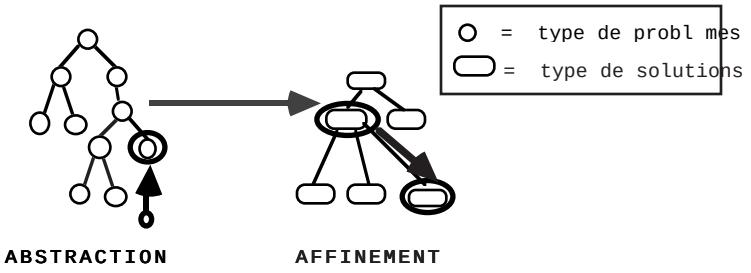


Fig. 4.1. Pour résoudre un problème, une personne le classe dans une structure de problèmes connus, lui associe le type de solution correspondant à sa place dans cette structure, puis il affine cette solution en la descendant dans la structure de solutions.

Ce processus, que Clancey appelle **classification heuristique**, est utilisé par des systèmes experts comme MYCIN (expert en diagnostic médical). MYCIN fait une liaison heuristique entre une caractérisation abstraite du patient selon ses différents symptômes et une classification des maladies possibles (Fig. 4.2). L'association heuristique est faite par des règles liant un groupe de symptômes à une famille de maladies.

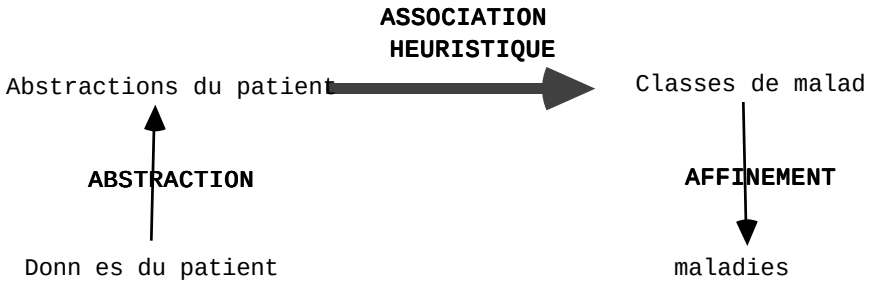


Fig. 4.2. Raisonnement du système expert Mycin : pour trouver la solution, le système suit trois étapes : abstraction des données du patient, association heuristique entre des classes de symptômes et des classes de maladies et affinement dans la structure de maladies pour trouver la classe de maladies la plus spécifique possible.

La classification est donc aussi bien un moyen de structurer la connaissance d'un domaine qu'un mécanisme de raisonnement ; ce mécanisme peut être utilisé comme méthode globale de solution d'un problème classificatoire ou bien comme une partie d'un processus de classification du problème - association heuristique avec des solutions possibles et affinement de la solution.

4.2. Classification dans les représentations à objets

Comme nous avons dit précédemment, en intelligence artificielle, le terme classification a été utilisé avec trois significations. Une première acception du terme correspond à la création de classes à partir d'exemples, c'est-à-dire au processus de division d'un ensemble d'individus d'un univers de discours en diverses classes selon différents critères, de façon à grouper dans une même classe des objets semblables (Fig. 4.3). Ce processus, aussi nommé **catégorisation** [NAP91], est utilisé pour la catégorisation conceptuelle, l'apprentissage et l'acquisition automatique de connaissances [AGU89], [MIC&86], [GEN&89].

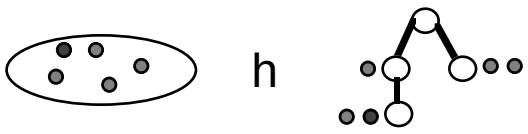


Fig. 4.3. Catégorisation : construction du graphe à partir des instances.

Dans les deux autres acceptions du terme, la classification consiste à trouver la localisation d'une nouvelle connaissance dans une base de connaissances déjà existante, qui est organisée en une structure de classes. Selon le type de connaissance que l'on veut ajouter à la base de connaissances, on distingue deux niveaux de classification : la

classification d'instances ou classification du premier ordre et la classification de classes ou classification du deuxième ordre [WEG87].

Classer une instance consiste à trouver les classes pour lesquelles elle satisfait les contraintes (Fig. 4.4). Etant donné un individu particulier de l'univers du discours et une structure de classes, la classification consiste ici à trouver les classes les plus spécialisées pour lesquelles l'instance satisfait les contraintes (les contraintes dans une représentation par objets sont décrites par des facettes de contraintes : sauf, domaine, intervalle, cardinalité, etc. § 2.2.3). Cette procédure entraîne des comparaisons entre une classe en tant que type d'une catégorie d'objets et un objet particulier, l'instance.

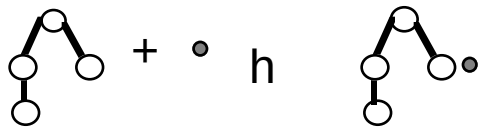


Fig. 4.4. Classification d'une instance dans le graphe de classes.

La classification d'instances, ou classification du premier ordre, est l'opération de manipulation la plus importante d'une base de connaissances structurée en classes ; elle joue aussi un rôle fondamental dans le raisonnement suivi pour résoudre un problème : en effet, la classification heuristique décrite par Clancey (§ 4.1) est une classification de premier ordre où un problème spécifique est localisé dans une structure de catégories de problèmes déjà existante.

La **classification d'une classe** ou classification de deuxième ordre, consiste à ajouter une nouvelle classe à une base de connaissances, de façon à respecter l'ordre partiel existant entre les classes (Fig. 4.5). Ce type de classification peut demander des modifications des liens de spécialisation entre classes, la mise à jour de la cohérence de la base, et éventuellement la modification des instances déjà stockées. La classification de classes est avant tout une opération de construction et de maintien de la base de connaissances ; dans certains systèmes, elle est utilisée pour faire des requêtes sur les objets de la base de connaissances [BOR&89].

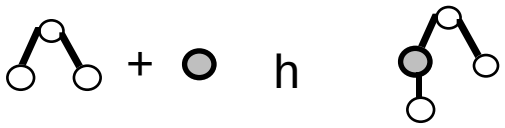


Fig. 4.5. Classification d'une classe dans le graphe de classes.

La classification d'instances est le mécanisme de raisonnement principal des représentations de connaissances à objets ayant l'approche classe / instance [REC88] ; en effet, dans une grande base de connaissances, ce mécanisme permet d'utiliser la connaissance existante pour résoudre un problème nouveau.

La classification de classes, pour sa part, sert surtout à mettre à jour la connaissance de la base et à faire respecter la cohérence de cette connaissance. Dans certains systèmes, comme les logiques terminologiques, la classification de concepts (correspondant aux classes dans les RCO) est le mécanisme de raisonnement principal [SCH&83], [McGRE91]. Enfin, ces deux types de classifications peuvent être utilisés pour la recherche d'information ; l'objet à classer joue alors le rôle d'un filtre d'instances.

Dans la suite du chapitre nous allons discuter plus en détails de la classification de classes et de la classification d'instances ou individus.

4.3. Composants de la classification

Le terme classe est couramment utilisé dans les représentations par objets pour décrire une structure particulier, un filtre qui désigne un ensemble d'individus semblables ; une base de connaissances étant un graphe de classes. Ce même type de connaissances est appelé dans les logiques terminologiques un concept. Par la suite nous allons utiliser le terme classe lorsque nous nous plaçons dans le cadre des représentations par objets et le terme concept pour les logiques terminologiques. Pour parler, de façon général, de la classification d'un élément du graphe (classe dans les RCO, concept dans les LT) nous allons utiliser le terme **catégorie**.

La classification, qu'elle soit comprise dans le sens classification de catégories ou classification d'instances, fonctionne sur une base de connaissances structurée en termes de catégories d'objets. Une catégorie groupe un ensemble d'individus semblables. Les catégories sont organisées dans un graphe selon un **ordre partiel** préétabli. Dans une telle structure, la classification d'un nouvel objet consiste à trouver sa place dans la structure, c'est-à-dire à rendre explicite les relations (d'ordre dans le cas de la classification d'une catégorie, d'appartenance dans le cas d'une instance) existantes entre ce nouvel objet et les catégories du graphe. L'algorithme de classification comporte deux étapes principales : le **parcours du graphe** des catégories existantes et, tout au long du parcours, la comparaison entre l'objet à classer et une catégorie particulière du graphe, processus appelé **appariement**.

Par la suite nous allons discuter plus en détails ces quatre aspects : les catégories, l'ordre partiel entre elles, le parcours du graphe et l'appariement.

4.3.1. Les catégories et leur relation d'ordre

L'interprétation donnée au terme catégorie, à la relation d'appartenance d'un objet à une catégorie et à la relation d'ordre entre catégories, détermine le fonctionnement du mécanisme de classification et les inférences que l'on peut en tirer. Dans cette partie nous rappelons les différents sens donnés à ces termes.

Une catégorie décrit un ensemble d'objets semblables. Une première façon de décrire cet ensemble est de le faire explicitement en donnant l'**extension** de la catégorie. Cette solution ne sert que pour des catégories décrivant des ensembles finis d'objets, et même dans ce cas, la représentation de la catégorie par ses membres rend difficile sa manipulation et le raisonnement que l'on peut faire avec. Une deuxième façon de décrire un ensemble d'objets est par un objet typique de l'ensemble (§ 1.3.4) ; cette approche **prototypique** pose des problèmes lors de la classification (§ 2.6.2). Enfin, une catégorie d'objets semblables peut être décrite en donnant sa définition (§ 2.2.4). Cette approche **intensionnelle** est la plus utilisée et la seule qui serve pour faire du raisonnement par classification.

L'approche intensionnelle suppose que pour toute catégorie, on puisse donner une définition générale de ses éléments ; cette définition peut être fonctionnelle ou structurelle. La définition fonctionnelle consiste à associer à la catégorie un prédicat logique sous forme d'une **fonction** permettant de filtrer les objets de la catégorie. La définition structurelle consiste à donner une **structure** ou type avec les propriétés que doivent avoir les objets de la catégorie et les contraintes qu'ils doivent satisfaire.

Définition fonctionnelle

Une catégorie peut être définie par un prédicat d'appartenance sous la forme d'une fonction qui s'applique aux éléments de l'univers du discours (les instances potentielles de la base de connaissances) et qui rend une valeur permettant de savoir la relation existante entre l'objet et la catégorie.

Fonction booléenne d'appartenance

Dans la logique classique, une fonction d'appartenance correspond à un prédicat qui rend la valeur *vrai* pour les éléments de la catégorie et la valeur *faux* pour les autres.

Ainsi, soit U l'univers du discours, et une catégorie C d'objets de cet univers, la fonction d'appartenance est définie par

f_C : U -> { vrai, faux }

f_C (x) = vrai ssi x relève de la catégorie C

L'extension de C est { x | f_C (x) = vrai }

Par exemple : la catégorie des nombres multiples de dix peut être décrite dans l'univers des entiers par le prédicat :

multiple_dix (x) = si (x mod 10 = 0) alors VRAI
sinon FAUX.

Fonction à trois valeurs pour la relation d'appartenance

La logique classique et ses prédicats à deux valeurs est bien adaptée pour la prise en compte d'un monde complet et fermé dans lequel on peut déterminer, pour tout objet du monde, s'il appartient à une catégorie donnée ou pas. Or, la plupart des systèmes à base de connaissances travaillent avec des connaissances incomplètes. La portée de la fonction d'appartenance est augmentée ici pour accepter la valeur "possible" (ou inconnue). Le prédicat rend "possible" pour une instance incomplète si la connaissance que l'on a de l'instance ne permet ni d'affirmer ni de nier son appartenance à la catégorie :

f_C : U -> { vrai, faux, possible }

f_C (x) = vrai ssi x relève de la catégorie C

f_C (x) = faux ssi x ne relève pas de la catégorie C

f_C (x) = possible ssi on ne peut pas affirmer f_C (x) = vrai ni on ne peut pas affirmer f_C (x) = faux

L'extension de la catégorie C a au moins tous les éléments sûrs, et au plus tous les éléments sûrs et tous les éléments possibles :

si vrai = { x | f_C (x) = vrai } et poss = { x ∈ U | f_C (x) = possible }

alors vrai ⊆ C ⊆ vrai ∪ poss

Relation d'ordre entre catégories

Les catégories d'une base de connaissances sont organisées dans un graphe induit par une relation d'ordre partiel, noté ≤. Une catégorie C est inférieure pour l'ordre à une catégorie D, noté C ≤ D (D est un prédécesseur de C et C est un successeur de D), si l'ensemble d'éléments potentiels de la catégorie C est inclus dans l'ensemble des éléments potentiels de la catégorie D. Avec la définition par prédicat C ≤ D ssi f_C entraîne logiquement (=>) f_d ; autrement dit :

C ≤ D ssi ∀ x ∈ U, f_C(x) = vrai => f_d(x) = vrai

Définition structurelle

Définir une catégorie par sa structure consiste à donner toutes les propriétés qu'ont les membres de cette catégorie et à indiquer, pour chaque propriété, son type (le type d'une propriété est décrit par son domaine de valeurs et éventuellement par des contraintes dynamiques exprimées sous forme de prédicats). Si la définition est complète, les propriétés établissent des conditions nécessaires et suffisantes d'appartenance à la catégorie : elles sont individuellement nécessaires et collectivement suffisantes ; c'est-à-dire qu'un objet doit les satisfaire pour être membre de la catégorie, et tout objet les satisfaisant en est un membre (§ 2.2.4). La définition structurelle de la catégorie établit ainsi une description qui sert d'une part à vérifier si une instance appartient à une catégorie, et d'autre part à créer des objets de cette catégorie avec la structure adéquate.

Vérifier l'appartenance d'une instance à une catégorie consiste ici à vérifier que l'instance ait des valeurs valides (qui satisfont le type) pour toutes les propriétés de la catégorie.

Deux valeurs pour la relation d'appartenance

De même que pour la définition fonctionnelle, lorsque l'on manipule des objets pour lesquels on a toute l'information, on peut dire pour un objet et une catégorie donnée si l'objet appartient à la catégorie : il suffit de vérifier qu'il a des valeurs pour toutes les propriétés de la catégorie et que ce[illegible]alident les contraintes imposées par elle. Si ce n'est pas le cas, l'objet n'app[illegible]atégorie.

Ainsi, soit C une catégorie déf[illegible]par $\{p_1$[illegible] $p_2 : T_2, \ldots, p_n : T_n\}$, où T_i est le type de la propriété p_i. Soit O un objet [illegible]ni par $\{$[illegible]$: v_1, p_2{:}v_2, \ldots, p_n : v_n, \ldots, p_x : v_x\}$ où v_i est la valeur de O pour la propri[illegible] ; alors, O appartient à l'ensemble décrit par C si et seulement si toutes ses valeu[illegible]our les propriétés de C satisfont le type établit dans C, c'est-à-dire:

SUR	O ∈ [illegible] $\leq n$, v_i satisfait T_i
IMPOSSIBLE	O ∉ [illegible] $i \leq n$ tq. v_i ne satisfait pas T_i

Trois valeurs pour la relation d'appartenance

Selon la définition structurelle de catégorie donnée précédemment, un objet appartient à une catégorie si et seulement si pour toutes les propriétés de la catégorie il possède des valeurs valides. Cette approche classique suppose que l'on a toujours toute la connaissance d'un objet. Cependant, dans la plupart des cas, on doit travailler avec des objets incomplets, c'est-à-dire des objets n'ayant pas de valeurs pour toutes les propriétés de la catégorie.

La relation entre un objet e[illegible]catégorie[illegible]'objets doit être étendue pour inclure des objets incomplets pour lesquels o[illegible]peut in[illegible]antir son appartenance à la catégorie ni la nier. En effet, si les propriétés [illegible]jet incomplet O valident les contraintes correspondantes d'une catégorie [illegible]ne peut p[illegible] affirmer que l'objet appartienne à cette catégorie (car les propriétés ma[illegible]ntes pour[illegible]ent ne pas satisfaire ses contraintes) ; mais on ne peut pas l'exclure [illegible]plus (ca[illegible]ses propriétés valuées satisfont bien les contraintes correspondantes de l[illegible]gorie).

SUR	O ∈ [illegible] $\leq n$, v_i satisfait T_i
IMPOSSIBLE	O ∉ [illegible] i [illegible]n tq. v_i ne satisfait pas T_i
POSSIBLE	O ?∈ [illegible]

$\forall i, 1 \leq i \leq n$, v_i satisfait T_i ou $v_i = \perp$ et $(\exists j, 1 \leq j \leq n$ tq. $v_j = \perp)$[1]

[1]Le symbole ⊥ est utilisé ici pour indiquer une valeur inconnue, manquante.

Relation d'ordre entre catégories

Une catégorie C est inférieure pour l'ordre à une catégorie D, noté $C \leq D$ (D est un prédécesseur de C et C est un successeur de D), si l'ensemble des éléments potentiels de la catégorie C est inclus dans l'ensemble des éléments potentiels de la catégorie D. Avec la définition structurelle $C \leq D$ ssi la structure de C est plus restrictive que celle de D, c'est-à-dire les propriétés et contraintes de D sont incluses dans la description de C : C enrichit D par l'ajout de contraintes aux propriétés existantes ou par l'ajout de propriétés.

Ainsi, soit D une catégorie définie par $\{p_1 : Td_1, p_2 : Td_2, \ldots, p_n : Td_n\}$ et

C une catégorie définie par $\{p_1 : Tc_1, p_2 : Tc_2, \ldots, p_m : Tc_m\}$ où Tx_i est l'ensemble de contraintes de la propriété p_i pour la catégorie X,

alors $C \leq D$ ssi $m \geq n$ et $\forall i : 1..n$, Tc_i est au moins aussi restrictif que Td_i , c'est-à-dire Tc_i impose les mêmes contraintes de Td_i plus éventuellement d'autres ($Tc_i \leq_t Td_i$).

La relation de sous-typage $\leq_t$ permet de construire la d'ordre entre classes lorsque celles-ci n'ont pas de contraintes dynamiques.

Certains systèmes, dont notre système TROPES, établissent une relation d'ordre stricte entre classes qui correspond à l'inclusion stricte d'ensembles. Pour ces systèmes la relation d'ordre une classe C est inférieure à une autre classe D dans l'ordre, si dans C il y a plus de propriétés ou si il y a les mêmes propriétés mais au moins une de ces propriété est plus restreinte dans C que dans D.

$C \leq D$ ssi $m \geq n$ et $\forall i\ 1 \leq i \leq n$, $Tc_i \leq_t Td_i$ (et si $m = n$ alors $\exists\ i\ 1 \leq i \leq n$ tel que $Tc_i \neq Td_i$)

Définition floue

Des systèmes acceptant que des instances d'une catégorie aient différents niveaux d'appartenance à la catégorie établissent des relations d'appartenance valuées, soit par degrés qualitatifs ou quantitatifs d'appartenance en utilisant des calculs probabilistes [GRA88], soit par la définition de catégories floues et l'utilisation de logiques floues [ROS90], [CAY82]. Dans ce dernier type de systèmes, une catégorie est composée d'une partie définitionnelle établissant des conditions nécessaires et une partie typique qui détermine, pour chaque attribut, le domaine de valeurs les plus fréquemment trouvées chez une instance de la catégorie. Ainsi une catégorie définit pour chaque attribut une zone possible et une zone typique. La relation entre une instance et une catégorie est donnée par un "degré" d'appartenance calculé en termes des valeurs possibles et crédibles de l'instance et leur localisation dans les zones possibles et typiques de la catégorie.

Pour la suite de la discussion, nous allons utiliser la définition structurelle : une catégorie est décrite par un ensemble de propriétés et de contraintes et un individu particulier par un ensemble de propriétés valuées.

4.3.2. Graphe de catégories

Dans une base de connaissances, les catégories sont structurées selon une relation d'ordre partiel. Auparavant (§ 4.3.1), nous avons défini la relation d'ordre partiel $\leq$ entre deux catégories, en termes de leur structure: étant donné deux catégories C et D, soit elles ne sont pas comparables (par exemple si C a une propriété p qui n'est pas présente dans D et D en a une, q, qui n'est pas présente dans C), soit on peut déterminer par un algorithme d'appariement (§ 4.3.3) si $D \leq C$ ou si $C \leq D$.

Description explicite des relations d'ordre partiel

La relation $C \leq D$ peut être décrite explicitement dans la base de connaissances par un lien de spécialisation (dans les langages terminologiques, par un lien de subsomption) allant de C vers D. Ainsi, pour un même domaine de connaissances, on peut construire une base de connaissances dans laquelle toutes les relations d'ordre entre catégories sont données explicitement, ou bien on peut avoir une base de connaissances "plate" dans laquelle toutes les relations d'ordre sont gardées implicites et doivent être récalculées à chaque fois. La première option est très utile lorsque l'on a une base statique, dont les catégories n'évoluent pas. En effet, cette approche répond efficacement à la question "$C \leq D$?" pour C et D données, car il suffit de regarder s'il y a un lien entre C et D. Par contre, un changement dans la description d'une catégorie entraîne en général la révision d'une grande partie des relations de la base. Dans la deuxième approche, le changement de la description d'une catégorie n'a pas de conséquences pour les autres connaissances de la base, mais la relation d'ordre entre deux catégories C et D doit être recalculée lors de chaque requête du style "$C \leq D$?".

L'option intermédiaire la plus utilisée consiste à représenter explicitement les relations d'ordre directes et à garder implicites celles que le système peut facilement déduire par les propriétés de réflexivité et de transitivité de la relation [NEB90]. La relation de préséance $<<$ ainsi obtenue est la plus petite relation dont la fermeture transitive et reflexive soit identique à la relation $\leq$, le graphe de couverture de la relation d'ordre. Cette relation est unique [AHO&74].

Pour un ordre partiel, $\leq$ sur un ensemble P, $<<$ désigne la relation de préséance de $\leq$,

$x << y$ ssi $x \leq y$ et $\neg\exists\ z, z \neq x, z \neq y$, tq. $x \leq z \leq y$.

Si $x \leq y$ on dit que x est un successeur de y et y est un prédécesseur de x. Si $x << y$, on dit que x est un successeur direct de y et y est un prédécesseur de x .

Description explicite des catégories intermédiaires

Dans la section § 2.3, nous avons présenté différentes structures du graphe de catégories : treillis, treillis complet, arbre, etc. Ces structures ont des propriétés mathématiques qui peuvent être utilisées par l'algorithme de parcours du graphe. Ainsi, par exemple, une structure d'arbre permet un parcours d'ordre logarithmique sur la base, tandis que dans une structure de treillis complet, ce parcours est plus coûteux. Par contre, un treillis complet contient explicitement toutes les catégories qui peuvent se former par agrégation ou intersection des catégories de base du domaine. Certains systèmes forcent la structure de treillis par l'ajout de catégories artificielles qui n'ont pas de sens dans le domaine, mais qui complètent le treillis, comme la catégorie inconsistante, $\perp$, et les catégories union et intersection de tout couple de catégories du domaine [MIS&89].

Partition du graphe dans des familles disjointes et distinction des points de vue

Une grande base de connaissances comporte des connaissances sur différentes familles d'objets. Dans l'approche classique, les représentations par objets structurent toutes les catégories de la base dans un graphe ayant comme racine la catégorie universelle, THING ou OBJET. Certains systèmes font une première partition de la base dans des familles disjointes de catégories[1], telles que "appartement" , "locataire" , etc. (§ 5.1.1) [BOB&77], [BOR&89]. La classification, dans ce groupe de systèmes, part du fait que l'on sait dès le départ à quelle famille appartient l'objet à classer et que l'on réduit ainsi l'espace de recherche au graphe de catégories de cette famille. Enfin, à l'intérieur d'une famille de catégories, les attributs peuvent être structurés en groupes selon les

[1]Dans les logiques terminologiques ces familles correspondent aux catégories primitives.

aspects qu'ils décrivent : les points de vue (§ 3). La comparaison entre catégories est établie modulo un de ces groupes d'attributs, ce qui peut réduire en pratique énormément la complexité du processus d'appariement et le nombre de relations d'ordre à considérer (§ 6).

4.3.3. Parcours du graphe

Il existe deux grands groupes d'algorithmes de classification, illustrés par les logiques terminologiques (§ 4.4.1) et les systèmes de types (§ 4.4.2) respectivement. Le premier groupe voit l'objet à classer comme un tout et le manipule comme une unité qu'il doit localiser dans le graphe de catégories. Le deuxième groupe le voit comme une agrégation d'attributs et classe chaque attribut dans le graphe de types sous-jacent au graphe de catégories, l'union de ces classifications d'attributs lui permettant d'identifier la place correcte de l'objet entier dans le graphe de catégories.

Que l'on parle d'une classification d'un objet dans un graphe de catégories ou d'une classification d'un attribut d'un objet dans un graphe de types, la classification doit faire un parcours du graphe correspondant. Le but de ce parcours est d'obtenir la relation existante entre le nouvel objet et chaque objet du graphe (pour la classification d'instances cette relation est la relation d'appartenance, et pour la classification de classes c'est la relation de spécialisation ; enfin, pour les attributs c'est la relation de sous-typage) .

La plupart des systèmes divisent le problème de la classification en trois parties (pour la classification d'instance seules la première et la troisième parties sont pertinentes) : la première partie consiste à trouver tous les objets du graphe qui sont plus généraux que l'objet à classer et la deuxième partie consiste à trouver les objets qui sont plus spécifiques. La classification s'achève par une troisième étape dans laquelle le nouvel objet est attaché au graphe en ajoutant des liens de préséance ou d'appartenance.

Les deux premières étapes demandent des parcours du graphe. Une première approximation est l'algorithme de force brute qui parcourt tout le graphe pour chaque étape. Pour la première étape ce parcours produit l'ensemble complet de catégories plus générales, qu'il doit après réduire pour ne garder que les catégories minimales par rapport à la relation d'ordre établie ; pour la deuxième étape l'algorithme produit l'ensemble des catégories plus spécifiques, qui doit être réduit après aux catégories maximales. Des algorithmes plus élaborés parcourent le graphe en profondeur ou en largeur en partant de la racine ; de plus ils marquent les nœuds déjà visités pour ne pas les revoir ; enfin, pour la deuxième étape ils commencent le parcours à partir de l'ensemble minimal obtenu dans la première étape (§ 1.3.5).

4.3.4. Appariement

L'appariement est l'étape principale de la classification ; c'est le processus qui permet de déterminer la relation existante entre une catégorie C de la base et l'objet O à classer. Pour la classification de catégories, l'appariement consiste à établir la relation d'ordre entre deux catégories. La classification d'instances apparie l'instance à classer avec une catégorie de la base pour déterminer la relation d'appartenance : l'appariement ici consiste à vérifier la satisfaction des contraintes de la catégorie par les valeurs de l'instance.

Nous avons montré auparavant (§ 4.3.1) qu'une catégorie est décrite de façon intensionnelle par une liste de propriétés ayant des contraintes. De plus, pour deux catégories C et D, si $C \leq D$ alors les propriétés de D sont présentes dans C (avec, probablement plus de restrictions). Ainsi, plusieurs systèmes ne gardent pas dans la base la description complète d'une catégorie, mais juste la liste des prédécesseurs directs et les propriétés et

contraintes ajoutées à leurs descriptions. De même, l'objet à classer peut être décrit en termes de toutes ses propriétés ou bien en termes des catégories dont on sait déjà qu'elles le précèdent et des propriétés aditionnelles, pas présentes dans ces catégories.

L'appariement comporte normalement deux phases : une première phase de normalisation dans laquelle la description complète de la catégorie et de l'objet à comparer sont obtenues en recopiant les propriétés de leurs prédécesseurs et une deuxième étape dans laquelle ces descriptions normalisées sont comparées pour déterminer la relation entre elles. Certaines améliorations ont été proposées récemment, comme la normalisation partielle pour éviter des vérifications redondantes (§ 4.4.1).

Par la suite (§ 4.4), nous allons présenter deux grands groupes de techniques de classification de catégories en termes des aspects que nous venons de présenter : représentation des catégories, graphe induit par la relation d'ordre, parcours du graphe et appariement. Dans la section suivante (§ 4.5), nous allons faire la même analyse pour la classification d'instances.

4.4. Classification de catégories

La classification de catégories consiste à trouver la localisation la plus appropriée d'une nouvelle catégorie dans une structure de catégories existantes, organisées selon une relation d'ordre. Par la suite nous allons décrire deux grandes familles de systèmes de classification de catégories : les logiques terminologiques et les RCO basés sur une structure de types.

4.4.1. La classification dans les logiques terminologiques

Les logiques terminologiques sont décrits en termes de concepts. Le mécanisme principal de raisonnement est la classification de concepts qui est utilisée pour enrichir la base de connaissances, pour faire des requêtes, pour trouver des inconsistances, etc. Les logiques terminologiques ainsi que leur algorithme de classification, ont été décrits dans la section § 1.3.5. Par la suite nous allons compléter cette description en montrant les différentes façons de mettre en œuvre l'algorithme de classification dans ces systèmes.

Représentation

Ces systèmes comportent un langage terminologique pour la description des concepts du monde et un langage assertionnel pour indiquer les faits valides dans un contexte particulier du monde. Pour la classification de concepts, seul le langage terminologique est concerné. Il permet de déclarer des concepts primitifs et des concepts définis ; les premiers établissent des conditions nécessaires mais non suffisantes et les deuxièmes indiquent des conditions nécessaires et suffisantes d'appartenance.

Un concept est décrit par son nom et sa description[1]. La description d'un concept C comporte une liste de concepts et une liste de rôles avec leurs restrictions [WOO91]. La liste de concepts représente les concepts C_1, C_2, C_3,…,C_k qui subsument explicitement le concept C. Ces concepts, qui peuvent être primitifs ou définis sont appelés concepts primaires de C. Les rôles $r_1 : v_1$,…, $r_n : v_n$ représentent des relations binaires r_i entre le concept C qui possède le rôle et le concept indiqué par les restrictions du rôle v_i.

[1]Certains concepts peuvent être donnés par une description sans nom ; c'est le cas des certaines requêtes dans Classic [PAT&91].

$C = C_1, C_2, C_3,\ldots,C_k / (r_1 : v_1,\ldots,r_n : v_n)$

Les restrictions d'un rôle v_i limitent les valeurs que peuvent prendre les individus associés au concept pour ce rôle ; ces restrictions peuvent parler de son type ou concept (descripteur all), de sa cardinalité (descripteurs at_least, at_most), de son domaine (descripteurs filled_by, one_of), de sa relation avec les valeurs d'autres rôles du concept (descripteur same_as), etc [BOR&89].

Par exemple, le concept primitif Personne peut être décrit avec trois attributs : son nom (chaîne de cardinalité 1), son prénom et sa nationalité (les deux décrits par une chaîne de cardinalité plus grande ou égale à 1)[1] :

```
Personne => ( PRIMITIVE
                (AND ( (ALL nom chaîne)
                       (ATMOST nom 1)
                       (ATLEAST nom 1)
                       (ALL prénom chaîne)
                       (ATLEAST prénom 1)
                       (ALL nationalité chaine)
                       (ATLEAST nationalité 1)))
```

La relation d'ordre entre catégories est la subsomption. Cette relation peut être définie au niveau ensembliste : C subsume D ($D \leq C$) si l'ensemble d'individus dénotés par C contient l'ensemble d'individus dénotés par D (§ 1.3.5) ou bien au niveau structurel : C subsume D si la description de D restreint celle de C (les quatre sortes de restrictions possibles sont décrites dans § 1.3.5). Bien que la subsomption extensionelle fournisse la sémantique de la relation de subsomption, c'est la subsomption structurelle qui est utilisée dans les algorithmes de classification.

Graphe de subsomption de concepts

La base de connaissances des logiques terminologiques comporte les descriptions des différents concepts du domaine. Ces concepts sont comparables par la relation d'ordre décrite précédemment, la subsomption. Les relations de subsomption entre concepts peuvent être données explicitement dans la base ou bien être laissées implicites. Ainsi, on peut avoir pour tout couple de concepts, plusieurs ìmplémentations (Fig. 4.6.a à 4.6.f) [BAA&92] :

a) Toutes les relations de subsomption entre les concepts de la base sont données explicitement et pour tout couple de concepts, le concept union et le concept intersection sont créés et liés à la base ; le concept le plus général est le concept THING et le plus spécifique est le concept contradiction $\perp$. Le graphe résultant est un treillis fortement connexe. Cette structure peut présenter beaucoup de liens de subsomption redondants et de concepts artificiellement créés, ce qui alourdit sa gestion.

b) Pour tout couple de concepts, les concepts intersection et union sont présents dans la base. De plus, tout concept est lié explicitement à son prédécesseur direct dans la base. La structure résultante est un treillis. À la différence de l'alternative précédente, la base n'a pas de liens redondants. L'avantage principal de cette solution est qu'elle simplifie l'algorithme de classification ; en effet, tous les concepts qui peuvent être formés en termes des couples rôle - restriction présentes dans la base, y sont déjà décrits explicitement. Le problème principal est le grand nombre de concepts artificiels qu'il faut créer et leur gestion.

[1] Cette syntaxe est prise du langage Classic.

c) Tous les concepts créés explicitement (descriptions) sont stockés dans la base ainsi que tous leurs liens vers leurs prédécesseurs.

d) Tous les concepts créés explicitement (descriptions) sont stockés dans la base ainsi que leurs liens vers leurs prédécesseurs directs. Cette structure est une des plus utilisées : le graphe résultant est un graphe orienté sans cycles, induit par la relation de préséance (c'est-à-dire la plus petite relation d'ordre dont la clôture soit identique à la relation de subsomption) et ayant comme racine unique le concept primitif THING. Le concept THING subsume des concepts primitifs qui subsument d'autres concepts, primitifs ou définis.

e) Tous les concepts ayant un nom (définitions) sont stockés dans la base ainsi que leurs liens vers leurs prédécesseurs directs. Cette structure est utilisée par le système Classic [PAT&91]. Dans Classic, une requête sur la base est décrite par une description de concept qui, après classement, permet d'identifier les concepts dont les individus satisfont la requête. Stocker toutes les requêtes agrandirait énormément la base, alors le système ne stocke que les définitions (requête avec un nom). Il est important de noter que, pour l'algorithme de classification, cette structure est identique à la précédente, à savoir un graphe acyclique orienté, induit par la relation de préséance ou subsomption directe.

f) Les relations de subsomption ne sont pas données explicitement ; la base de connaissances est une liste de concepts dont les relations de subsomption doivent être calculées par le système à chaque fois qu'il en a besoin.

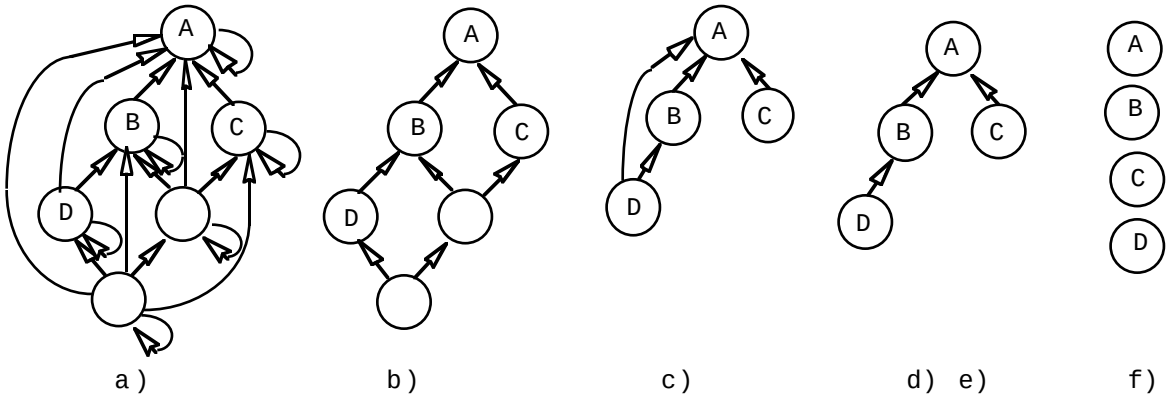


Fig. 4.6. Structures du graphe d'une base ayant les concepts A, B, C et D avec les relations de subsomption B ≤ A, D ≤ B, D ≤ A, C ≤ A (A = THING).

Parcours du graphe

Dans les deux parties précédentes nous avons discuté de la représentation dans les logiques terminologiques. Par la suite nous allons discuter de l'algorithme de classification, le "**classifieur**" (classifieur), qui est utilisé sur cette représentation (en prenant pour la structure du graphe de concepts celle utilisée par Classic (options d) et e) de la section précédente)).

Le classifieur cherche à trouver la place correcte du nouveau concept c dans la base. Cela revient à trouver les prédécesseurs directs de c dans la base (SPS subsumants plus spécifiques) et les successeurs directs dans la base (SPG : subsumés plus généraux), puis à lier le nouveau concept à la base (§ 1.3.5). La dernière partie revient à ajouter des liens des SPG vers c et de c vers les SPG, processus simple qui demande un temps linéaire. Pour les deux premières étapes plusieurs méthodes ont été proposées [BAA&92] :

- **la méthode aveugle** : pour calculer les SPS, la méthode calcule l'ensemble complet de concepts qui subsument c en comparant c avec tous les concepts de la base (∀x, test c

≤ x), puis elle réduit cet ensemble aux concepts dont les successeurs directs ne subsument pas c. Le calcul des SPG est symétrique : calculer l'ensemble de concepts subsumés par c ($\forall$x, test c ≥ x) puis réduire l'ensemble pour ne garder que les concepts dont les prédécesseurs directs ne sont pas subsumés par c. Cette méthode fait O(n) comparaisons, n étant le nombre de concepts de la base.

- **la méthode de parcours simple** : le calcul des SPS se fait en parcourant le graphe en largeur à partir de la racine, THING. Pour tout concept x examiné, l'algorithme valide s'il a un successeur direct y subsumant c (c≤y). Si c'est le cas, il garde ce concept y dans la liste des possibles SPS pour l'examiner après ; si x n'a pas de successeurs directs subsumant c, il est ajouté à la liste de SPS. Les nœuds examinés sont marqués pour éviter de refaire des vérifications. Cet algorithme est celui utilisé par le classifieur de KL-ONE (§ 1.3.5). Le calcul des SPG est le dual, fait à partir du concept contradictoire.
- **le parcours simple amélioré** : l'algorithme précédent peut être amélioré pour tirer parti, d'une part, pendant le calcul des SPS des comparaisons intermédiaires et, d'autre part, pendant le calcul des SPG du résultat obtenu dans la première partie. L'algorithme proposé par [BAA&92] inclut dans le calcul des SPS une vérification : avant de considérer la relation de subsomption entre un nœud x et le concept c, l'algorithme considère la relation existante entre les prédécesseurs directs de x, qui n'ont pas été examinés, et le concept c ; ainsi si un prédécesseur direct de x ne subsume pas c, la comparaison avec x est inutile et elle n'est pas faite. L'amélioration dans le calcul des SPG se fait en réduisant l'espace de recherche initial aux concepts qui sont des successeurs de tous les prédécesseurs directs (de tous les éléments de SPS), car c'est seulement parmi eux que se trouvent les successeurs directs de c. Bien que la complexité de ces deux dernières méthodes dans le pire des cas reste égale à O(n), en pratique l'algorithme est nettement plus efficace [NEB90].

Pour le calcul des SPS l'espace de recherche de ces trois méthodes est toute la base, en partant de la racine THING. Or, seuls les sous-graphes des concepts primitifs apparaissant dans la description du nouveau concept sont concernés par sa classification. En effet, tout concept non primitif subsumé par un concept primitif doit avoir dans sa description normalisée une référence explicite à ce concept primitif (il doit être parmi ses concepts primaires) ; ainsi, si un concept primitif ne fait pas partie des concepts primaires du concept à classer, ni ce concept primitif ni ses successeurs ne sont pas des prédécesseurs de c. Une même analyse est valide pour le calcul des SPS. Ainsi, une dernière amélioration des algorithmes consiste à normaliser le concept à classer et à réduire l'espace de recherche aux graphes de ses concepts primaires.

Appariement entre deux concepts

L'appariement dans la classification de concepts est le calcul qui permet d'établir la relation de subsomption existante entre deux concepts (§ 4.3.1). L'algorithme de subsomption dépend des descripteurs offerts par le langage. Cependant tous les algorithmes comportent des tâches générales comme la vérification des relations de subsomption entre concepts primaires et la vérification de subsomption au niveau des rôles et des descripteurs.

Dans la plupart des systèmes, l'appariement comporte une première étape de **normalisation** des concepts à comparer[1] [NEB90]. La normalisation d'un concept C se fait à deux niveaux, celui des concepts subsumant explicitement C et celui des rôles de C et de leurs descripteurs.

[1]Lorsque la subsomption est utilisée à l'intérieur de l'algorithme de classification, comme c'est le cas qui nous occupe, la normalisation du concept à classer est faite une seule fois au début de la classification.

Normaliser les subsumants explicites consiste à “aplatir” la description du concept C en recopiant toute l’information de ses catégories primaires et en ne laissant comme catégories primaires que des concepts primitifs. Par exemple, si la base de connaissances a le concept primitif Personne (décrit précédemment) et les concepts définis Enfant et Enfant_étranger décrits par :

Personne = (Prénom un chaîne, Nom un chaîne, Nationalité un chaîne)

Enfant = Personne / (Age un entier dom [0..12])

Enfant_étranger = Enfant / (Nationalité NOT française)[1]

alors la normalisation du concept Enfant_étranger donne :

Enfant_étranger = Personne / (Prénom un chaîne ;
Nom un chaîne ;
Age un entier dom [0..12] ;
Nationalité un chaîne NOT française)

La normalisation des descripteurs est faite par une algèbre d’équivalence qui reflète la sémantique des descripteurs [BOR&92]. Cette normalisation enlève les descripteurs redondants et génère des versions équivalentes d’une description donnée. Ainsi, par exemple, le “normaliseur” de CLASP [BOR92b] a les règles suivantes :

(AND (ATLEAST (3,p) , ATLEAST (4,p)) => ATLEAST (4,p)

ATMOST (0,p) => AND (ATMOST (0,p), ALL (p, NOTHING))

La relation de **subsomption** sur les concepts normalisés correspond à la relation d’ordre structurelle présentée dans § 4.3.1. En effet, si le concept D est décrit par

$D = D_1, D_2, D_3,\ldots,D_x / (r'_1 : vd_1,\ldots,r'_z : vd_z)$

et le concept C est décrit par

$C = C_1, C_2, C_3,\ldots,C_k / (r_1 : vc_1,\ldots,r_n : vc_n)$

alors D subsume C ssi

- $k \geq x$, $\forall j, 1 \leq j \leq x, \exists i\ 1 \leq i \leq k$ tel que $Ci \leq Dj$ (tout concept primitif de D subsume un concept primitif de C)

- $n \geq z$, et $\forall j, 1 \leq j \leq z, \exists i\ 1 \leq i \leq n$ tel que $ri = r'j$ et $vci \leq vdj$ (les contraintes dans C pour un rôle sont plus fortes que celles de D pour le même rôle)

Il est intéressant de noter que la normalisation complète des objets à comparer avant la classification n’est pas toujours la solution la plus efficace [BAA&92]. En effet, si la catégorie de la base et l’objet à classer ont un prédécesseur commun, la normalisation complète perd cette connaissance, et la phase de comparaison va comparer des propriétés que l’on sait identiques (celles de ce prédécesseur commun). Supposons, dans l’exemple précédent, que l’on veut établir la relation entre le concept Enfant_étranger et un concept individuel, Paola, décrit par Paola = Enfant + { Prénom = “Paola” ; Nom = “Gianini” ; Age = 5 ; Nationalité = “italienne” }. Avant la normalisation, le système peut utiliser le fait que aussi bien Paola que Enfant_étranger sont subsumés par le concept Enfant et ne pas regarder les descripteurs des rôles hérités de ce concept (comme le prénom, le nom et le type de l’age). Après la normalisation, cette information est perdue et l’algorithme de subsomption doit comparer tous les rôles.

Enfin, la vérification d’inclusion de contraintes des rôles ($vci \leq vdi$) peut être très coûteuse. Sa complexité dépend des descripteurs offerts par le langage et elle peut

[1]On a utilisé ici la notation proposée par [WOO91] pour définir un concept en termes de ses catégories primaires et le type de chaque rôle.

entraîner des processus auxiliaires complexes comme des classifications partielles des concepts anonymes (sans nom, formés par une description) présents dans la description des rôles [NEB90]. Actuellement, plusieurs travaux cherchent à déterminer le point d'équilibre entre un langage expressif et un algorithme de subsomption efficace, la tendance étant vers les langages ouverts, n'offrant dans leurs noyaux que quelques descripteurs de base mais permettant l'ajout d'autres descripteurs et leur validation correcte par la méthode de subsomption [BOR&92].

4.4.2. Classification de classes par des types

Dans le chapitre 2, nous avons décrit les systèmes de représentation de connaissances par objets. Ces systèmes, issus des schémas de Minsky et des langages orientés objets, ne possèdent pas un formalisme permettant de garantir la correction de la base, ni la correction et la complétude des mécanismes d'inférence. La théorie des types avec sa relation bien définie de sous-typage et des algorithmes corrects de vérification de types offrent un cadre adéquat pour formaliser les éléments des représentations par objets [CAR85]. Dans cette partie nous allons présenter l'intégration du formalisme de types dans une représentation par objets proposée par Caponi et Chaillot, ainsi que l'algorithme de classification de classes qui repose sur ce formalisme [CAP&93].

Représentation

La représentation par objets comporte deux éléments de base : les classes et les instances. Les classes sont décrites de façon intensionnelle par une liste d'attributs et contraintes. De même que pour un concept dans les logiques terminologiques, la description d'une classe peut comporter une liste explicite de prédécesseurs ; la structure finale d'une classe, sa **description complète**, est obtenue en recopiant toute l'information des attributs de ses sur-classes (prédécesseurs).

La description complète d'une classe est donc une liste d'attributs et pour chaque attribut sa description complète pour la classe. La description complète d'un attribut pour une classe est donnée en termes des facettes (descripteurs) définies dans la classe ou héritées des sur-classes. Cette description inclut des facettes de restriction de type, de cardinalité, de domaine, etc.[1]. L'union de ces facettes détermine complètement le domaine de valeurs possibles pour l'attribut dans cette classe. Un même domaine de valeurs peut être décrit par différentes combinaisons de facettes (Fig. 4.7). Toutes ces formes équivalentes peuvent être ramenées à une forme normale unique par des règles de réécriture, le type de l'attribut.

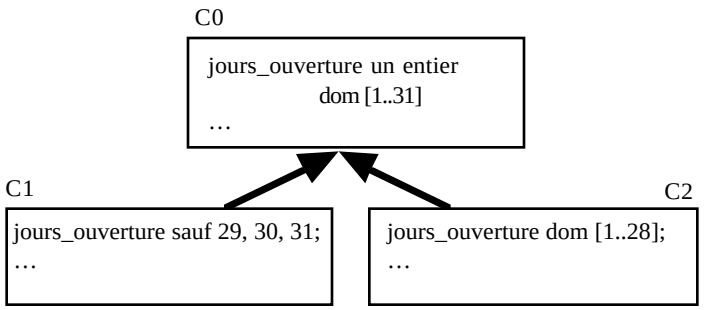


Fig. 4.7. Les définitions de jours_ouverture dans C1 et C2 sont équivalentes même si elles sont exprimées différemment ; en effet, les deux décrivent l'ensemble {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28}.

Le **type d'un attribut pour une classe** détermine le domaine des valeurs possibles pour cet attribut dans la classe. Le domaine d'un attribut peut être concret ou abstrait. Un

[1]Les facettes dynamiques : démons, attachements procéduraux, etc. ne sont pas prises en compte ici.

domaine concret est un ensemble de valeurs de types primitifs (entier, réel, etc.) ou un ensemble de valeurs obtenues par l'application des constructeurs de types (intersection, union, liste) sur les types primitifs. Un domaine abstrait correspond à un sous-ensemble des instances d'une classe.

La relation entre classes est la spécialisation stricte (§ 2.2.6), qui représente l'inclusion ensembliste stricte ; pour les descriptions complètes de classes, cette relation correspond à la relation d'ordre strict correspondant à la relation d'ordre entre descriptions intensionnelles présentée précédemment (§ 4.3.1).

Si D est une description complète de classe définie par $\{p_1 : Td_1, p_2 : Td_2, \ldots, p_n : Td_n\}$ et

C une description complète de classe définie par $\{p_1 : Tc_1, p_2 : Tc_2, \ldots, p_m : Tc_m\}$

où Tx_i est le type de l'attribut p_i pour la classe X,

alors $C \leq D$ ssi $m \geq n$ et $\forall i\ 1 \leq i \leq n$, $Tc_i \leq_t Td_i$ (et si $m = n$ alors $\exists\ i\ 1 \leq i \leq n$ tel que $Tc_i \neq Td_i$)

La relation de sous-typage $\leq_t$ est définie comme la relation d'inclusion portant sur les domaines d'attributs et elle peut être validée à partir des formes normales des types de ces attributs.

Dans le cas où le domaine d'attributs est abstrait (il prend ses valeurs dans un concept de la base), le type de l'attribut est le type d'une classe du concept (ou une restriction de ce type). **Le type d'une classe** est l'enregistrement des types de ses attributs. Dans ce cas, la relation de sous-typage est une relation sur des types d'enregistrements [CAR85] associés aux classes. Cette relation de sous-typage est en accord avec la relation de spécialisation (si C spécialise D alors le type de C est un sous-type de D).

Graphe de classes et graphes de types

Les classes d'une représentation par objets sont organisées selon la relation de préséance << de ≤, qui lie une classe à ses sur-classes directes. De plus, pour tout attribut présent dans la base de connaissances, le système contient ses types pour les différentes classes qui le contiennent. Les types d'un attribut sont organisés dans un graphe de types induit par la relation de sous-typage : à chaque nœud de ce graphe de types sont associés un type et la liste des classes pour lesquelles l'attribut a ce type (Fig. 4.8).

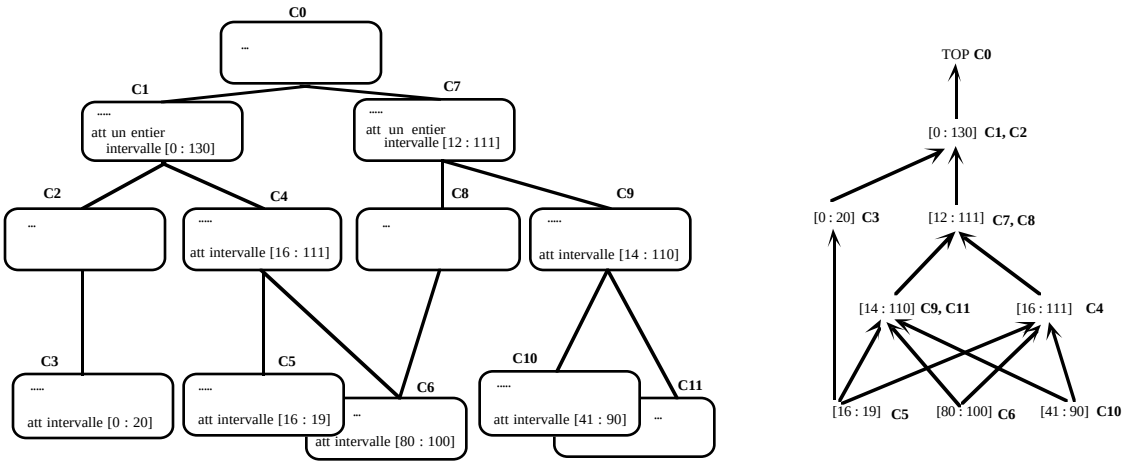


Fig. 4.8. Graphe de spécialisation avec les définitions de l'attribut *att*, et graphe des types de *att* correspondant.(Extrait de [CAP&93])

Parcours des graphes

La position d'une classe dans le graphe de spécialisation peut être calculée à partir des graphes des types de ses attributs. En effet, le graphe des types associé à l'attribut *a* fournit d'une part l'ensemble des sur-classes possibles de C par rapport à l'attribut *a*, *s(a,C)* (les classes associées aux sur-types du type de *a* de C) et, d'autre part, l'ensemble des sous-classes possibles de C, *i(a,C)* par rapport à *a* (les classes associées aux sous-types du type de *a* de C). Les sur-classes directes possibles de C par rapport à *a*, *sup(a,C)* sont les éléments minimaux de *s(a,C)* ; les sous-classes directes possibles de C par rapport à *a*, *inf(a,c)* sont les éléments maximaux de *i(a,C)* .

$$\sup(a, C) = \bigcup_{\substack{(t, l) \in GTa \\ t \geq_\tau t_0}} l \qquad \inf(a, C) = \bigcup_{\substack{(t, l) \in GTa \\ t \leq_\tau t_0}} l$$

où *GTa* est le graphe de types associées à un attribut *a* ; la relation de sous-typage $\leq\tau$ est équivalente à la relation d'inclusion portant sur les domaines des attributs. Un type t_1 est sous-type d'un type t_2 si est seulement si t_1 correspond à un domaine inclus dans le domaine correspondant à t_2 . Une classe est composée de plusieurs attributs. Ainsi, pour qu'une classe D soit sur-classe directe de la nouvelle classe C, tous les attributs de D doivent être présents dans C et D doit être une sur-classe directe possible de C par rapport à chacun de ces attributs. De même pour qu'une classe E soit sous-classe directe de la nouvelle classe C, tous les attributs de C doivent être présents dans E, et E doit être une sous-classe directe possible de C par rapport à chacun de ces attributs. De plus, les sur-classes obtenues doivent être minimales (*sup(C)*) et les sous-classes obtenues maximales (*inf(C)*) :

$$\sup(C) = \left[\bigcap_{a \in A(C)} \sup(a, C) \right] \cap \{ C' \ / \ A(C') \subseteq A(C) \ \}$$

$$\inf(C) = \left[\bigcap_{a \in A(C)} \inf(a, C) \right] \cap \{ C' \ / \ A(C) \subseteq A(C') \ \}$$

L'algorithme de classification d'une classe C fait pour chaque attribut *a* de C :

- le calcul des sur-classes directes de C par rapport à a, *sup(a,C)*,
- le calcul des sous-classes directes de C par rapport à a *inf(a,C)* et
- la mise à jour des sur-classes directes possibles PSD et des sous-classes directes possibles PsD de C

enfin, il garde, parmi les PSD, celles dont les attributs sont inclus dans la liste d'attributs de C ; et parmi les PsD celles dont la liste d'attributs contient les attributs de C.

Appariement

La comparaison entre deux classes est divisée ici en plusieurs comparaisons entre attributs. Deux attributs du même nom de deux classes différentes sont comparés en comparant la version normalisée de leurs types.

Les notions de description complète de classe, de normalisation de type d'attribut et de graphe de type d'attribut se trouve aussi dans notre modèle TROPES. Dans TROPES, le graphe des types d'un attribut n'est pas comme ici un graphe complet contenant explicitement toutes les relations de sous-typage de la base, mais un arbre plongé[1] dans le graphe de classes (§ 5.6.4).

---

[1] Le graphe A est plongé dans le graphe B si tous les éléments de A sont dans B et toute relation entre deux éléments dans A est présente dans B.

4.4.3. D’autres systèmes de classification de catégories

Missikoff et Scholl [MIS&88] font aussi la classification de types à partir du formalisme de types de Cardelli [CAR84]. Ils développent un algorithme pour insérer un nouveau type dans un treillis de types. Bien que l’algorithme de classification apporte des idées intéressantes, la mise à jours du treillis est couteuse et le graphe comporte des types artificiels qui ne correspondent pas à une structure particulière du monde représenté.

Il est important de noter que la structure de l’algorithme de Caponni et Chaillot [CAP&93] que nous venons de présenter favorise la classification interactive ; en effet, à chaque étape de la classification l’utilisateur peut voir les résultats partiels et modifier l’ordre d’ajout d’information en conséquence. L’utilisateur peut donner graduellement la description du nouveau concept, et il peut voir clairement les conformations successives des sur-classes et sous-classes directes possibles, observant ainsi l’impact de chaque attribut sur le résultat final de la classification de la classe, ce qui n’est pas en général le cas pour les algorithmes de classification des logiques terminologiques.

Finin [FIN86] propose aussi un algorithme interactif pour faire de la classification graduelle de classes dans un arbre de classes. Le système part d’une description initiale qui peut être incomplète. Si, avec l’information initiale, l’algorithme n’arrive pas à classer l’instance, il détermine un ensemble de questions pertinentes (dont la réponse aiderait à la descente de l’objet dans le graphe) pour les poser à l’utilisateur. Finin propose deux stratégies d’accélération pour la classification : *classification par profil d’attribut* et *classification par exclusion*. La première stratégie rejoint les idées de l’algorithme précédent de choisir, à partir des relations de sous-typages entre les types des attributs, les prédécesseurs possibles de la classe. La deuxième stratégie tire parti de la structure d’arbre pour choisir, dans chaque niveau, une seule classe comme le prédécesseur de la classe à classer. L’algorithme utilise un langage de concepts et il fait une pré-classification des concepts qui sont présents dans la description des rôles mais qui ne sont pas encore localisés explicitement dans la base.

Tous les algorithmes présentés précédemment visent à trouver la place correcte pour une catégorie dans un graphe de catégories existantes induit par une relation d’ordre. Par la suite, nous allons discuter des algorithmes qui classent, dans ce même graphe de catégories, un individu particulier. Le problème consiste, dans ce cas, à trouver la(s) catégorie(s) (ou classe) d’appartenance la(es) plus spécialisée(s) pour l’individu à classer.

4.5. Classification d’instances

La classification d’une instance consiste à trouver les catégories les plus spécialisées auxquelles l’instance appartient. Ce mécanisme part d’une instance dont on a une connaissance totale ou partielle et d’un graphe de catégories. Une catégorie est décrite par son intension ou structure et elle représente un ensemble potentiel d’instances (celles qui satisfont la structure de la catégorie) ; le graphe de catégories est induit par une relation d’ordre qui doit être cohérente avec la relation d’inclusion ensembliste existant entre les différentes catégories.

Par la suite nous allons décrire deux systèmes qui font de la classification d’instances : SHIRKA et LOOM. SHIRKA est un système de représentation par objets ayant l’approche classe-instance ; son modèle et son mécanisme de classification d’instances ont été utilisés par des applications des domaines aussi divers que la biométrie [ROU88], la prévention d’avalanches [BUI90], l’électromyographie [BÖH91], etc. LOOM

[McGRE91] est une logique terminologique qui offre un mécanisme de classification d'instances original, indépendant du mécanisme de classification de classes.

4.5.1. Classification dans SHIRKA

Représentation

SHIRKA est un système de représentation de connaissances par objets ayant l'approche classe-instance (§ 3). Ses éléments de base sont les classes et les instances : une classe est décrite par son intension (liste d'attributs avec leurs facettes) et une instance par une liste d'attributs avec leurs valeurs. Toute instance a une classe d'instanciation, qui est la classe à laquelle elle est attachée lors de sa création ; les valeurs des attributs de l'instance satisfont les contraintes de cette classe.

SHIRKA permet la manipulation d'instances incomplètes. La relation d'appartenance d'une instance à une classe est alors basée sur une logique à trois valeurs : sûre, possible et impossible (§ 4.3.1).

Les classes sont organisées dans un graphe de spécialisation (§ 2.2.5) ayant comme racine la classe *Objet* qui représente l'ensemble universel du monde à représenter. Les successeurs directs de la classe *Objet* établissent une partition du monde dans des familles disjointes décrivant des groupes d'individus aussi différents que *Personne* et *Boisson*. Un attribut est supposé avoir la même signification chaque fois qu'il est utilisé à l'intérieur d'une même famille. La classification d'instances dans SHIRKA se déroule à l'intérieur d'une seule de ces familles.

Ainsi, par exemple, une base de connaissances sur les boissons (Fig. 4.9) est décrite en SHIRKA par

```
{ boisson
     sorte-de = objet ;
     nom      $un chaine }

{ avec-alcool
     sorte-de      = boisson ;
     pourcalcool   $un entier
                   $intervalle [1 100] }

{ a-base-de-fruits
     sorte-de = boisson ;
     base     $liste-de cha ne
              $domaine  fraise   ananas   citron   orange  }

{ cocktail-a-base-de-fruits
     sorte-de = avec-alcool a-base-de-fruits
     chaude   $un bool en
              $valeur faux}

{ bi res
     sorte-de = avec-alcool ;
     nom $domaine  Adelscott   Heineken  }

{ jus
     sorte-de      = a-base-de-fruits ;
     pourcalcool   $un entier
                   $valeur 0 }
....
```

```
{  Adelscott
     est-un         =  avec_alcool;
     nom       =   Adelscott
     pourcalcool    = 4 }
```

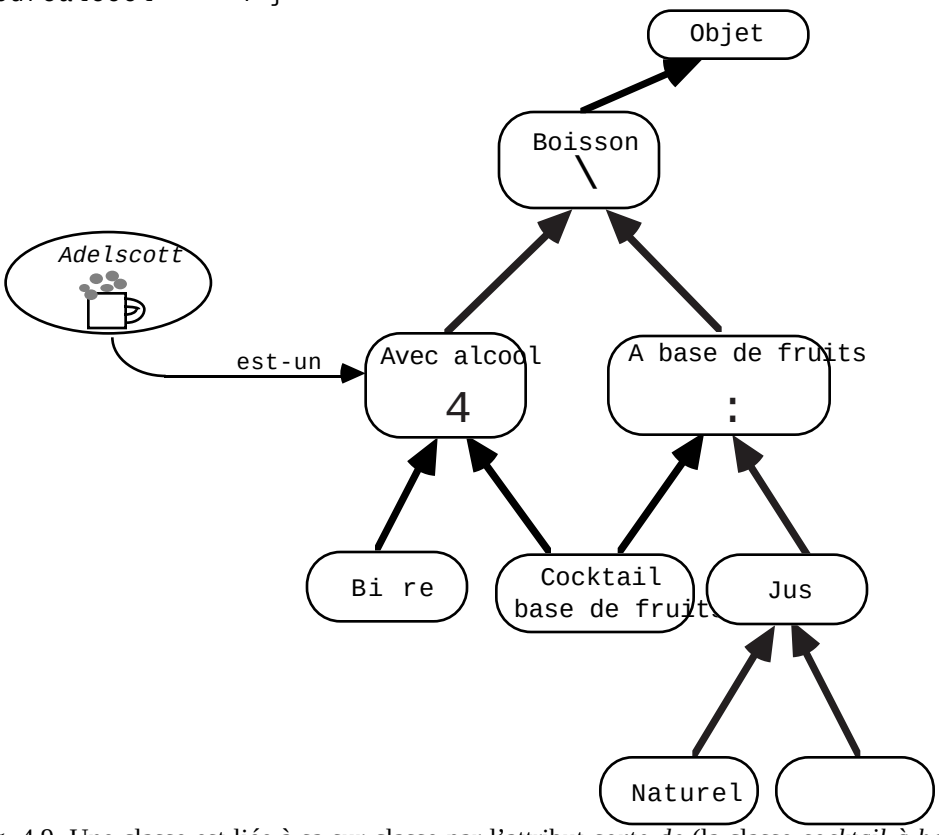


Fig. 4.9. Une classe est liée à sa sur-classe par l'attribut *sorte-de* (la classe *cocktail à base de fruits* est liée à deux sur-classes : *avec alcool* et *à base de fruits*) ; une instance est liée à sa classe d'appartenance par l'attribut *est-un* (ex : *Adelscott* est liée à sa classe *avec alcool*)

Parcours du graphe

L'espace de recherche pour la classification d'une instance est la famille à laquelle appartient l'instance. La procédure de classification commence avec l'instance à classer déjà rattachée à une ou plusieurs classes de cette famille (dans le pire des cas l'instance est rattachée à la classe la plus générale de la famille, le chef de la famille) [PIV&87]. Dans l'exemple précédent (Fig. 4.9) le chef de la famille pour la classification de l'instance *Adelscott* est la classe *Boisson*, mais l'instance est déjà rattachée à la classe *avec_alcool.*

L'algorithme de classification de SHIRKA fait un parcours en profondeur du graphe de classes de la famille. Ce parcours prend en compte un ensemble de propriétés qui découlent de la sémantique de la relation de spécialisation et des marques sûre, possible et impossible, pour réduire le graphe de recherche et pour simplifier les comparaisons :

1° Si une classe est sûre pour une instance, toutes ses sur-classes doivent aussi être sûres. Dans l'exemple les classes *Boisson* et *Objet*, sur-classes de *avec-alcool* , sont sûres.

2° Si une classe est impossible pour une instance, toutes ses sous-classes deviennent aussi impossibles. Dans l'exemple, comme la classe *jus* est impossible, ses sous-classes *Naturel* et *Artificiel* deviennent aussi impossibles

3° Les sous-classes d'une classe possible pour une instance ne peuvent pas être sûres pour cette instance (seulement possibles ou impossibles).

La classification d'une instance dans SHIRKA donne comme résultat final trois listes : la liste des classes sûres, la liste des classes possibles et la liste des classes

impossibles. Dans l'exemple, les classes sûres pour l'instance sont {Objet, Boisson, Avec_alcool, Bière}, les classes possibles : {Cocktail, Avec_des_fruits } et les classes impossibles {Jus, Naturel, Artificiel } (Fig. 4.10).

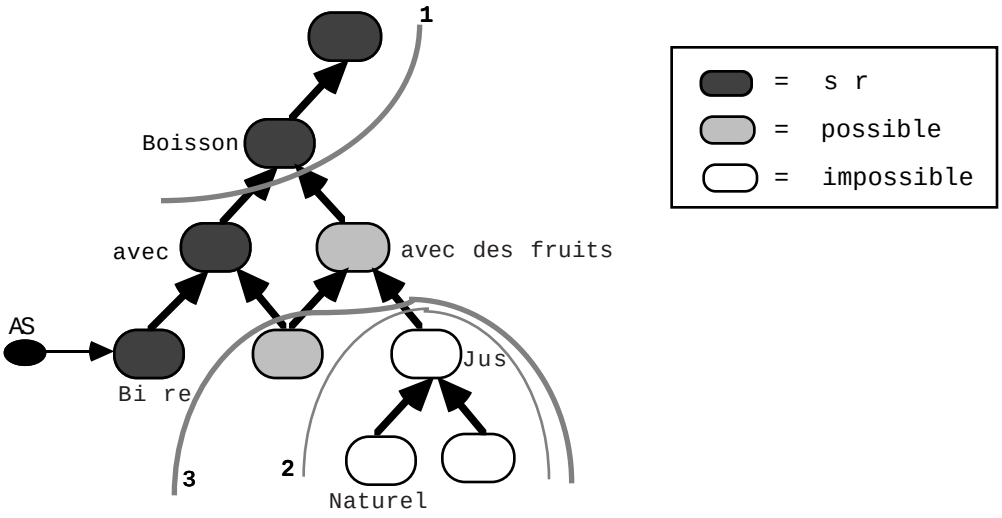


Fig. 4.10. L'application des trois propriétés précédentes (1,2,3) permet de réduire le nombre de comparaisons entre l'instance et les classes. Les marques des classes *Avec alcool, Boisson, Objet, Naturel,* et *Artificiel* ont été déduites grâce à ces propriétés).

Appariement

La comparaison entre une instance et une classe peut donner trois résultats différents : **sûre** si l'instance appartient à la classe, **impossible** si elle est en contradiction avec la classe et **possible** si a priori elle n'est pas en contradiction avec la classe, mais qu'il manque des informations pour être sûr de son appartenance.

Une classe C est une classe **sûre** de I (I appartient à C) si pour chaque attribut de C (défini dans la classe même ou hérité) la valeur de cet attribut dans I satisfait les contraintes de C (d'intervalle, domaine, etc.). Donc, cette appartenance peut être déterminée seulement si I est complète et satisfait les contraintes de C, ou si à partir des contraintes de la classe d'appartenance initiale de I on peut garantir la satisfaction des contraintes de C[1]. Dans l'exemple (Fig. 4.9) l'instance *Adelscott,* membre de la classe *avec_alcool* est une *Bière* parce qu'elle satisfait toutes les contraintes héritées par *Bière* et en plus son nom satisfait la contrainte de domaine ajoutée dans *Bière*. La classe *Bière* est donc une classe **sûre** pour l'instance (Fig. 4.11)

Fig. 4.11. L'instance *Adelcott* satisfait la contrainte de l'attribut *nom* de la classe *Bière*

Lorsque la valeur d'un attribut de l'instance ne satisfait pas les contraintes établies pour cet attribut dans la classe, le système donne la réponse **impossible**, car on sait que l'instance ne peut pas appartenir à cette classe, même si elle n'est pas complète. Dans

[1]SHIRKA classifie les instances déjà incluses dans la base, c'est-à-dire ayant déjà une classe d'appartenance initiale.

l'exemple précédent, *Adelscott* ne peut pas être un jus parce que son pourcentage d'alcool est 4% et les jus n'ont pas d'alcool ; la classe *jus* est une classe **impossible** pour l'instance (Fig. 4.12)

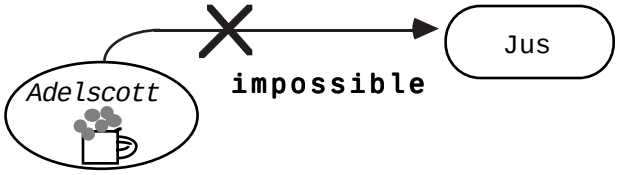


Fig. 4.12. *Adelscott* ne satisfait pas la contrainte de l'attribut *pourcalcool* de la classe *Jus*

Si une instance n'a pas de valeur pour tous les attributs définis dans une classe et si les attributs valués de l'instance ne sont pas en contradiction avec la classe, on dit que cette classe est une classe **possible** pour l'instance (son appartenance ne peut pas être déterminée pour l'instant car il manque des informations). Ainsi, *Adelscott* peut être un *cocktail à base de fruits* car elle satisfait la condition d'avoir au moins 1% d'alcool et comme elle n'a pas d'attributs *base* ni *chaude*, ces deux attributs ne sont, a priori, pas en contradiction avec la classe. La classe *Cocktail à base de fruits* est alors une classe **possible** pour *Adelscott* (Fig. 4.13).

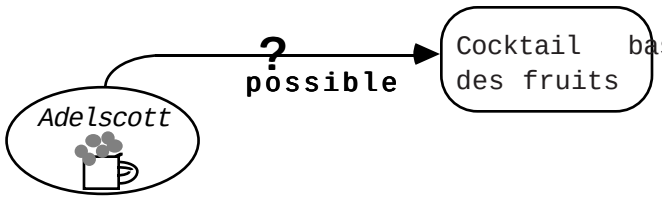


Fig. 4.13. *Adelscott* n'a pas de valeur pour l'attribut *chaude* de *Cocktail_a_base_de_fruits*

La classification d'instances dans SHIRKA est un processus interactif et itératif. L'information initiale de l'instance étant incomplète, le système peut avoir besoin d'informations manquantes lors de la descente de l'instance dans le graphe. Lorsque le système a besoin d'une connaissance manquante pour continuer la descente de l'instance, il essaye de l'inférer à partir des connaissances de la base ; s'il n'y arrive pas, il la demande à l'utilisateur.

SHIRKA offre une option de spécialisation d'instances, cas particulier de la classification où l'utilisateur indique le nom d'une classe et d'une instance existant dans la base et le système retourne la mention de cette classe pour cette instance (sûr, possible ou impossible).

4.5.2. Classification d'instances dans les logiques terminologiques

Dans les logiques terminologiques, la connaissance d'un individu est décrite à deux niveaux. D'une part il y a une connaissance définitionnelle de l'individu qui est décrite, de même que pour les concepts génériques, par un concept de la base, construit à partir des opérateurs du langage terminologique (TBox) (§ 1.3.5); ce concept est appelé un **concept individuel**. D'autre part, l'instance est utilisée dans des phrases du langage assertionnel (ABox) qui décrivent des faits. Ces axiomes concernent aussi bien l'appartenance explicite de l'instance à un concept ou un ensemble de concepts {Ci} que certaines caractéristiques (restrictions de rôles) connues {Fi} et ses relations avec d'autres instances (les valeurs de certains de ses rôles).

Une instance appartient à un concept (elle est un membre de l'ensemble décrit par le concept) si elle satisfait la description du concept, c'est-à-dire si elle satisfait chacune des caractéristiques de la description du concept (§ 4.3.1). Ainsi toute instance d'un concept doit être instance des concepts primaires du concept, doit posséder les rôles décrits dans le concept et les "fillers" (valeurs) de ces rôles doivent satisfaire les caractéristiques de ces rôles dans le concept.

La classification d'instances (individus) dans les logiques terminologiques consiste à trouver les concepts les plus spécialisés du graphe de concepts dont l'instance est un membre. Ainsi, si T dénote l'ensemble d'axiomes terminologique (TBox) du langage et W les assertions du monde (Abox), le processus de classification d'une instance c, revient à trouver l'ensemble de concepts les plus spécialisés de la base, MSC(c), tels que l'appartenance de c à chacun de ces concepts est une conséquence valide de $T \cup W$.

Ainsi, si $A \in MSC(c)$ alors $T \cup W \models A(c)$ et

si $T \cup W \models B(c)$ alors il y a un A tq. $A \in MSC(c)$ et $T \cup W \models A \subseteq B$,

où $\subseteq$ est la relation de spécialisation de concepts [NEB&89], [NEB&91]. Lorsque l'ensemble d'instances intervenant dans la définition des termes est disjoint de l'ensemble d'instances intervenant dans les assertions (ce qui est souvent le cas), la relation de spécialisation de concepts correspond à la relation d'ordre donnée par le graphe de subsomption de concepts $\leq$.

La plupart des logiques terminologiques utilisent, pour la classification d'instances, l'algorithme de classification de concepts, le classifieur. À partir des assertions connues sur l'instance (concepts d'appartenance {Ci} et rôles et caractéristiques {Fi}), ces systèmes construisent une abstraction de l'instance sous la forme d'une description. Cette description devient un concept "individuel" qui est classé dans le graphe par l'algorithme de classification de concepts. Les deux problèmes principaux de cette approche sont le grand nombre de concepts individuels qui sont ajoutés à la base et l'utilisation de l'algorithme de classification de concepts au lieu d'un algorithme spécialisé et moins coûteux d'appariement entre une instance et un concept.

La logique terminologique LOOM [McGRE91b] est une des seules logiques terminologiques à utiliser un algorithme de classification spécifique pour classer des instances, le "**réaliseur"** ; cet algorithme fait une comparaison efficace entre l'instance et chaque concept de la base et il n'a pas besoin de stocker dans la base la description de chaque instance à classer. Par la suite nous allons décrire les différentes étapes de cet algorithme de classification d'instances.

Représentation

A part les concepts et les instances, le système LOOM permet la définition de règles [McGRE&91]. Ces règles servent à contrôler la cohérence de la base et à faire des inférences particulières au domaine ; par exemple, à partir des règles d'implication :

((>= age 21) => peut-être-élu) et (vétéran =>(>=age 60)) et de l'assertion (vétéran(fred)),

le système déduit (peut-être-élu(fred))

car il sait (age>=60) => (age>=21).

Enfin, pour la classification d'instances, LOOM utilise une hiérarchie de rôles et une hiérarchie pour chacun des types de caractéristiques de rôles. Dans LOOM un rôle peut être décrit par une des restrictions ou caractéristiques suivantes : *same-as, one-of, through, at-least, at-most, all* or *filled-by*. Ces caractéristiques restreignent la cardinalité ou domaine de valeurs possibles d'un rôle. Ainsi, par exemple la caractéristique *(at-least 1*

enfants) du concept *père* indique que tout instance *I* de *père* doit avoir au moins un enfant. À chacun de ces types de caractéristiques correspond une taxonomie[1]; dans chaque taxonomie les caractéristiques sont organisées par une relation d'ordre correspondante à l'inclusion ensembliste. Ainsi, par exemple dans la taxonomie des caractéristiques *at-least*, la caractéristique *at-least(1,enfants)* est un prédécesseur de (est plus générale que) la caractéristique *(at-least 2 enfants)* .

Parcours du graphe

L'instance à classer est représentée, à tout moment de la classification, par l'ensemble de concepts qu'elle satisfait {Ci}, l'ensemble de caractéristiques qu'elle satisfait {Fi} et les valeurs connues pour certains de ses rôles.

Normalisation de l'instance

Pour simplifier la comparaison entre l'instance et les différents concepts de la base, le système fait une normalisation de l'instance par l'algorithme suivant [McGRE91b] :

- calculer l'ensemble de caractéristiques héritées des concepts {Ci} et les combiner avec les caractéristiques de {Fi} pour produire le nouvel ensemble {Fu} de caractéristiques (probablement plus spécifiques).
- remplacer les caractéristiques filled-by par des valeurs des rôles correspondants
- classer les autres caractéristiques dans la taxinomie appropriée (same-as, one-of, through, at-least). Cette classification dépend de la sémantique de la caractéristique ; par exemple, *(at-least 2 fils)* est placé dans la taxinomie *at-least* en dessous de *(at-least 1 fils)* .
- marquer chacune des caractéristiques de {Fu} et ses prédécesseurs dans la taxonomie correspondante, .

La normalisation de l'instance I est faite une seule fois au début de la classification et c'est cette version normalisée qui va être classée.

Parcours du graphe

L'algorithme de classification fait un parcours du graphe de concepts en regardant, pour chaque concept, la relation d'appartenance de l'instance au concept. Le parcours du graphe correspond au parcours fait par la classification d'un concept pour trouver les subsumants les plus spécialisés SPS.

Appariement entre une instance et un concept

L'appariement entre une instance et un concept est un cas particulier de la subsomption : il s'agit ici de déterminer si une instance I satisfait la description d'un concept particulier D, où D est représenté par un ensemble de caractéristiques {Fd}. I satisfait la description D ssi elle satisfait chacune des caractéristiques de {Fd} ; soit F une de ces caractéristiques, I satisfait F si une des trois conditions suivante est vraie

- F est marquée,
- Les valeurs correspondant à F pour I satisfont les contraintes de F (par exemple, si F = (at-least 2 R), l'algorithme récupère les instances qui valident ce rôle pour i : les x pour lesquelles R(I,x) est vrai et il les compte pour voir s'il y en a au moins deux),
- I satisfait une autre caractéristique ou un concept qui implique F (par exemple par l'application d'une règle).

Si F est satisfaite par une des deux dernières conditions alors l'algorithme :

[1]sauf pour la caractéristique *filled-by* qui donne un ensemble explicite d'instances.

- marque F et tous ses prédécesseurs dans sa taxonomie,
- ajoute l'information de la satisfaction de F à la description de I,
- normalise la description de I avec cette nouvelle information,
- marque dans les arbres correspondants les nouvelles caractéristiques reconnues comme satisfaites par I après sa normalisation.

À la fin de la classification, l'instance a été enrichie par toutes les caractéristiques qu'elle satisfait et par la liste de concepts les plus spécialisés auxquels elle appartient. Les premières versions de LOOM attachaient (logiquement) l'instance à tous ces concepts ; la dernière version crée un concept nouveau à partir de la conjonction des concepts de l'instance et attache l'instance à ce concept. La raison de ce choix est de garder la description structurelle complète, obtenue, sous une forme "normalisée", pour pouvoir l'utiliser dans des classifications postérieures [McGRE&92].

Il est intéressant de noter la similitude entre les taxinomies de caractéristiques de LOOM et les graphes de types présentés dans (§ 4.4.2) : les deux utilisent ces structures auxiliaires pour inférer des relations entre la structure d'une instance et celle d'une catégorie à partir des relations entre les composants de ces structures. La différence entre les deux approches concerne la granularité des composants. En effet, dans les graphes de types chaque nœud représente le type normalisé (incluant l'information de tous les descripteurs) d'un attribut d'une (ou plusieurs) classe(s) ; dans les taxinomies de caractéristiques chaque nœud représente une caractéristique, c'est-à-dire un descripteur d'attribut. En fait, la richesse expressive des langages terminologiques rendrait impossible le regroupement de toutes les caractéristiques d'un rôle sous une forme normale unique (le type du rôle), alors que dans la classification par des types présentée précédemment, le langage est plus restreint et à une description corresponde une forme normale unique.

Conclusion

La classification est le mécanisme de raisonnement principal des représentations par objets. La classification est un processus qui, à partir d'une base de connaissances structurée et d'un nouvel objet, trouve la localisation appropriée de l'objet dans la base. Pour accomplir sa tâche, l'algorithme de classification doit parcourir la base de catégories et, pour chaque catégorie visitée, il doit faire une comparaison entre la catégorie et l'objet à classer pour déterminer la relation d'ordre (ou d'appartenance) entre eux.

Ainsi, le mécanisme de classification peut se décrire à partir de quatre aspects principaux ; d'une part, les éléments de la représentation et leur structuration dans la base, et d'autre part les étapes de l'algorithme concernant le parcours du graphe et l'appariement entre objets.

Les bases de connaissances d'une représentation par objets comportent des catégories d'objets et des individus particuliers. Une catégorie représente un ensemble d'individus et elle peut être décrite de façon fonctionnelle ou de façon structurelle. En termes de structure, une catégorie C est plus spécifique qu'une catégorie D ($C \leq D$), (D est prédécesseur de C) si la description de C affine celle de D, c'est-à-dire si elle ajoute des propriétés ou si elle restreint l'ensemble de valeurs possibles d'une propriété de D. La plupart des systèmes ne représentent explicitement que les liens d'une catégorie vers les prédécesseurs directs. Un individu appartient aux classes dont il satisfait les contraintes. En général, seuls les liens d'appartenance vers les classes les plus spécialisées sont représentés explicitement.

Dans une telle représentation, la classification d'une catégorie C consiste à trouver toutes les catégories de la base qui précèdent C (qui sont plus générales que C), puis toutes celles qui suivent C. Nous avons présenté deux types d'approches pour la classification de catégories. La première approche, illustrée par les logiques terminologiques, parcourt le graphe de catégories en faisant des comparaisons de la catégorie C avec chacune des catégories du graphe. La deuxième approche, utilise, pour chaque attribut de la base, un graphe des types (induit par la relation de sous-typage) comportant tous les types de ces attributs qui apparaissent dans les catégories de la base. L'algorithme de classification d'une catégorie C décompose la structure de la catégorie C en ses attributs et classe chacun de ces attributs dans le graphe de types correspondant, puis regroupe cette information pour obtenir les prédécesseurs et les successeurs de C.

Le parcours du graphe de catégories (ou dans le deuxième cas, graphes de types) pour trouver les prédécesseurs directs de la catégorie à classer, C, se fait, pour la plupart de systèmes, par niveaux, en partant des prédécesseurs connus de la catégorie C et en descendant vers leurs successeurs qui n'ont pas encore été examinés. La recherche des successeurs directs se poursuit à partir des prédécesseurs directs obtenus.

L'appariement entre catégories se fait par comparaison des contraintes des attributs. Pour les systèmes basés sur les types, cette comparaison, après normalisation des facettes des attributs, est une vérification simple d'inclusion ensembliste. Pour les logiques terminologiques, qui ont en général des facettes ou caractéristiques très élaborées, elle peut être très coûteuse à cause de la richesse expressive de la plupart de ces langages.

La classification d'instances, de même que la classification de catégories, comporte un parcours du graphe de catégories avec des comparaisons (appariements) successives entre l'instance et les catégories du graphe. Le parcours du graphe est fait avec un algorithme semblable à celui de la recherche des prédécesseurs directs d'une catégorie. L'appariement entre une instance et une catégorie consiste à vérifier le type des valeurs des attributs de l'instance. Cet appariement est moins coûteux que l'appariement entre catégories où il s'agit de comparer des ensembles de valeurs.

Les bases de connaissances manipulent, en général, des connaissances incomplètes. Un algorithme de classification d'instances doit permettre un enrichissement graduel de la connaissance de l'instance pendant la classification ; une telle classification graduelle doit être interactive pour pouvoir demander à l'utilisateur des connaissances manquantes et pour lui proposer des valeurs "intéressantes". Dans le chapitre 6 nous présentons un algorithme de classification d'instances. Cet algorithme, qui travaille sur une représentation multi-points de vue, rejoint les idées présentés par la classification de classes par des types ainsi que des stratégies d'optimisation du parcours du graphe proposées par LOOM et SHIRKA.

Bibliographie

[AGU89] AGUIRRE, J.L., *Construction automatique de taxonomies à partir d'exemples dans un modèle de connaissances par objets.* Thèse Informatique, INPG, Grenoble, juin 1989.

[AHO&74] AHO A., HOPCROFT J., ULLMAN J., *The design and Analysis of Computer Algorithms* Addison-Wesley, Reading, MA, 1974.

[BAA&92] BAADER F., HOLLUNDER B., NEBEL B., PROFITLICH H-J., 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, Cambridge, MA, pp.270-281, octobre 1992.

[BOB&77] BOBROW D.G., WINOGRAD T., *An overview of KRL, a Knowledge Representation Language.* Cognitive Science, vol. 1, n°. 1, pp.3-45, 1977.

[BOB&80] BOBROW D.G., GOLDSTEIN P., *Description for a Programming Environment*, in Proceedings of the AAAI, Stanford, 1980.

[BÖH91] BÖHM V., *Modélisation du raisonnement électromyographique à l'aide de la classification,* Rapport de stage DESS Intelligence Artificielle, Université de Savoie, Chambéry, juin 1991.

[BOR92] BORGIDA A., *From Type Systems to Knowledge Representations : Natural Semantics Specifications for Description Logics*, International Journal on Intelligent and Co-operative Information Systems, vol.1 n°.1, World Scientific Publ. Singapore, 1992.

[BOR92b] BORGIDA A., *Towards the Systematic Development of Description Logic Reasoners : CLASP reconstructed*, in Proceedings 3°. International Conference on Principles of Knowledge Representation and Reasoning, KR'92, Cambridge MA, pp.259-269, octobre 1992.

[BOR&89] BORGIDA A., BRACHMAN R.J., McGUINNESS D.L., RESNICK L.A., *CLASSIC : A Structural Data Model for Objects*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, Portland, Oregon, mai-juin, 1989.

[BOR&92] BORGIDA A., BRACHMAN R.J., *Customizable Classification Inference in the ProtoDL Description Management System,* in Proceedings of the Conference on Information and Knowledge Management, Baltimore, pp.1-9, novembre 1992.

[BUI90] BUISSON L., *Le raisonnement spatial dans les systèmes à base de connaissances. Application à l'analyse de sites avalancheux,* Thèse d'informatique, UJF, Grenoble, 1990.

[CAP&93] CAPPONI C., CHAILLOT M., *Construction incrémentale d'une base de classes correcte du point de vue des types,* Actes Journée Acquisition-Validation-Apprentissage, Saint-Raphael, 1993.

[CAR84] CARDELLI L., *A semantics of Multiple Inheritance*, LNCS, vol.137, Springer-Verlag, pp.51-67, 1984.

[CAR85] CARDELLI L., *Typechecking Dependent Types and Subtypes*, in Foundations of Logic and Functional Programming Workshop, LNCS, vol. 306, Springer-Verlag, pp.44-57, 1985.

[CLA85] CLANCEY W.J., *Heuristic Classification,* Artificial Intelligence Journal, vol. 27, n°. 4 , 1985.

[CAY&82] CAYROL M., FARRENY H., PRADE H., *Fuzzy Pattern Matching*, Kybernetics vol.11, pp.103-116, 1982.

[FIN86] FININ T. W., *Interactive Classification : A Technique for Acquiring and Maintening Knowledge Bases*, in Proceedings of the IEEE, vol. 74, n°. 10, pp.1414 - 1421, octobre 1986.

[GEN&89] GENNARI J.H., LANGLEY P., FISCHER D., *Models of Incremental Concept Formation.* . Artificial Intelligence, nº. 40, pp.11-61, 1989.

[GRA88] GRANGER C., *CLASSIC - générateur de systèmes experts en classification et en diagnostic*, Manuel de l'Utilisateur Ver 2.2, ILOG, 1988.

[McGRE91] MAC GREGOR R.M., *The Evolving Technology of Classification-Based Knowledge Representation Systems*, in Principles of Semantic Networks. Explorations in the Representation of Knowledge. J.Sowa (ed.), Morgan Kaufman Publishing, chapitre 13 pp.385-400, 1991.

[McGRE91b] MAC GREGOR R.M., *Inside the LOOM Description Classifier*, SIGART bulletin, vol. 2, n°. 3, Special Issues on Implemented Knowledge Representation and Reasonong Systems, juin, 1991.

[McGRE&91] MAC GREGOR R.M., BURSTEIN M.H. *Using a Description Classifier to Enhance Knowledge Representation*, IEEE Expert Intelligent Systems and Applications, juin, 1991.

[McGRE&92] MAC GREGOR R.M., BRILL D., *Recognition Algorithms for the LOOM Classifier*, AAAI, San José, CA, juillet, pp.774-779, 1992.

[MAR89] MARIÑO O., *Classification d'objets dans un modèle multi-points-de-vue*, Rapport DEA d'informatique INPG, Grenoble, 1989.

[MIC&86] MICHALSKI R.S., STEPP R.E., *Conceptual Clustering : Inventing Goal-Oriented Classifications of Structured Objects*, in Michalsky, Carbonell et Mitchell (eds.), Machine Learning, an Artificial Intelligence Approach vol.2., Morgan Kaufmann Publishers, Los Altos, CA., pp.471-198, 1986.

[MIS&89] MISSIKOFF M., SCHOLL M., *An Algorithm for Insertion into a Lattice : Application to Type Classification*, Foundations of Data Organization and Algorithms, LNCS 367, W. Lirwin, M.-J. Schek (eds.), pp.64-82, 1989..

[NAP92] NAPOLI A., *Représentations à objets et raisonnement par classification en intelligence artificielle*, Thèse d'état, Université de Nancy 1, 1992.

[NEB&89] NEBEL B., SMOLKA G., *Representation and Reasoning with Attributive Descriptions*, IWBS Report 81, Stuttgart, Allemagne, 1989.

[NEB90] NEBEL B., *Reasoning and Revision in Hybrid Representation Systems*, Lecture Notes in Artificial Intelligence, LNCS, vol. 422, Springer-Verlag, Berlin, 1990.

[NEB&91] NEBEL B., SCHMOLKA G., *Attribut Description Formalism … and the rest of the world*, in Text O:Herzog, C.-R. Rollinger (eds.) Understanding in LILOG, LNCS 546, pp.439-452, Springr-Verlqg, Berlin, 1991.

[PAT&91] PATEL-SCHNEIDER P., McGUINESS D., BRACHMAN R., RESNICK L., BORGIDA A., *The CLASSIC Knowledge Representation System : Guiding Principles and Implementation Rational*, SIGART Bulletin, vol.2, n°. 3, pp.108-113, 1991.

[PIV&87] PIVOT B., PROKOP M., *Définition et réalisation d'une fonctionalité de classification dans SHIRKA.*, Année Spéciale Intelligence Artificielle, INPG, 1987.

[REC88] RECHENMANN F., *SHIRKA : système de gestion de bases de connaissances centrées-objet*, Manuel d'utilisation 1988.

[ROS90] ROSSAZZA Jean-Paul,*Utilisation de hiérarchies de classes floues pour la représentation de connaissances imprecises et sujettes à exceptions : le système SORCIER*, Thèse d'Informatique, Université Paul Sabatier de Toulouse, 1990.

[ROU88] ROUSSEAU B., *Vers un environnement de résolution de problèmes en biométrie : Apport des techniques de l'intelligence artificielle et de l'interaction graphique*, Thèse de doctorat, Université Claude Bernard Lyon 1, Lyon, 1988.

[SCH&83] SCHMOLZE J.G., LIPKIS T.A, *Classification in the KL-ONE Knowledge Representation System*, in Proceedings of the 8th. IJCAI, Karlsruhe, Germany, 1983.

[VOG88] VOGEL C., Génie Cognitif, Collection Sciences cognitives, MASSON, PAris, pp.97 1988.

[WEG87] WEGNER,P.,*The Object-Oriented Classification Paradigm*, dans Research Directions in Object-Oriented Programming, Bruce Shiver, Peter Wegner (éd.), The MIT Press, Cambridge, MA, 1987.

[WOO91] WOODS W.A., *Understanding Subsumption and Taxonomy : a Framework for Progress*, in Principles of Semantic Networks. Explorations in the Representation of Knowledge. J.Sowa (éd.), Morgan Kaufman Publishing, chapitre 1, pp.45-94, 1991.

[ZHA91] ZHANG J. *Discussion sur la construction de catégories*, mail groupe comp.ai, 1991.

Chapitre 5

Le modèle TROPES

Chapitre 5

Le modèle TROPES

Introduction

Les systèmes de représentation de connaissances comportent, d'une part, un modèle déclaratif de représentation de la connaissance et d'autre part, les mécanismes d'inférence qui travaillent sur ce modèle. Dans ce chapitre, nous présentons le modèle de représentation multi-points de vue TROPES, qui est le modèle de base de notre travail. À partir du chapitre 6, nous décrivons les mécanismes d'inférence par classification d'instances que nous avons développés sur ce modèle.

TROPES est un modèle de représentation de connaissances à objets qui permet la représentation d'objets complexes, incomplets et évolutifs. TROPES sépare la connaissance du monde dans des familles ou concepts disjoints. Un concept peut être vu et manipulé selon des points de vue différents ; chaque point de vue donne lieu à une structuration particulière de la connaissance du concept dans un graphe de classes. L'aspect pluridisciplinaire d'une base de connaissances se reflète dans l'ensemble de points de vue. Les points de vue d'un concept peuvent être connectés par des passerelles ; une passerelle établit une relation d'inclusion ensembliste entre des classes de points de vue différents. Les passerelles reflètent l'aspect interdisciplinaire d'une base de connaissances et s'avèrent très utiles pour les mécanismes d'inférence par classification. Dans TROPES un individu du monde est représenté par une instance d'un concept, qui est rattachée à sa classe la plus spécialisée dans chacun des points de vue du concept. Enfin, TROPES permet la manipulation d'objets composites. Un objet composite peut être décomposé de diverses façons dans les différents points de vue de son concept.

Dans ce chapitre, nous décrivons les éléments de représentation du modèle TROPES. La première section présente les deux dimensions d'une base de connaissances TROPES : les concepts et les points de vue. La section deux présente plus en détail la définition d'un concept. Les sections trois et quatre sont dédiées à l'étude des points de vue et des passerelles d'un concept. Dans la section cinq, nous décrivons la structure et la sémantique des classes d'un point de vue et dans la section six leurs attributs et descripteurs. La section sept présente la description des instances et leurs relations avec les autres éléments de la base. Enfin, la section huit montre l'extension du modèle aux objets composites multi-points de vue.

5.1. Une base de connaissances TROPES

La base de connaissances est la structure du modèle contenant la connaissance déclarative du monde représenté. Une base de connaissances TROPES est composée de plusieurs familles d'objets appelées concepts. La connaissance d'une telle base (et ses concepts) peut être considérée selon différents points de vue ou perspectives.

5.1.1. Les concepts d'une base de connaissances

Une base de connaissances assez complexe manipule des familles
nature différente telles que *Personne* et *Appartement.* Dans TROPES, une tel
d'objets s'appelle un **concept**. Une **base** de connaissances TROPES est co
ou plusieurs **concepts** indépendants : ces concepts sont disjoints deu
déterminent une partition de l'univers du discours, c'est-à-dire que toute i
base de connaissances appartient à un et un seul concept de la base. Par exemp
base de connaissances immobilière contient les concepts *Locataire*[1], *Apparte*
Agence (Fig. 5.1).

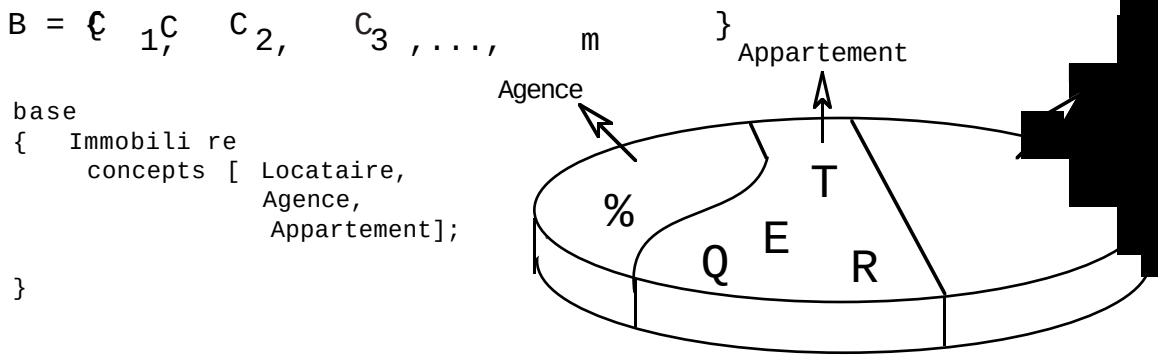


Fig. 5.1. Une base de connaissances est composée de concepts disjoints. Par exemple, la base *Immobilière* comporte les concepts indépendants : *Locataire*, *Appartement* et *Agence immobilière*.

Souvent les modèles de représentation à objets qui permettent la migration d'instances, ne font pas cette partition en concepts : tous les objets de leurs bases de connaissances font partie d'un même graphe de classes dont la racine est la classe *Objet*. Cette représentation rend difficile la définition de la sémantique associée à la base de connaissances. D'une part, la disjonction des concepts n'est pas garantie, car rien n'empêche la création d'instances ou de classes issues de plus d'un concept, comme la classe Appartement-Locataire, qui n'ont aucun modèle dans le monde réel. D'autre part, toute instance de la base appartient à la classe racine *Objet* , donc elle peut naviguer dans toute la base, migrer d'un concept à un autre, ce qui alourdit les mécanismes de raisonnement et leur enlève une partie de leur sens.

Par ailleurs, il est important de remarquer que, bien que les concepts TROPES soient disjoints deux à deux, et qu'ils décrivent ainsi des ensembles distincts d'individus, ils ne sont pas complètement indépendants (Fig. 5.2).

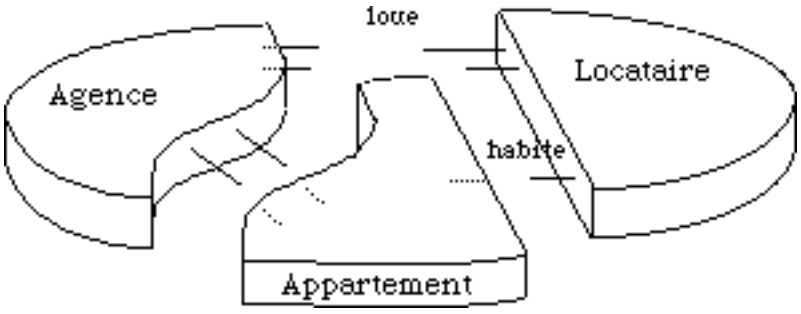


Fig. 5.2. L'attribut *habite* du concept *Locataire* prend comme valeur une instance du concept *Appartement* ; l'attribut *loue* du concept *Agence* est défini sur les instances du concept *Locataire.*

[1] Le nom d'un concept est unique dans la base de connaissances.

En effet, si on les a définis dans une même base de connaissances, c'est parce que dans le monde que l'on veut modéliser ces différentes familles interagissent. Dans TROPES cette interaction est décrite au niveau des attributs des concepts. On peut avoir, par exemple dans le concept *Locataire* l'attribut *habite* qui prend ses valeurs dans le concept *Appartement.* La notion d'attribut d'un concept est présentée dans § 5.2.1.

5.1.2. Les points de vue d'une base de connaissances

Une base de connaissances représente et structure la connaissance du monde réel. On a vu auparavant que cette connaissance ressort de domaines différents, des familles différentes d'objets. Si la connaissance du monde est variée, il en est de même des observateurs qui veulent la manipuler : chacun a ses intérêts particuliers ; chacun regarde des propriétés et relations particulières des objets du monde, et chacun organise les objets selon sa propre perspective. Dans TROPES, cette diversité est reflétée dans la base de connaissances par les points de vue : un point de vue de la base décrit une perspective particulière d'observation du monde représenté. Les points de vue dans TROPES suivent l'hypothèse d'un monde unique vu de façons complémentaires par les divers observateurs (§ 1.2.3 l'agent et le monde à représenter, option 2). Une base de connaissances comporte alors deux types d'éléments : les concepts et les points de vue (Fig. 5.3.).

$B = \{C_1, C_2, C_3, \ldots, C_m\} + \{PV_1 + PV_2, + \ldots + PV_n\}$

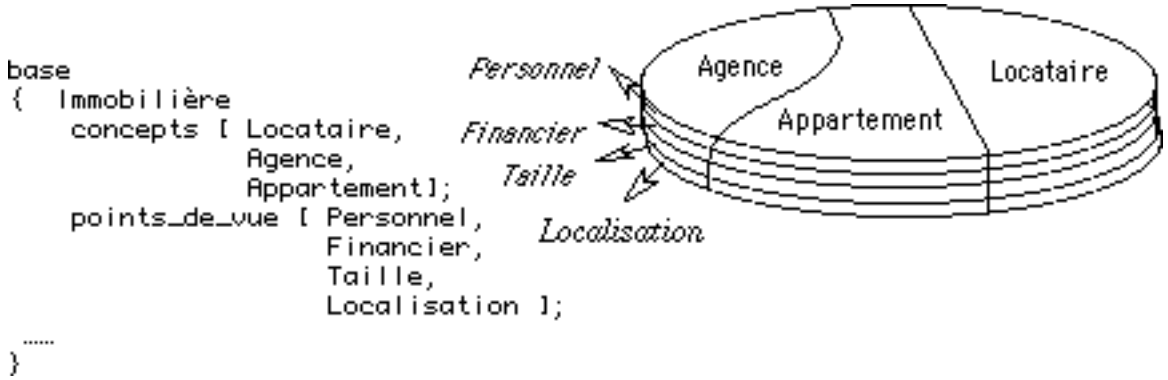


Fig. 5.3. La base *Immobilière* comporte quatre points de vue : *Personnel, Financier Taille* et *Localisation.* Les points de vue sont orthogonaux aux concepts.

Les points de vue sont définis au niveau de toute la base et leurs noms sont visibles (connus) pour tous les concepts de la base[1]. Or, certains points de vue, qui sont pertinents pour un concept, peuvent ne pas l'être pour un autre.

C'est le cas du point de vue *Personnel* dans l'exemple de la base *Immobilière*, qui permet de regarder les caractéristiques personnelles du *Locataire* mais qui n'a pas de sens pour les autres concepts de la base ; par contre, l'observateur expert en finances (point de vue *Financier*) regarde aussi bien les finances du locataire que les conditions financières des appartements et celles des agents qui les louent (Fig. 5.4). Alors; lors de la définition d'un concept de la base, le concepteur doit indiquer les points de vue pertinents pour ce concept (§ 5.3).

[1] Le nom d'un point de vue est unique dans la base de connaissances. Ce choix permet la manipulation d'objets composites (formés d'objets de différents concepts) selon un point de vue particulier (§ 5.8).

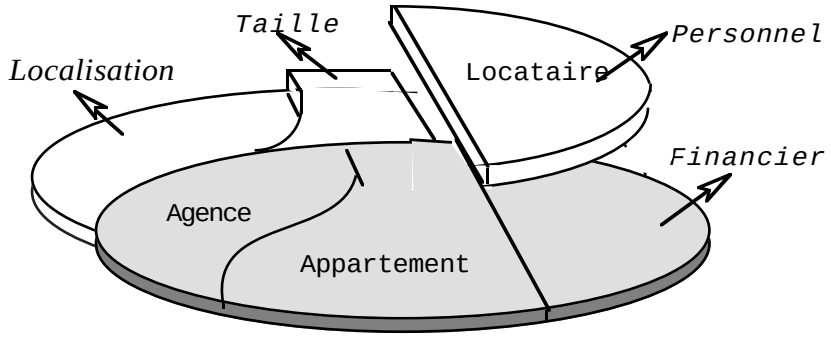


Fig. 5.4. L'ensemble des points de vue de la base est l'union des points de vue des concepts de la base. Dans la base *Immobilière*, le concept *Appartement* est visible des points de vue *Financier*, *Taille* et *Localisation*, tandis que pour *Agence* seuls *Financier* et *Localisation* sont pertinents et le concept *Locataire* est décrit des points de vue *Personnel et Financier*.

5.2. Les concepts de la base

Un concept de la base de connaissances décrit une famille d'individus semblables ; ces individus, appelés instances sont décrits en termes d'un ensemble d'attributs, les attributs du concept, et ils sont structurés dans différentes structures de classes correspondant aux différents points de vue du concept.

Si la base est l'espace des noms des concepts et des points de vue, le concept devient l'espace de noms des éléments de plus bas niveau du modèle : les attributs, les classes et les instances. La portée du nom d'un attribut est le concept. Cela signifie, d'une part que deux occurrences d'un même nom d'attribut à l'intérieur d'un concept font référence au même attribut ; ainsi, par exemple si l'on utilise dans le concept *Appartement* de la base *Immobilière* un attribut *age* pour représenter l'age du bâtiment dans lequel se trouve l'appartement, toute occurrence d'*age* dans le concept *Appartement* fait référence à cet attribut[1]. D'autre part, deux concepts différents peuvent utiliser un même nom d'attribut et le système les traite comme deux attributs différents. Par exemple, dans le concept *Locataire* un attribut *age* peut représenter l'age de la personne qui loue un appartement, qui n'est pas confondu avec l'age d'un bâtiment ; cette réduction de la portée des noms au concept permet la modification d'un concept de la base de façon relativement indépendante des autres concepts.

Le concept est aussi l'espace de recherche des mécanismes d'inférence : en effet, les mécanismes de classification, filtrage, analogie, etc., ne sont vraiment utiles que lorsque l'on a restreint l'espace de recherche à l'ensemble d'objets concernés par ce raisonnement. Ainsi, par exemple, avant de classer un nouvel objet dans la base de connaissances immobilières, il faut, au moins, savoir s'il s'agit d'un appartement, d'une personne ou d'une agence.

5.2.1. Les attributs d'un concept

Les instances d'un concept sont décrites par un ensemble d'attributs. Ces attributs sont **définis** au niveau du concept. La définition d'un attribut d'un concept comporte, d'une part, l'information générale liée à l'attribut qui est valide pour toutes ses occurrences dans le concept et d'autre part, le plus grand domaine de valeurs possibles pour cet attribut, le

[1]Ce choix élimine le conflit de nom d'attributs [DUC&92] lors d'une spécialisation multiple (§2.4.2).

domaine maximal. Ce domaine correspond au type maximal de l'attribut ; la description de cet attribut dans une classe détermine un sous-ensemble du domaine maximal et donc un sous-type du type maximal.

L'information générale de l'attribut inclut sa nature, propriété, composant ou relation[1], son caractère, mono ou multi-valué[2] et les tâches ou schémas de méthodes définis dans la base pour valuer cet attribut[3]. Cette information générale est valide et figée pour toutes les instances du concept. Ainsi, par exemple, si l'on définit l'attribut *pièce* du concept *Appartement* comme étant multi-valué et de nature composant, alors pour toute instance d'*Appartement* l'attribut *pièce* est un attribut-composant multi-valué.

Le domaine maximal des valeurs d'un attribut est l'ensemble maximal de valeurs que peut prendre cet attribut pour une instance du concept. Cet ensemble maximal peut être un domaine **primitif** si l'attribut prend ses valeurs dans un type de base du système tels que entier, chaîne, etc,… ou bien un domaine **défini**, s'il prend ses valeurs dans un autre concept défini de la base. Un concept ayant des attributs qui prennent des valeurs dans d'autres concepts de la base est appellé concept composé et ses instances, instances composées[4].

Par exemple, le concept *Appartement* peut avoir les attributs : *adresse, surface, étage, caution, loyer, nb_pièces,* etc ; L'attribut *nb_pièces* peut être défini comme étant à caractère mono-valué et de nature propriété et prenant ses valeurs dans le domaine primitif entier (Fig. 5.5)

```
attribut
{  nb_pi ces
   concept  appartement  ;
   valeurs_dans  entier  ;
   tache  NULL  ;
   nature   propri t    ;
   multi-valeur   faux   ;
}
```

Fig. 5.5. Un attribut d'un concept est défini par un schéma qui indique le concept dans lequel il prend ses valeurs ainsi que sa nature et caractère.

Chaque description d'attribut dans TROPES a un type associé ; le **type d'un attribut** est déterminé par le domaine de valeurs qu'il peut prendre (§ 4.4.2). Le **type maximal** d'un attribut est le type donné lors de la définition de cet attribut dans le concept, car c'est cette définition qui contient le domaine maximal de valeurs que peut prendre l'attribut : si l'attribut est mono-valué, ce domaine est le domaine maximal de base, si l'attribut est multi-valué son domaine maximal est établi par toutes les combinaisons des valeurs du domaine maximal de base. Le domaine maximal de valeurs d'un attribut d'un concept peut être restreint dans une classe particulière du concept par l'ajout des descripteurs de contraintes (§ 2.2.3, § 5.5.2) ; ces restrictions donnent lieu aux sous-types du type maximal. La structure des types d'un attribut induit par la relation de sous-typage est présentée dans § 5.6.4.

---

[1]Les attributs de nature relation sont traités actuellement comme les attributs de nature propriété.

[2]Par rapport à la distinction faite dans §2.2.2.2., la version actuelle de TROPES traite les propriétés multi-valuées comme des ensembles qui représentent une unité indivisible et il traite les composants multiples comme des listes ordonnées des valeurs simples. La justification de ce choix est donnée dans §5.8.3.

[3] Voir [ORS90] pour une discussion sur les tâches et les méthodes dans TROPES.

[4]À ne pas confondre avec les concepts et les instances composites (ayant des attributs de nature composant). Des caractéristiques particulières de la classification d'objets composées sont décrites dans § 6.4.

5.2.2. La clé d'un concept

Les systèmes à base de connaissances doivent pouvoir manipuler des instances ayant des informations manquantes, des instances incomplets. TROPES offre cette possibilité. Cependant une information minimale est nécessaire pour pouvoir raisonner avec l'instance, pour pouvoir même l'introduire dans la base. Cette information minimale doit permettre au système de distinguer l'instance de toutes les autres instances de son concept.

Chaque concept TROPES possède un ensemble d'**attributs clés** dont l'évaluation permet d'identifier de façon unique une instance du concept ; cet ensemble minimal d'attributs doit être valué dans toute instance d'un concept pour pouvoir dire qu'elle existe dans le concept. Ainsi, l'information minimale associée à une instance est le nom de son concept et les valeurs des attributs clés de ce concept.

Pour la base immobilière de l'exemple, le concept *Locataire* peut avoir comme attributs clés le *nom* complet de la personne et la *date_de_naissance* ; un appartement est identifié de façon unique par l'ensemble d'attributs clés *adresse*, *étage* et *numéro* (Fig. 5.6) ; enfin une agence immobilière peut s'identifier par son *nom* et le nom de la *succursale.*

```
concept
{  Appartement
   clefs  [adresse,   tage,  num ro]  ;
   points_de_vue  [  Financier,  Taille,  Localisation  ]
}
```

Fig. 5.6. La description d'un concept comporte son nom, la liste des attributs qui forment la clé et la liste des points de vue du concept.

Un concept c de la base **B** est complètement défini par la définition de ses attributs : a_{1c}, ...,a_{mc}, la liste des attributs clés et ses points de vue (les points de vue de la base qui sont pertinents pour ce concept) : PV_{1c}, ..., PV_{xc}. L'ensemble des points de vue d'un concept donne une vision complète du concept :

$c = [PV_{1c}, PV_{2c},..., PV_{xc}] \cup [a_{1c}, a_{2c},...,a_{mc}] \cup [aclé_{1c}, aclé_{2c},...,aclé_{nc}]$

c dans **B**, $\forall_i$, PV_{ic} dans $c \Rightarrow PV_{ic}$ dans **B** , $\forall_j$ $aclé_{jc}$ dans $[a_{1c}, a_{2c},...,a_{mc}]$

5.3. Les points de vue d'un concept

La notion de point de vue prend tout son sens à l'intérieur du concept ; la famille d'objets décrite par un concept particulier peut être regardée selon différents points de vue. Ce regard sélectif permet, d'une part de ne voir que les attributs du concept qui sont pertinents pour le point de vue en question et d'autre part, de structurer les instances du concept dans une hiérarchie de classes significative pour le point de vue. Ainsi, chaque point de vue d'un concept détermine une hiérarchie particulière de classes et un sous-ensemble des attributs du concept à prendre en compte, les attributs "visibles" pour ce point de vue. Une instance du concept est visible depuis tous les point de vue du concept ; chaque point de vue en a une vision partielle (ses valeurs pour les attributs visibles pour ce point de vue) (Fig. 5.7.).

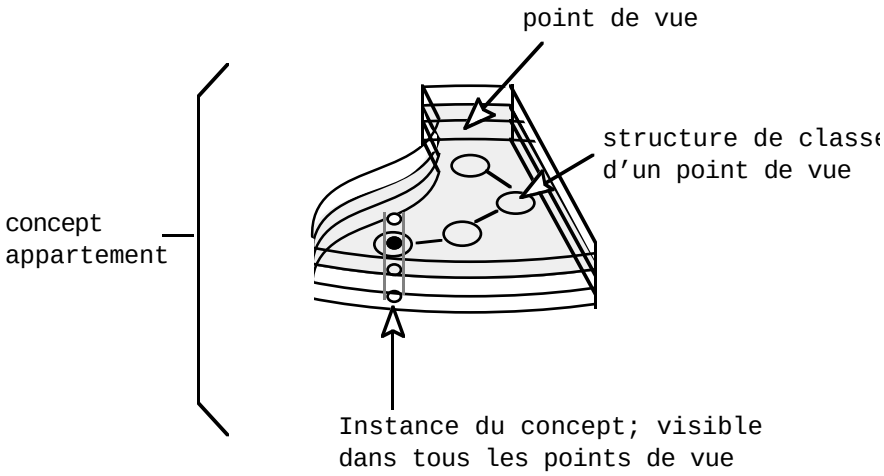


Fig. 5.7. Un concept est structuré en points de vue : chaque point de vue détermine un sous-ensemble d'attributs du concept et une structure de spécialisation de classes ; une instance du concept est visible depuis tous les points de vue.

5.3.1. Attributs visibles depuis un point de vue

Un concept représente une famille d'objets ayant des caractéristiques communes. Un point de vue du concept reflète un intérêt particulier dans l'observation des objets de ce concept et de leurs caractéristiques ; chaque point de vue considère un sous-ensemble de ces caractéristiques. Ainsi, dans TROPES, de chaque point de vue, seuls les attributs qui ont un sens pour ce point de vue sont visibles, ce sont les **attributs visibles** pour le point de vue.

Les instances du concept sont partiellement visibles depuis tous les points de vue ; pour que tout point de vue ait une identification complète des instances du concept, il doit pouvoir connaître les attributs clés du concept ; ces attributs doivent donc faire partie des attributs "visibles" de tous les points de vue. Les autres attributs du concept sont visibles pour les points de vue qui, par leur nature, s'y intéressent. Ainsi, pour le concept *Appartement* de notre exemple, le point de vue *Taille* contient les attributs qui concernent la conception, l'architecture et la distribution de l'appartement, tandis que le point de vue *Localisation* prend en compte l'environnement de l'appartement et *Financier* s'occupe des dépenses de location de l'appartement (Fig. 5.8).

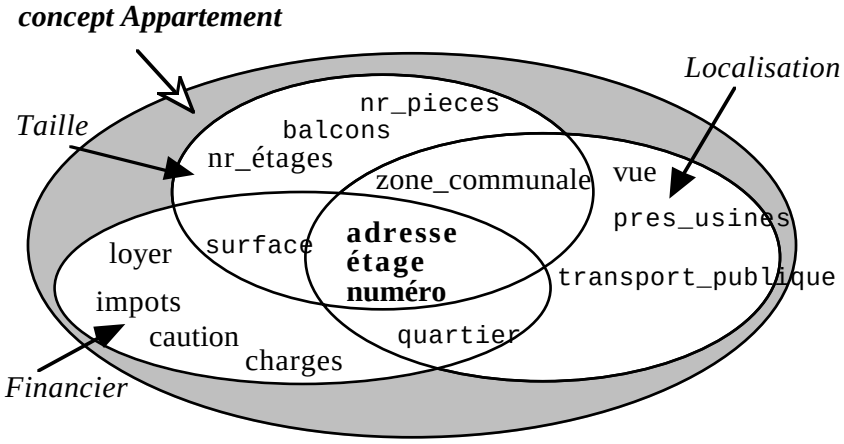


Fig. 5.8. Les attributs clés sont visibles depuis tous les points de vue. Les autres attributs depuis un ou plusieurs points de vue. Par exemple l'attribut *quartier* est visible du point de vue *Localisation,* mais aussi du point de vue *Financier* , car il est nécessaire pour calcules les *impôts.*

Les **attributs visibles** d'une instance dans un point de vue sont les attributs visibles dans ce point de vue .

5.3.2. Structure d'un point de vue

Un point de vue d'un concept détermine, non pas seulement les attributs visibles, mais l'organisation des instances du concept dans une structure particulière de spécialisation de classes ; ce graphe de spécialisation est induit par les attributs visibles et la connaissance du concept associée à ce point de vue.

La structure de spécialisation de classes comporte les éléments classiques des représentations à objets : classes et instances et leurs relations **sorte-de** et **est-un** : une classe représente un ensemble d'instances, décrit par un schéma d'attributs et descripteurs (§ 5.5). Les classes dans la hiérarchie d'un point de vue sont liées par le lien de spécialisation sorte-de qui correspond à l'inclusion ensembliste (§ 2.2.5). Une classe (appelée sur-classe) est spécialisée dans des classes plus restrictives (ses sous-classes) qui représentent des sous-ensembles d'instances de l'ensemble représenté par la sur-classe. La relation de spécialisation est transitive, une classe est sous-classe de la classe qu'elle spécialise (sa sur-classe directe) mais aussi des sur-classes de celle-ci.

Le graphe de classes d'un point de vue d'un concept a une structure d'**arbre** (§ 2.3.4). La classe racine de l'arbre représente l'ensemble universel des instances du concept ; elle n'a pas de sur-classes. Toutes les autres classes de l'arbre de spécialisation ont une seule sur-classe directe[1]. Ainsi, les classes d'un concept sont organisées dans un forêt d'arbres, chaque arbre correspondant à un point de vue sur le concept : ces arbres ont tous comme classe racine une classe qui représente tous les individus du concept (et qui possède en général le nom du concept) (Fig. 5.9). De plus, on suppose que les classes d'un même niveau d'un arbre décrivent des ensembles d'individus mutuellement exclusifs.

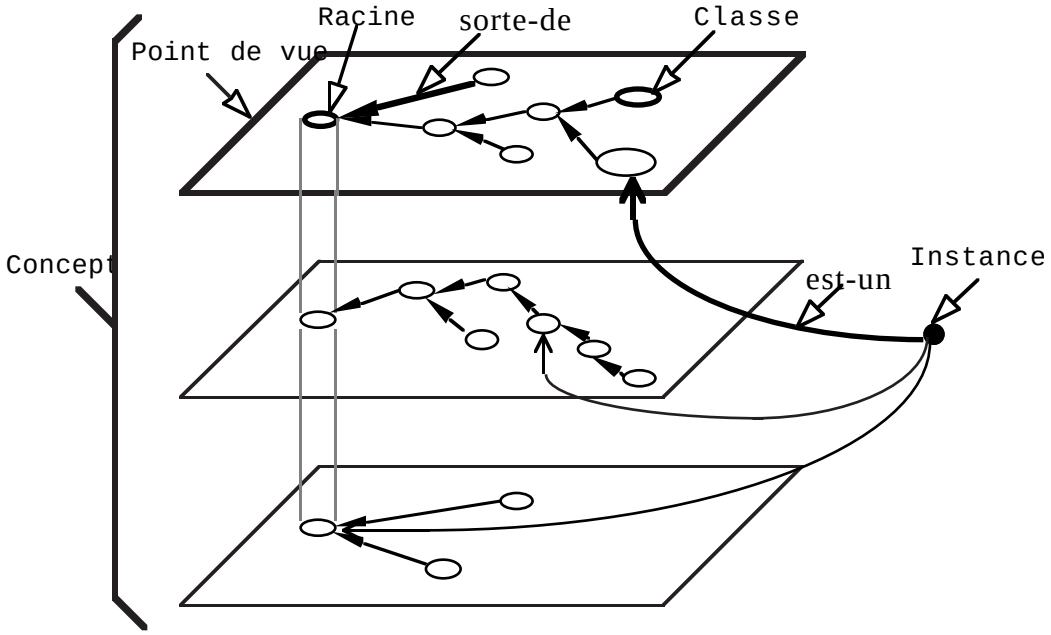


Fig. 5.9. Un concept TROPES est structuré selon différents points de vue, chacun organisé dans un arbre de spécialisation de classes. Les racines de ces arbres décrivent l'univers d'instances du concept. Les instances du concept sont visibles dans tous les points de vue.

[1] Une justification de cette prise de position est présentée dans (§2.4) et plus en détail dans [MAR89].

Une instance du concept est visible depuis tous les points de vue du concept : ils connaissent tous au moins sa clé. Dans chaque point de vue, une instance est rattachée à sa classe d'appartenance la plus spécialisée (§ 2.2.4). L'hypothèse d'exclusivité des classes sœurs d'un arbre d'un point de vue entraîne la **mono-instanciation au niveau d'un point de vue** : chaque instance ne peut appartenir qu'à une seule classe (et à ses sur-classes). Par exemple, un appartement, vu du point de vue Taille est, soit une chambre, soit un studio, soit un F1, soit un F2, soit un appartement de plus de trois pièces … mais un seul d'entre eux (Fig. 5.10).

Au niveau du concept, une instance a autant de liens d'appartenance qu'il y a de points de vue pour le concept. En effet, toute instance est visible dans tous les points de vue et dans chaque point de vue elle peut se rattacher à une classe d'appartenance (dans le pire de cas à la classe racine). Ainsi, TROPES autorise la **multi-instanciation au niveau du concept**.

Dans l'exemple de la base immobilière, on peut identifier trois points de vue pour le concept *Appartement* : le point de vue *Localisation*, le point de vue *Taille* et le point de vue *Financier*. Le point de vue *Localisation* distingue les appartements situés au centre ville de ceux situés dans des quartiers résidentiels, industriels ou dans la banlieue. Dans le point de vue *Taille,* on a des chambres, des studios, des F1, des F2, des F3 et des appartements de plus de 3 pièces. Enfin, pour le point de vue *Financier,* on peut définir des classes HLM, prix moyen, haut prix. Une instance du concept est visible des trois points de vue et elle est rattachée à une classe d'appartenance dans chacun des points de vue : par exemple un appartement de plus de trois pièces situé dans un quartier résidentiel correspond, dans le modèle, à une instance du concept *Appartement* liée aux classes *résidentiel* du point de vue *Localisation*, à la classe *Plus_de_3* du point de vue *Taille* et à la classe racine *Appartement* du point de vue financier pour lequel le système ne possède pas d'information particulière (Fig. 5.10).

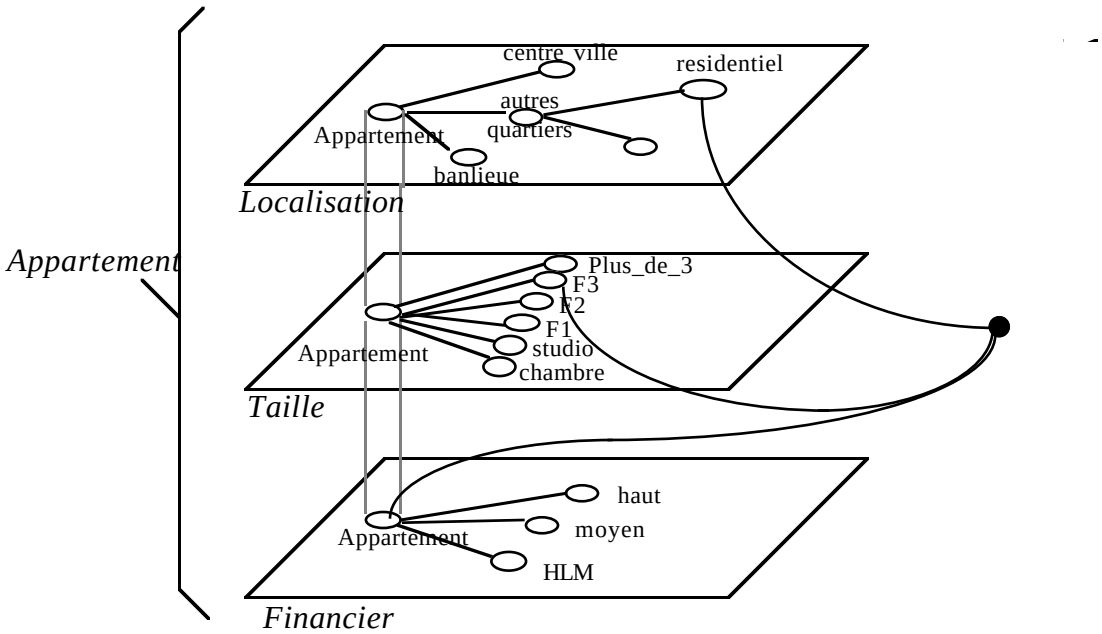


Fig. 5.10. Arbres de classes des points de vue *Localisation*, *Taille* et *Financier* du concept *Appartement*, et description d'un appartement de plus de 3 pièces dans un quartier résidentiel, pour lequel on n'a pas d'information financière.

5.4. Les passerelles entre points de vue

Les différents points de vue d'un concept produisent des arbres de classes différents. Cependant ces arbres ne sont pas complètement indépendants les uns des autres ; ainsi, par exemple, dans tous les points de vue, la classe racine représente le même ensemble potentiel d'individus, l'ensemble de tous les individus du concept. Dans chaque point de vue cet ensemble est décrit en terme des attributs propres au point de vue (les intensions des racines peuvent être différentes), mais toute instance de la racine dans un des points de vue est instance de la racine dans tous les autres points de vue (toutes les racines décrivent la même extension).

Ainsi, par exemple, les classes racines des points de vue *Personnel* et *Financier* du concept *Locataire* de l'exemple précédent possèdent la même extension : tous les locataires potentiels de la base , mais elles ont des intensions différentes (Fig. 5.11)

```
classe
{ Locataire / Personnel
  age       un   entier ;
  situation-de-famille    un cha ne
                          domaine   c libataire ,   marie ,   veuf   ;
}

classe
{ Locataire / Financier
  profession    un cha ne ;
  activit   un chaine
            domaine    tudiant ,   salarier ,
                            ch meur , retrait  ,   lib ral   ;
  revenus_annuels un r el ;
}
```

Fig. 5.11. La classe racine du point de vue *Personnel* du concept *Locataire* décrit tous les locataires potentiels de la base (ils ont tous un age et une des trois situations de famille décrites) ; la classe *Locataire* du point de vue *Financier* représente aussi tous les locataires.

Cette égalité ensembliste entre deux classes de deux points de vue différents d'un même concept s'appelle en TROPES une passerelle bidirectionnelle. En général, une passerelle TROPES exprime une inclusion ensembliste entre deux classes de deux points de vue différents d'un même concept. Par la suite, nous explicitons les différents types de passerelles de TROPES.

5.4.1. Passerelle unidirectionnelle

Une passerelle unidirectionnelle exprime l'inclusion ensembliste entre l'extension d'une classe d'un point de vue, **source** de la passerelle, et celle d'une classe d'un autre point de vue, **destination** de la passerelle (Fig. 5.12). En termes de logique, la passerelle peut être vue comme une implication: être instance de la classe source implique être instance de la classe destination[1]. Si on instancie la classe source, l'instance créée est automatiquement rattachée à la classe destination.

En termes de schémas de description des classes : la satisfaction de toutes les contraintes des attributs de la classe source entraîne la satisfaction de toutes les contraintes des attributs de la classe destination.

[1] Les passerelles dans TROPES correspondent à peu près aux règles dans les logiques terminologiques [PAT&91]

Soit C un concept, PV_1 et PV_2 deux points de vue de C et soient E une classe de C du point de vue PV_1 (notée C.E / PV_1) et D une classe de C du point de vue PV_2 (noté C.D / PV_2). Une passerelle unidirectionnelle de E vers D décrite par :

```
pass
{
   de = { C.E / PV1 } ;
   vers = C.D / PV2
}
```

correspond aux assertions :

$\forall x \in C, x \in E / PV_1 \Rightarrow x \in D / PV_2$, ou bien

$\forall x \in C$, x satisfait les contraintes de E / $PV_1 \Rightarrow$ x satisfait les contraintes de D / PV_2

Par exemple, si l'on sait que les jeunes locataires ne sont pas imposables, on peut exprimer cette connaissance pour le concept *Locataire* de la base immobilière comme une passerelle allant de la classe *Jeune_Locataire* du point de vue *Personnel* vers la classe *Non_Imposable* du point de vue *Financier*, de façon à indiquer que toute instance du concept *Locataire* qui appartient à la classe *Jeune_Locataire* appartient aussi à la classe *Non_Imposable* (Fig. 5.12).

```
pass
{
   de = { Locataire.Jeune_Locataire/ Personnel } ;
   vers = Locataire.Non_Imposable / Financier
}
```

Fig. 5.12. Passerelle unidirectionnelle entre la classe jeune_Locataire du point de vue Personnel et la classe Non_Imposable du point de vue Financier. Tout jeune locataire est non imposable.

Il est important de voir que l'implication inverse n'est pas garantie. Ainsi, par exemple, il peut y avoir des locataires non imposables qui ne soient pas des jeunes (des salariés ayant un très bas salaire ou les chômeurs).

5.4.2. Passerelle avec plusieurs sources

Une passerelle unidirectionnelle établit une implication entre une classe source et une classe destination : toute instance de la classe source est instance de la classe destination. Dans certains domaines d'application, l'instance doit appartenir à plusieurs classes de différents points de vue pour que l'on puisse déduire son appartenance à une classe destination d'un autre point de vue. Dans ce cas, qui généralise le cas précédent, une passerelle est décrite par la liste de ses classes sources des différents points de vue et la classe destination d'un autre point de vue.

Soit C un concept, PV_1 , …,PV_i des points de vue de C et soit E_1 ,…,E_i des classes de C, E_i étant une classe du point de vue PV_i (notée C.E_i / PV_i) et soit D une classe de C du point de vue PV_n, n≠1,…,i. Une passerelle unidirectionnelle des E_i vers D décrite par :

```
pass
{
   de = { C.E1 / PV1, C.E2 / PV2, , C.Ei / PVi} ;
   vers = C.D / PVn
}
```

correspond aux assertions :

$\forall x \in C, x \in E_1 / PV_1 \wedge x \in E_2 / PV_2 \wedge \ldots \wedge x \in E_i / PV_i \Rightarrow x \in D / PV_n$

$\forall x \in C$, x satisfait les contraintes de E_1 / PV_1 et celles de E_2 / PV_2 et celles de $E_i / PV_i \Rightarrow$ x satisfait les contraintes de D / PV_n .

Supposons, par exemple, que l'on sache que tous les appartements de plus de trois pièces qui se trouvent en centre ville sont chers ; on peut exprimer cette information dans TROPES par une passerelle indiquant que tout *appartement* qui est à la fois instance de la classe *plus_de_3_pièces* du point de vue *Taille* et instance de la classe *centre_ville* du point de vue *Localisation*, est aussi instance de la classe *haut_prix* du point de vue *Financier*. (Fig. 5.13).

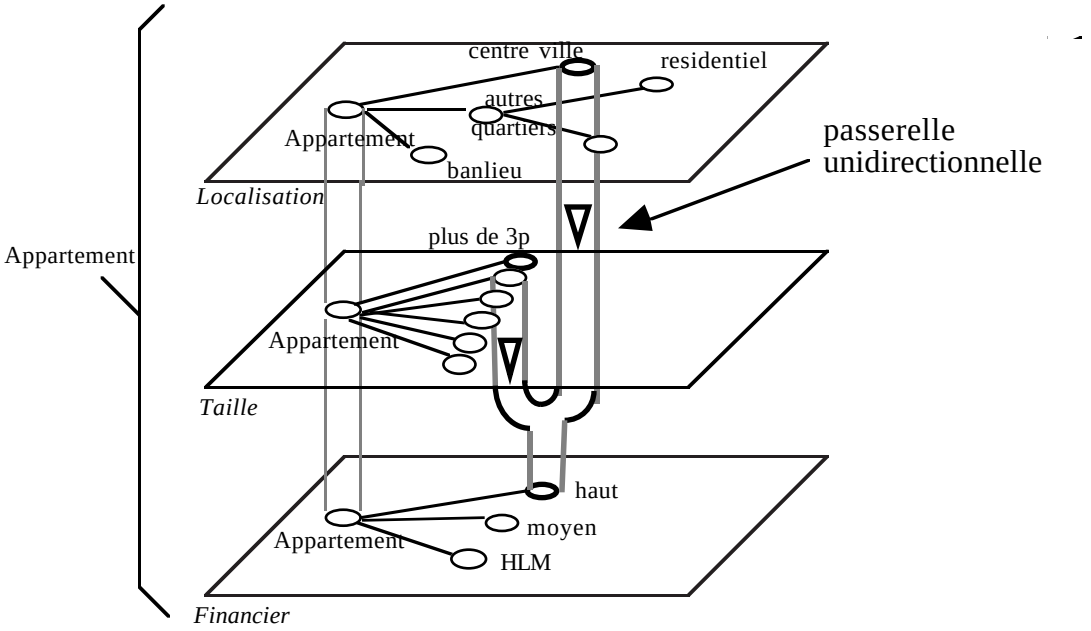


Fig. 5.13. Une passerelle peut avoir plusieurs sources (mais une seule destination). Pour appartenir à la classe destination, une instance doit appartenir à toutes les classes sources. La conjonction de l'appartenance de l'instance à chaque classe source entraîne son appartenance à la destination.

Le cas précédent est décrit en TROPES par le schéma de passerelles suivant (Fig. 5.14).

```
pass
{
  de = {   Appartement.plus_de_3p/Taille,
                Appartement.centre_ville/Localisation  }  ;
  vers =   Appartement.haut_prix  /  Financier
}
```

Fig. 5.14. Description d'une passerelle avec plusieurs sources. Toute instance du concept Appartement, appartenant aussi bien à la classe plus_de_3p du point de vue Taille qu'à la classe centre_ville selon Localisation, appartient à la classe haut_prix du point de vue Financier.

5.4.3. Passerelle bidirectionnelle

Une passerelle bidirectionnelle, entre deux classes C et D de deux points de vue différents PV1 et PV2 d'un même concept, représente l'égalité ensembliste ; une telle passerelle est équivalente à deux passerelles unidirectionnelles : l'une ayant comme source la classe C et comme destination la classe D et l'autre, l'inverse, ayant comme source la classe D et comme destination la classe C. La définition d'une passerelle

bidirectionnelle est donc équivalente à la définition des deux passerelles unidirectionnelles correspondantes.

Ainsi, par exemple, si l'on veut indiquer que tous les HLM sont dans la banlieue et que tous les appartements de banlieue sont des HLM, on peut définir une passerelle bidirectionnelle entre ces classes (Fig. 5.15).

```
pass
{
   de  =  {  Appartement.Banlieue/Localisation}  ;
   vers  =  Appartement.HLM/Financier
}

pass
{
   de  =  {  Appartement.HLM/Financier  }  ;
   vers  =  Appartement.Banlieue/Localisation
}
```

Fig. 5.15. Une passerelle bidirectionnelle est comme une double implication logique ; elle se décrit par deux passerelles unidirectionnelles opposées.

Outre la façon explicite de définir une passerelle bidirectionnelle, TROPES offre une façon implicite, en donnant le même nom aux deux classes de la passerelle ; en effet, l'espace des noms des classes d'un concept étant le concept entier, deux classes de deux points de vue différents ayant le même nom sont reconnues par le système comme représentant le même ensemble potentiel d'instances. C'est le cas des classes racines de toutes les points de vue d'un concept, qui ont toutes le même nom, celui du concept. Ayant le même nom elles sont reliées, deux à deux, automatiquement, par une passerelle bidirectionnelle implicite.

5.4.4. Passerelles et spécialisation de classes

Les classes d'un point de vue d'un concept sont structurées dans un graphe induit par la relation de spécialisation, ≤. Cette relation représente l'inclusion stricte des ensembles potentiels décrits pas les classes. Ainsi, B/PVi ≤ A/PVi si l'ensemble potentiel des instances de B est un sous-ensemble strict de l'ensemble potentiel des instances de A (le schéma de A affine les attributs du schéma de B) (§ 5.5.2).

Une passerelle unidirectionnelle entre deux classes de deux points de vue différents établit aussi une relation d'inclusion ensembliste (§ 5.4.1). La propriété de transitivité de la relation d'inclusion ensembliste (qu'elle soit exprimée par l'affinement d'un schéma de classe ou par une passerelle) permet l'identification des passerelles implicites dans un concept.

Soit E une classe d'un point de vue PV_j et B et A deux classes d'un même point de vue PV_i $i \neq j$, A étant sur-classe de B (noté $B/PVi \leq A/PVi$). S'il existe une passerelle de A / PV_i vers E / PV_j (noté $A / PV_i \Rightarrow E / PV_j$), alors l'ensemble potentiel d'instances de A/ PV_i est inclus dans celui de E/PV_j et, comme l'ensemble potentiel d'instances de B/ PV_i est inclus dans celui de A/ PV_i , on déduit, par transitivité, que l'ensemble potentiel d'instances de B/ PV_i est inclus dans celui de E/ PV_i , donc qu'il y a une passerelle de B/ PV_i vers E/ PV_j :

Si $B / PV_i \leq A / PV_i$ et $A / PV_i \Rightarrow E / PV_j$, alors $B / PV_i \Rightarrow E / PV_j$

En général, s'il y a une passerelle allant d'une classe B d'un point de vue PV_i vers une classe E d'un autre point de vue PV_j, alors il y a une passerelle implicite de chacune des sous-classes de B vers E (Fig. 5.16).

Enfin, si l'on définit un chemin sur un concept comme une suite de classes : C_1-> C_2,...-> C_r , où la classe C_{i+1} est soit une sur-classe directe de C_i , soit la classe destination d'une passerelle ayant comme source C_i , alors le chemin est une suite monotone (croissante selon l'ordre d'inclusion ensembliste) : il ne peut pas y avoir de chemins avec des cycles autres que ceux produits par des passerelles bidirectionnelles (où la relation de spécialisation n'intervient pas). Par exemple, si A/PVi ⇒ E/PVj et E/PVj ≤ F/PVj, alors on ne peut pas définir une passerelle : F/PVj ⇒ A/PVj , car cela donnerait un cycle A,E,F,A faisant intervenir la relation E ≤ F.

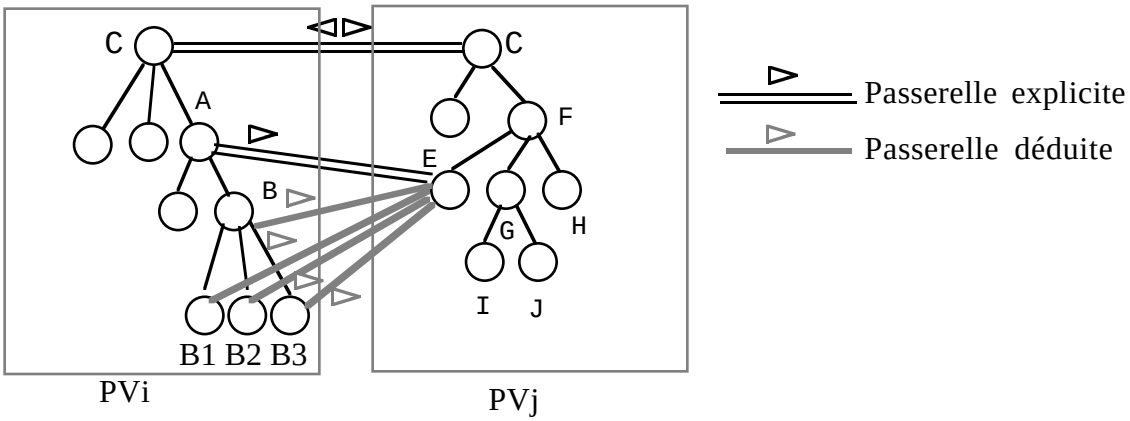


Fig. 5.16. La passerelle unidirectionnelle explicite entre la classe A/PVi et la classe E /PVj entraîne l'existence des passerelles implicites de chaque sous-classe de A/PVi vers la classe E/PVj. L'ordre ainsi étendu interdit la création de passerelles contradictoires comme par exemple de F/PVj vers B/PVi.

Lors de la création d'une base de connaissances TROPES, le système vérifie que les passerelles définies par le concepteur ne mènent pas aux contradictions comme celles du cycle précédent. Cette vérification garantit la correction des inférences faites par le mécanisme de classification d'instances, qui utilise les passerelles comme raccourcis lors du placement d'une instance dans un concept (§ 6).

5.5. Les classes d'un point de vue

Un point de vue TROPES comporte les éléments classiques des représentations à objets (§ 1.3.4, § 2.2, § 4.3.1.2) : classe, attribut, descripteur, instance, etc., avec le sens classique, c'est-à-dire qu'une classe représente un ensemble d'individus du concept décrit par la liste de ses attributs communs, le schéma de la classe. Les attributs sont définis en termes de descripteurs et représentent les propriétés, composants et relations des instances de la classe ; les descripteurs d'un attribut permettent de restreindre son domaine. Une classe en spécialise une autre par l'ajout d'attributs ou de contraintes sur les attributs existants. Enfin, une instance est un individu particulier du concept rattaché aux classes plus spécialisées du concept pour lesquelles il valide les contraintes.

Dans TROPES, la description et la sémantique de ces éléments sont légèrement modifiées pour prendre en compte les notions nouvelles de concept et de point de vue. De plus, les descriptions des attributs sont basées sur la notion de types et d'arbre de types, ce qui permet la vérification de la cohérence du graphe à tout moment de son évolution.

Une classe TROPES est définie pour un point de vue d'un concept ; elle est identifiée de façon unique par son nom, le nom de son concept et celui de son point de vue. Une classe est définie en intension par une structure (§ 4.3.1). La structure d'une classe, appelée **schéma** d'une classe, décrit la classe en termes de ses sur-classes et des modifications ou ajouts d'attributs : le schéma a une description partielle des attributs de la classe ; la **description complète** peut être retrouvée en combinant l'information donnée explicitement dans la classe avec celle héritée des schémas des sur-classes. Enfin, la description complète d'une classe ne comporte que les attributs pour lesquels elle ou ses sur-classes établissent des contraintes ; le **type** de la classe étend cette description pour prendre en compte tous les attributs du concept.

5.5.1. Schéma d'une classe

Le schéma d'une classe comporte le nom de sa sur-classe directe et un ensemble d'attributs du concept, visibles dans ce point de vue. Les attributs présents dans une classe sont les attributs pour lesquels la classe ajoute des contraintes par rapport à ses sur-classes. Chaque attribut de la classe a une liste de descripteurs qui établissent de nouvelles contraintes pour l'attribut dans la classe (et qui réduisent ainsi l'ensemble de valeurs possibles pour cet attribut pour les instances de la classe). La classe racine du point de vue n'a pas de sur-classes ; elle est décrite seulement par ses attributs.

Dans un point de vue PV_i, une classe D d'un concept C est décrite par sa sur-classe directe D' (unique dans le point de vue)[1] et par un ensemble d'attributs : a_{c1}, a_{c2},...,a_{cx} , visibles dans PV_i avec leurs contraintes dans D, t_1, t_2, ... t_x :

C. D / PV_i = D' + [$a_{c1} : t_1$, $a_{c2} : t_2$,...,$a_{cx} : t_x$]

Le schéma d'une telle classe est écrit dans TROPES par :

```
classe
{ C. D / PVi
    sorte_de D  ;
    ac1  d11, d12   ;
    acx  dx1, dx2  ;
}
```

où la description t_i de l'attribut i est donnée en termes des descripteurs d_{i1}, d_{i2},… qui le décrivent.

Par exemple, la classe racine *Locataire* du concept *Locataire* du point de vue *Financier* est décrite par les attributs *profession, activité* ("étudiant" , "travailleur" , "chômeur" ou "retraité") *et revenus_annuels.* La classe Travailleur du point de vue *Financier* du concept *Locataire* peut être définie à partir de la classe racine *Locataire :* un travailleur est un *Locataire* dont l'activité ne peut être que "travailleur" (Fig. 5.17) :

```
classe
{ Locataire. Locataire / Financier
  profession un cha ne ;
  activit  un chaine
           domaine   tudiant ,  travailleur ,  ch meur ,
                          retrait   ;
  revenus_annuels  un  r el;
}
```

[1] L'identification complète de la classe D' est C / D' / PVi, mais le nom du concept et celui du point de vue restent implicites quand il n'y a pas d'ambiguïté.

```
classe
{ Locataire. Travailleur / Financier
  sorte_de      Locataire ;
  activit       domaine   travailleur   ;
}
```

Fig. 5.17. La classe *Travailleur* du point de vue *Financier* du concept *Locataire* est sous-classe de la racine dont elle hérite les attributs *profession*, *activité* et *revenus_annuels* et leurs restrictions.

5.5.2. Description complète d'une classe

Le schéma d'une classe montre, d'une part, la relation de la classe avec les autres classes de la base de connaissances et, d'autre part, une description partielle des attributs de la classe. La description complète des attributs de la classe est répartie dans les schémas de la classe et de ses sur-classes. La description complète d'une classe est l'union des descriptions complètes de ses attributs. La description complète d'un attribut d'une classe est obtenue en héritant ses descriptions des sur-classes de la classe et en le normalisant[1]. Cette description complète et normalisée est appelée **le type de l'attribut dans cette classe**.

La description complète de la classe D du point de vue PVi du concept C est la liste des descriptions complètes de ses attributs : C. D / PV_i = [a_{c1}: T_1, a_{c2} : T_2,...,a_{cx}: T_x], où T_i est le type de a_{ci} dans D.

Ainsi, par exemple, la description complète de la classe *Travailleur*, sous-classe de *Locataire*, obtenue à partir de l'information héritée de la classe *Locataire*, est :

Locataire. Travailleur / Financier = [profession un chaine,
activité un chaine domaine "travailleur",
revenus_annuels un reel]

Les attributs d'une classe C peuvent être hérités, affinés ou des ajoutés dans la classe. Les attributs **hérités** sont déjà présents dans les sur-classes de C et ne sont pas donnés explicitement dans la description de C. La description complète de ces attributs se trouve en parcourant les liens sorte-de (du point de vue) à partir de la sur-classe directe de C. Les attributs **affinés** sont déjà présents dans une sur-classe de C, mais leurs contraintes sont modifiées dans la classe pour restreindre les valeurs permises. Pour obtenir la description complète d'un tel attribut, il faut parcourir les liens sorte-de en partant de la classe C. Enfin, les attributs **ajoutés** dans la classe C ne sont pas présents dans ses sur-classes. Leurs descriptions complètes pour C se trouvent dans le schéma de la classe C. L'information complète de la description d'un attribut pour une classe, après le parcours des classes concernées, est une liste complète des descripteurs contraignant l'ensemble de valeurs possibles pour l'attribut. Cette liste de descripteurs peut être normalisée pour arriver à une forme unique simplifiée qui décrit le **type de l'attribut**[2] pour cette classe (dans la section § 5.6.3 nous présentons les règles d'affinement et normalisation d'attributs).

La description complète d'une classe, c'est-à-dire l'ensemble des informations relatives aux attributs de la classe (possédées et héritées), est calculée lors de sa création et elle est mise à jour à chaque modification du schéma de la classe ou d'une de ses sur-classes. Cette description rend plus rapide la vérification de l'appartenance d'une instance à une classe (§ 6).

---

[1] La normalisation d'un attribut est expliquée dans § 5.6.3.

[2] Type d'un attribut pour une classe signifie le type normalisé.

5.5.3. Type de la classe

La description complète d'une classe est donc l'union des types des attributs présents dans la classe (c'est-à-dire des attributs pour lesquels la classe impose des contraintes, de façon explicite ou par héritage). Cet ensemble d'attributs doit faire partie des attributs visibles du point de vue de la classe.

Le type de la classe est l'union des types pour cette classe de TOUS les attributs du concept. Ainsi, à la différence de la description complète, le type de la classe comporte, outres les attributs définis ou affinés dans la classe, tous les autres attributs du concept[1]. Pour les attributs qui ne sont pas présents dans la classe, le type pour la classe est le type maximal donné lors de la définition de l'attribut dans le concept, car ni la classe ni ses sur-classes n'imposent de restrictions additionnelles.

Ainsi, par exemple, le type de la classe *Travailleur* du concept *Locataire* du point de vue *Financier* inclut les attributs age et situation de famille avec leurs types maximaux :

Locataire. Travailleur / Financier = [age un entier,
situation-de-famille un chaine,
profession un chaine,
activité un chaine domaine "travailleur",
revenus_annuels un reel]

Le type d'une classe d'un concept `C` est utilisé pour vérifier la cohérence des points de vue de `C` lors des modifications sur leurs schémas de classes ; il est aussi pris en compte lorsque des attributs d'autres concepts prennent des valeurs dans `C`. Cependant, les mécanismes de classification de TROPES n'utilisent pas le type d'une classe mais son schéma et sa description complète. En effet, pour établir si une instance du concept appartient à la classe, il suffit de vérifier qu'elle satisfait la description complète de la classe ; si l'instance satisfait la description complète de la classe, elle satisfait aussi son type (car toute instance du concept satisfait le type maximal des attributs du concept).

5.6. Les attributs d'une classe

Un attribut est décrit dans un schéma de classe par son nom et une liste de descripteurs. Ces descripteurs peuvent restreindre le type ou le domaine des valeurs de l'attribut, ou bien proposer une valeur par défaut. L'ensemble des descripteurs de TROPES est un sous-ensemble des descripteurs présentés dans § 2.2.3. (en particulier le descripteur d'attachement procédural n'est plus présent dans les classes mais au niveau du concept [REC92])

5.6.1. Descripteurs d'un attribut dans un schéma de classe

Descripteurs de type

Un attribut d'un concept prend ses valeurs dans un type de base. Ce type de base peut être un domaine primitif comme entier, chaîne, etc ou bien un domaine défini, formé par l'univers d'instances d'un autre concept de la base. Ainsi, par exemple dans un

[1] L'idée de définir le type d'une classe comme la composition des types de ses attributs est inspirée des travaux de Wegner [WEG87]et de Cardelli [CAR85], [CAR&85], [CAR&91].

concept composé *Famille* d'une base de connaissances, l'attribut *nom* prend ses valeurs dans le domaine primitif chaîne, tandis que l'attribut *fils* les prend dans le domaine défini *Personne*.

Dans une classe, le type d'un attribut peut être restreint à un sous-type du type de base (les descripteurs utilisés sont **un** pour les attributs mono-valués et **ens_de** pour les multi-valués). Ainsi, par exemple, si la racine du concept *Famille* se spécialise en des classes *Une_génération* et *Plus_d'une_générations,* alors l'attribut *fils* peut être restreint dans la classe *Une_génération* aux types des classes *Enfant* ou *Adolescent* (sous-classes de la classe *Personne*).

Lorsque le type de base d'un attribut d'un concept `C` est un domaine défini (un autre concept `D`) - `C` est un concept composé - le type de cet attribut dans une classe peut être restreint à un groupe particulier d'instances de `D`. Ce groupe d'instances peut être décrit

- soit par une classe d'un point de vue du concept, ce qui indique que toutes les instances de cette classe sont des valeurs possibles pour l'attribut,
- soit par une conjonction de classes de points de vue différents, pour indiquer que seules les instances appartenant à toutes ces classes sont des valeurs possibles pour l'attribut,
- soit, enfin, par une disjonction de classes d'un même point de vue : toute instance d'une de ces classes est une valeur valide pour l'attribut (c'est le cas de l'attribut fils de l'exemple précédent).

Par exemple, on peut définir pour la base immobilière les locataires riches (du point de vue financier) comme des personnes qui habitent dans un grand appartement au centre ville et (&) qui ont des voitures de luxe ou (|) classiques (Fig. 5.18) :

```
classe
{ Locataire. Riche / Financier
  sorte-de Locataire ;
  habite   un       centre_ville/Localisation  &
                    plus_de_3/Taille  ;
  voiture  ens_de   de_luxe/Physique  |  classique/Physique
  ...
}
```

Fig. 5.18. Dans la classe Riche, l'attribut habite est restreint aux instances appartenant à la classe centre_ville du point de vue localisation **et** (&) à la classe plus_de_3 du point de vue taille ; l'attribut voiture peut prendre ses valeurs dans la classe de_luxe du point de vue physique (d'un concept automobile) **ou** (|) dans la classe classique de ce même point de vue.

Descripteurs de valeurs valides

Les descripteurs contraignant les valeurs admises pour un attribut dans une classe sont les descripteurs **domaine** et **sauf**. Le descripteur **domaine** indique l'ensemble de valeurs possibles : soit par énumération exhaustive des éléments, soit par des intervalles (si le concept est ordonné[1]). Si le domaine n'est pas donné explicitement, il comporte toutes les valeurs du type défini pour l'attribut pour cette classe. Le descripteur **sauf** indique les valeurs du domaine qui ne sont pas valides. Ainsi, par exemple, la description d'une classe *appart_bonne_chance* du point de vue *Localisation* du concept *Appartement* peut inclure pour l'attribut clé *étage* un descripteur domaine indiquant les valeurs possibles et un descripteur sauf en excluant quelques unes (Fig. 5.19)

[1]Un concept ordonné est un concept pour lequel les instances ont une relation d'ordre total : par exemple, le concept primitif *entier* et le concept définit *date* avec une relation d'ordre établie sur les attributs clés: année, mois, jours.

```
classe
{  Appartement.Appart_bonne_chance  /  Localisation
   sorte_de  Appartement  ;
    tage     domaine  [0  :15]
             sauf  13  ;

}
```

Fig. 5.19. Le domaine de valeurs valides pour étage est : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14 et 15 ; c'est-à-dire toutes les valeurs indiquées dans le descripteur domaine moins celle exclue par le descripteur sauf.

Descripteur de cardinalité

Pour les attributs à caractère multi-valué, le descripteur **card** indique le nombre de valeurs élémentaires que peut comporter la valeur de l'attribut ; card peut être une valeur simple ou un intervalle de valeurs permises. Ce descripteur sert à indique par exemple que les locataires assez riches ont au moins deux voitures et au plus quatre(Fig. 5.20).

```
classe
{  Locataire.  Assez_Riche  /  Financier
   sorte_de  Riche  ;
   ...
   voiture   card  [2  :  4]  ;
}
```

Fig. 5.20. Toute instance de la classe Assez_Riche a deux, trois ou quatre instances du concept automobile valuant son attribut voiture.

La valeur par défaut

La valeur par défaut d'un attribut est la valeur typique, la plus représentative de cet attribut pour les instances de la classe. Ainsi, par exemple, l'on peut dire que "la couleur typique de la mer est bleue". À la différence des descripteurs précédents, le descripteur par défaut n'établit pas de contraintes sur les valeurs d'un attribut, mais une façon d'inférer la valeurs la plus représentative pour manipuler les instances incomplètes. La valeur par défaut ne sert pas aux mécanismes qui font des inférences certaines ; en particulier, le mécanisme principal de raisonnement de TROPES, la classification d'instances, ne l'utilise pas. Cependant, ce descripteur peut être utilisé pour faire du raisonnement hypothétique - en particulier de la classification hypothétique (§ 8.2) lorsque la valeur réelle de l'attribut n'est pas connue pour une instance.

5.6.2. Le type d'un attribut d'une classe

La description complète d'un attribut pour une classe est donnée en terme des descripteurs de contraintes de type, domaine et cardinalité ainsi que des descripteurs d'inférence de valeurs ; ces descripteurs peuvent être introduits dans la classe ou dans une de ses sur-classes. L'union des descripteurs de contraintes (définie dans la classe ou héritées) détermine complètement le domaine de valeurs possibles pour l'attribut dans cette classe. Un même domaine de valeurs peut être décrit par différentes combinaisons de descripteurs (Fig. 4.7). Le **type d'un attribut pour une classe** est la forme normale de ces combinaisons de descriptions équivalentes. De même que pour [CAP&93], la relation de spécialisation de classes dans TROPES correspond à la relation d'ordre entre les catégories définies par intension (§ 4.3.1 p.88) où la relation de **sous-typage $\leq_t$ est définie comme la relation d'inclusion** portant sur les domaines d'attributs. Ainsi, par exemple, le type de l'attribut étage de la classe *Appart_bonne_chance* du concept *Appartement* du point de vue *Localisation* (Fig. 5.19) est l'ensemble de valeurs {0, 1, 2, 3,

4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15}, qui est sous-type du type défini par *domaine [0 : 12], [14 : 16]* (où le virgule signifie réunion) et sur-type du type défini par *domaine [8 : 11].*

5.6.3. Affinement et normalisation d'attributs

La spécialisation de classes se réalise en ajoutant des nouveaux attributs à la sous-classe ou en affinant des attributs déjà existants. Affiner un attribut a_x, dans la sous-classe C' de la classe C, consiste à ajouter des nouvelles contraintes à a_x dans C : t_{Cx}, de façon à avoir un nouvel ensemble de contraintes $t_{C'x}$ restreignant les valeurs valides pour l'attribut. L'ensemble des valeurs valides de l'attribut affiné dans la sous-classe C' (le type de l'attribut dans C') doit être un sous-ensemble de l'ensemble des valeurs valides de l'attribut dans la sur-classe C : $t_{C'x} \leq_t t_{Cx}$.

Ajouter un attribut à une classe revient aussi à faire un enrichissement. En effet, comme nous l'avons présenté auparavant (§ 5.5.3), le type d'une classe C comporte tous les attributs du concept, aussi bien ceux qui sont présents explicitement dans C ou dans une de ses sur-classes, que ceux pour lesquels aucune de ces classes n'a établi de contraintes : le type de ces derniers attributs, dans la classe C, est le type maximal donné lors de leur définition dans le concept. Ainsi, ajouter un attribut à une sous-classe C' de C consiste à ajouter des restrictions (sous forme de descripteurs) aux contraintes établies lors de la définition de cet attribut dans le concept, en réduisant ainsi le domaine de valeurs possibles.

Affinement du type

Affiner le type d'un attribut **a** d'une classe **C** dans une sous-classe **C'** consiste à définir dans C' des nouvelles classes d'appartenance permises pour les instances (valeurs) de cet attribut. Ces nouvelles classes doivent être des sous-classes des classes données comme type de a dans C (dans la hiérarchie du concept de base de l'attribut pour le point de vue correspondant). Normaliser plusieurs descripteurs de type consiste à garder le plus spécialisé.

Par exemple, on peut définir les locataires *Jeune_Riche* du point de vue *Financier* par une sous-classe de la classe *Riche* (Fig. 5.18) qui affine le descripteur *voiture* pour n'inclure que des voitures de luxe à deux portes (sous-classe des voitures de luxe) (Fig. 5.21)

```
classe
{ Locataire. Jeune_Riche / Financier
  sorte-de Riche ;
  voiture ens_de     de_luxe_sportive/Physique
  ...
}
```

Fig. 5.21. La classe Jeune_riche affine l'attribut voiture de la classe Riche en réduisant l'ensemble de voitures acceptées pour les instances Jeune_Riche à celles appartenant à la classe de_luxe_sportive, sous-classe de la classe de_luxe .

Affinement des valeurs valides

L'affinement des valeurs valides d'un attribut **a** d'une classe **C** dans une sous-classe **C'** décrit en termes des descripteurs domaine et sauf, revient à définir un nouveau domaine de valeurs permises, sous-domaine du domaine précédent. Cet affinement peut se faire en réduisant l'ensemble décrit par le descripteur domaine et/ou en augmentant les exceptions données par le descripteur sauf.

Lorsque l'on réduit le domaine d'éléments valides avec une nouvelle définition du descripteur **domaine**, le nouvel ensemble d'éléments exclus est réduit implicitement à ceux encore dans le domaine. Par exemple, si on définit une classe *Appart_bas*, sous-classe de la classe *Appart_bonne_chance* (Fig. 5.19), en affinant l'attribut étage pour inclure seulement les appartements des 5 premiers étages alors, le type de l'attribut étage pour cette classe est tout le domaine {0, 1, 2, 3, 4, 5} ; il n'y a plus d'élément exclus (car 13 n'est plus dans le domaine valide) (Fig. 5.22)

```
classe
{  Appartement.Appart_bas  /  Localisation
   sorte_de  Appart_bonne_chance  ;
    tage     domaine [0 : 5] ;

}
```

Fig. 5.22. La classe Appart_bas sous-classe de Appart_bonne_chance réduit le domaine des valeurs de l'attribut étage à [0 : 5] sauf 13, c'est-à-dire à {0,1,2,3,4,5}

La normalisation des descripteurs de valeurs valides consiste à créer l'ensemble minimal de valeurs permises en enlevant de cet ensemble les valeurs énoncées dans le descripteur sauf. Ainsi par exemple, la description : *domaine [0 : 30], sauf 12* est normalisée par les intervalles *[0 : 11] , [13 : 30].*

Affinement de la cardinalité

Affiner la cardinalité d'un attribut multi-valué consiste à réduire l'intervalle de nombres d'éléments permis. Ainsi, si dans la classe C l'attribut a est défini avec une cardinalité card [cmin : cmax], dans une sous-classe C', la nouvelle cardinalité card' [c'min : c'max] doit être incluse dans card (c'est-à-dire : c'min >= cmin et c'max <= cmax).

Affinement du défaut

La valeur par défaut d'un attribut a de la classe C peut être modifiée ou ajoutée dans une sous-classe C' de C, aucune relation n'existant entre les deux valeurs. La valeur dans la sous-classe doit évidemment être une valeur valide pour l'attribut dans cette sous-classe, c'est-à-dire qu'elle doit satisfaire les nouvelles contraintes de l'affinement de l'attribut.

5.6.4. L'arbre de types d'un attribut d'un concept

Comme nous l'avons présenté auparavant, la description complète d'un attribut pour une classe détermine le type de cet attribut pour la classe, l'ensemble des valeurs possibles. Affiner un attribut att dans une classe C consiste à déterminer un sous-ensemble de l'ensemble des valeurs possibles de la sur-classe directe de C, c'est-à-dire à décrire un sous-type du type de l'attribut att dans C'. Ainsi, la relation de sous-typage est en accord avec la relation de spécialisation de classes. Pour chaque attribut d'un point de vue du concept, TROPES garde un arbre des différents types de cet attribut dans les classes de ce point de vue du concept : les types sont organisés selon la relation de sous-typage et à chaque type d'attribut est associée la classe la plus générale contenant ce type pour cet attribut (lapremière en partant de la racine dans laquelle ce type apparaît). L'arbre des types permet un parcours du graphe de classes dirigé par les attributs, ce qui réduit l'espace de recherche lors des modifications ou vérifications ponctuelles ne concernant qu'un attribut.

5.7. Les instances d'un concept

Une instance TROPES est un individu particulier d'un concept, identifiable de façon unique par sa clé dans le concept, et différent de tous les autres individus qui dérivent de ce concept. Elle est décrite en termes de sa **clé**, de ses classes d'appartenance les plus spécialisées dans les différents points de vue, qui déterminent son **type** (minimal) et d'une liste de tuples attribut-valeur pour les différents attributs du concept, qui forment sa **valeur** (Fig. 5.23). Une instance TROPES peut être complète ou incomplète, certaine ou hypothétique. Elle peut évoluer et subir des modifications au cours de sa vie.

```
C. C1/PV1 & C2/PV2 & ... & CX/PVX
{ nom_instance1
  attcl 1 = v1 ;    ;
  attcl X = vX ;
  attX+1 = vX+1 ;    ;
  attn = vn ;
}
```

Fig. 5.23. Une instance est décrite dans TROPES par sa liste de classes d'appartenance et une liste de couples attribut-valeur. Elle est identifiée par la liste des valeurs de ses attributs clés (att1…attn).

5.7.1. L'identification d'une instance

Toute instance de la base appartient à un et un seul concept. Une instance est identifiée à l'intérieur de son concept par sa **clé** : les valeurs qu'elle a pour l'ensemble d'attributs qui forment la clé du concept (une valeur par attribut). La clé est unique, c'est-à-dire que deux instances ayant les mêmes valeurs pour les attributs de la clé représentent la même instance. Outre l'identification sémantique d'une instance, on peut lui associer un nom, une étiquette qui est traitée par le système comme un synonyme de la clé.

Par exemple, une instance du concept *Appartement* est identifiée par les valeurs des attributs clés du concept : adresse, étape, numéro et éventuellement un nom. Par exemple, l'instance ("6 rue Monge 75005", 2, 4) du concept *Appartement* peut être identifiée par le nom "chez_Dugand" (Fig. 5.24).

```
Appartement.F2/ Taille & centre_ville / Localisation
{ Chez_Dugand
  adresse  =  6 rue Monge 75005  ;
   tage    = 2 ;
  num ro   = 4 ;
  surface  = 76 ;
  loyer    = 4500 ;
  nb_pi ces = 2 ;

}
```

Fig. 5.24 : L'instance *Chez_Dugand* identifiée par la clé ("6 rue Monge 75005",2,4) appartient aux classes F2 du point de vue Taille, centre_ville du point de vue Localisation et à la racine Appartement du point de vue Financier. Elle n'a des valeurs connues que pour la clé et trois autres attributs du concept.

5.7.2. Le type d'une instance

Toute instance d'un concept C possède tous les attributs de ce concept ; ainsi la structure globale de l'instance est déterminée par le concept. La structure la plus générale

d'une instance d'un concept est un enregistrement formé par les types maximaux des divers attributs du concept. Cette structure est le **type global** de toutes les instances du concept.

Lorsque l'instance est rattachée à une classe d'un point de vue du concept, son type est contraint, pour chaque attribut présent dans la classe, par le type de cet attribut dans cette classe ; autrement dit, **le type de l'instance** est l'intersection entre le type global et le type de la nouvelle classe. Comme l'instance est rattachée à plusieurs classes (une classe par point de vue), son type est l'intersection des types de toutes ces classes.

5.7.3. La valeur d'une instance

La valeur de l'instance est la liste (l'enregistrement) des couples attribut-valeur, correspondant aux attributs du concept. La valeur de l'instance "chez_Dugand" est la liste :

```
{adresse= 6 rue Monge 75005 ; tage=2 ; num ro=4 ; surface=76 ;
loyer=4500˚; nb_pi ces=2}
```

En fait la valeur d'une instance est l'instance elle-même ; elle peut être donnée explicitement comme dans l'exemple précédent ou bien en indiquant sa clé ou identification. La valeur v_X d'un attribut a_X d'une instance I satisfait le type de a_X de toutes les classes de I, c'est-à-dire le type intersection des types de a_X des classes d'appartenance de I.

Enfin, il est important de signaler qu'une instance peut être incomplète, c'est-à-dire elle peut avoir des attributs non valués (avec une valeur inconnue; marquée $\perp$). Cette caractéristique est d'ailleurs une des grandes différences entre les bases de connaissances et les bases de données ; en effet, dans ces dernières, les instances ne peuvent pas avoir des valeurs inconnues, elle doivent être complètes.

5.7.4. Relations d'appartenance

Toute instance d'un concept est visible, par sa clé, depuis tous les points de vue du concept et dans chaque point de vu, elle et rattachée à la classe la plus spécialisée pour laquelle l'instance satisfait toutes les contraintes sur les attributs.

Création d'une instance

Lors de la **création d'une instance** dans la base, l'utilisateur peut donner explicitement ses classes d'appartenance (aussi nommées, *dans ce cas,* classes d'instanciation). Ces classes sont données dans l'entête du schéma de l'instance. Si pour un point de vue la classe de rattachement n'est pas donnée, elle est supposée être la classe racine du point de vue. Par exemple, une instance du concept *Appartement* appartenant à la classe *F2* du point de vue *Taille* et à la classe *centre_ville* du point de vue *Localisation* (et pour laquelle on n'a pas d'information sur le point de vue *Financier*) (Fig 5.24) est décrite par un schéma ayant l'entête :

```
Appartement.F2/ Taille & centre_ville / Localisation
```

Au moment de la création d'une instance, l'utilisateur peut ne pas connaître toutes les valeurs des attributs du concept pour l'instance. Les valeurs connues doivent satisfaire le type de cet attribut pour l'instance, c'est-à-dire l'intersection des types (des domaines) de cet attribut dans ses différentes classes d'appartenance. Pour les autres attributs, le système peut supposer qu'ils satisfont les types de ces attributs pour les classes d'appartenance, même si leurs valeurs ne sont pas connues. Cela est vrai car lorsque l'on crée une instance I comme membre d'une classe C, l'appartenance de I à la

classe C est une **condition suffisante** pour qu'elle satisfasse le type de la classe (les contraintes de ses attributs).

Appartenance d'une instance à une classe

Pour qu'une instance puisse être rattachée à une classe, sa valeur doit satisfaire le type de la classe. Une instance satisfait le type d'une classe, si les valeurs de ses attributs satisfont les types de ces attributs pour la classe. Pour rattacher une instance incomplètes à une classe, le système doit pouvoir garantir que les attributs inconnus satisfont aussi le type de la classe. Ce cas se présente lors de la création d'une instance incomplète (cas présenté précédemment) et dans certains cas, lors du déplacement d'une instance I d'une classe C vers une sous-classe C' de C. En effet, si pour un attribut *att* du concept, l'instance a la valeur *inconnue* et la classe C' ne contrainte pas le type de cet attribut dans C, alors on sait que l'instance valide les contraintes de att dans C' (car elles sont les mêmes qu'elle validait pour C).

En général, on peut dire qu'une instance appartient à une classe (relation d'appartenance **sûre**) si et seulement si pour tout attribut de la classe, soit elle possède une valeur valide, soit elle possède la valeur *inconnue* (⊥) mais que l'on sait qu'elle satisfait le type de cet attribut pour la classe. Si une des valeurs des attributs de l'instance ne satisfait pas le type de cet attribut dans la classe, alors l'instance ne peut pas appartenir à cette classe (relation d'appartenance **impossible**). Enfin, si le système ne peut ni garantir ni nier l'appartenance de l'instance à la classe, alors la relation d'appartenance est **possible**. (La sémantique de cette relation d'appartenance à trois valeurs est donnée dans § 4.3.1 p.88)

Une instance I déjà créée et rattachée à une classe C peut migrer à une autre classe C' si elle valide le type de C'. I valide le type de C' si la valeur de tout attribut valué de I satisfait le type de cet attribut dans C' et si les contraintes connues comme étant valides pour les attributs à valeur inconnue sont compatibles avec celles imposées dans C.

Le type d'une classe D d'un concept `C` de la base établit des **conditions nécessaires** d'appartenance pour une instance I de la base. Pour une instance I_C du concept `C`, le type de D établit des conditions **nécessaires et suffisantes** d'appartenance. Donc, à l'intérieur d'un concept les types des classes établissent des conditions nécessaires et suffisantes d'appartenance.

Par exemple, si l'on regarde la classe *Enfant* dont la description complète est :

Personne. Enfant / Physique = [nom un chaîne, age un 0 : 12]

alors, pour qu'un objet soit un *Enfant* il est **nécessaire** qu'il ait un attribut *nom* de type chaîne et un attribut entier *age* avec une valeur entre 0 et 12. Mais cela n'est pas suffisante ; un effet, l'objet pourrait être un vin de 5 ans. Par contre, si de plus on spécifie que l'objet est dans le concept *Personne* , alors toute personne avec un nom et ayant un age entre 0 et 12 est un enfant. Sachant que l'objet est une personne, les conditions de la classe *Enfant* sont **nécessaires et suffisantes** pour l'appartenance.

5.7.5. L'instance vue d'un point de vue du concept

Une instance d'un concept représente un objet de l'ensemble du monde décrit par le concept. Une instance complète du concept, c'est-à-dire une représentation complète de l'individu du monde, a tous les attributs représentés dans le concept. Si l'on regarde une instance d'un concept de façon globale, on peut voir toute son information, les valeurs qu'elle a pour tous les attributs du concept. Par contre, lorsque l'on regarde une instance d'un point de vue particulier, on ne voit que les attributs de l'instance qui sont pertinents

pour (connus par) ce point de vue ; on a une vision partielle de l'instance (le point de vue établit une sorte de masque sur l'instance).

Par exemple, l'instance "chez_Dugand" décrite précédemment (Fig. 5.24) est traitée du point de vue *Taille* comme ne possèdant que les attributs qui sont pertinents pour ce point de vue : *surface, nb_pièces*, etc., tandis que le point de vue *Financier* contient les attributs *loyer, charges*, etc.(Fig. 5.25).

```
Appartement.F2/  Taille
{  Chez_Dugand
   adresse   =  6 rue Monge 75005  ;
    tage     = 2 ;
   num ro    = 4 ;
   surface   = 76 ;
   nb_pi ces = 2 ;

}

Appartement.centre_ville  /  Financier
{  Chez_Dugand
   adresse   =  6 rue Monge 75005  ;
    tage     = 2 ;
   num ro    = 4 ;
   loyer     = 4500 ;

}
```

Fig. 5.25. Instance Chez_Dugand du concept Appartement vue des points de vue Taille et Financier. Les attributs clés sont visibles des deux points de vue.

Il est important de noter que, dans chaque point de vue, une instance appartient à sa classe de rattachement et à toutes les sur-classes de celle-ci. Une instance incomplète, pour laquelle on n'a aucune information autre que la clé, appartient dans chaque point de vue à la classe racine qui décrit toutes les instances du concept. Lorsque l'on obtient plus d'information sur l'instance, celle-ci peut "descendre" dans les différents points de vue pour être rattachée aux classes plus spécialisées.

5.8. Les objets composites

Par la suite nous allons présenter la représentation d'objets composites dans TROPES. Un objet composite est un objet qui a, parmi ses attributs, des attributs de type composant. Un composant est une partie constitutive de l'objet, un objet à part entière. Pour l'explication des différentes notions qui interviennent dans la représentation des objets composites et de ses composants, nous utilisons un exemple d'une base de connaissances sur les automobiles (Fig. 5.26).

Le concept principal de la base est le concept *Automobile*, structuré selon trois points de vue : *mécanique* (décrivant une voiture en tant que machine), *physique* (aspects externes de la voiture) et d'*utilisation* (type de transport). En dehors de la passerelle existant entre les racines des points de vue, il existe une passerelle unidirectionnelle menant du point de vue *utilisation* vers le point de vue *physique*. Cette passerelle indique que tout véhicule de transport de marchandises et de matériaux est économique.

Une voiture a des propriétés, telles que sa plaque (sa clé pour l'identifier de façon unique), sa couleur, ses dimensions, son poids, etc. Puis, elle peut être décomposée de différentes façons. Du point de vue mécanique ses composants sont : le moteur, la

transmission, la suspension, la direction et le freinage ; du point de vue physique elle possède une carrosserie et des roues ; du point de vue utilisation il n'y a aucune décomposition pertinente, la voiture est vue comme un objet non-décomposable. Chaque composant de la voiture prend ses valeurs dans un concept de la base. L'attribut composant *moteur*, par exemple, prend ses valeurs dans le concept moteur, qui est structuré en deux points de vue : mécanique et électrique ; dans ce dernier le moteur est décomposé en batterie, générateur de courant, démarreur, etc. Le composant carrosserie peut être décomposé du point de vue physique en *portes* et *carroserie_fixe.*

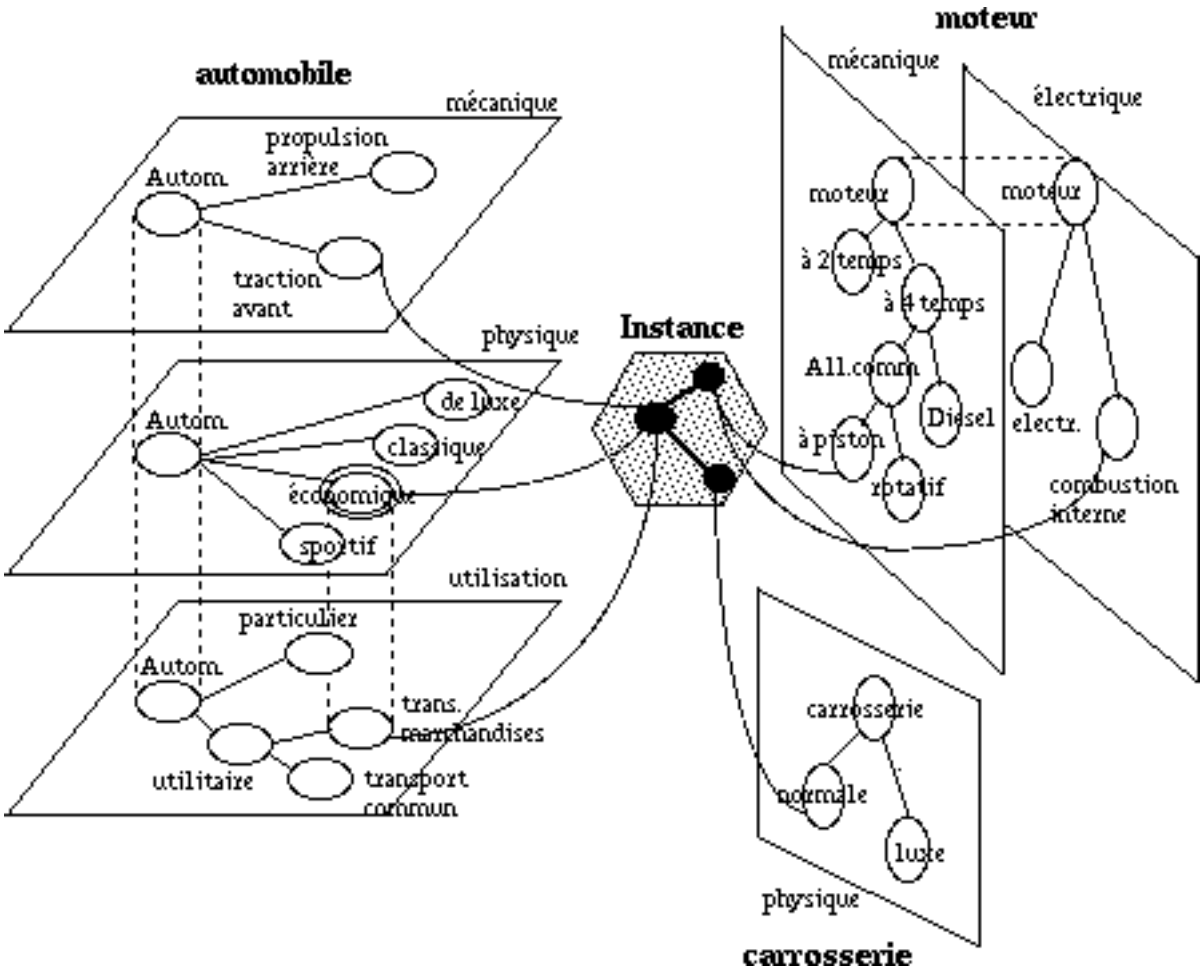


Fig. 5.26. Une automobile peut être considérée sous les points de vue mécanique, physique et utilisation. C'est aussi un objet composite ayant une décomposition physique en carrosserie, roues, etc. et une décomposition mécanique en L'automobile est un objet composite, ayant une décomposition mécanique en moteur , freins , etc. Le composant moteur, lui peut être observé du point de vue mécanique et du point de vue électrique.

Finalement, une instance particulière de voiture, "Ma_Clio5302YX38", appartient à la classe *traction_avant* du point de vue *mécanique*, à la classe *économique* du point de vue *physique* et à la classe *particulier* du point de vue *utilisation*. Du point de vue mécanique, son composant moteur est de type à piston (appartient à la classe *à_piston*) ; du point de vue physique, elle a une carrosserie *normale*, qui comporte 4 *portes.*

5.8.1. Concept composite - concepts composants

La description des objets composites dans TROPES met en rapport plusieurs concepts de la base. La relation entre le concept décrivant l'objet composite et les concepts décrivant les composants est une relation de type maître-esclave : l'accès aux composants d'un objet composite se fait toujours à partir du concept de l'objet composite

(concept maître). Cette relation maître-esclave peut avoir plusieurs niveaux d'imbrication. Dans notre exemple, le concept *porte* est esclave du concept *carrosserie* et ce dernier est esclave du concept *automobile* (le concept maître).

Au niveau des points de vue, le concept d'un composant doit avoir tous les points de vue pour lesquels le composant est accessible dans le concept maître et aucun des points de vue du concept maître dans lesquels ce composant n'apparaît pas. De plus, le concept d'un composant peut avoir des points de vue qui ne sont pas présents dans le concept composite. Ainsi, par exemple, la *carrosserie*, composant de *automobile* du point de vue *physique*, a une hiérarchie physique, de même que le *moteur*, présent dans la décomposition *mécanique* de *automobile* a un point de vue *mécanique*. Le moteur n'a pas un point de vue physique, car dans ce cas il serait présent dans la décomposition physique de la voiture ; enfin, le point de vue *électrique* du concept moteur n'est pas visible au niveau de l'automobile.

5.8.2. Le concept maître

La décomposition dans un point de vue

Les objets d'un concept peuvent être vus comme des objets simples d'un point de vue et comme des objets composites d'un autre (c'est le cas des points de vue utilisation et mécanique des automobiles). Dans un point de vue décrivant des objets composites, la décomposition peut être établie dès la classe racine, ou bien dans des classes plus spécialisées. Ce dernier cas permet d'avoir des décompositions différentes dans un même point de vue (c'est le cas des classes *à_piston* et *moteur_rotatif* du point de vue *mécanique* du concept *moteur*, car pour les moteurs rotatifs le piston est remplacé par un composant triangulaire excentré). Cependant, une fois qu'une décomposition est établie dans une classe, elle doit être conservée dans toutes ses sous-classes[1]. Ce choix, bien que restrictif, est cohérent avec le sens "décomposition structurelle" que nous avons voulu donné à la relation de composition dans TROPES. La structure d'un objet ne change pas ; une fois que l'on a reconnu dans un objet une structure de composition particulière, celle-ci celle-ci doit être conservée.

Lorsque dans une classe on parle de la décomposition des objets de la classe, on donne tous les composants ; cela veut dire que dans les spécialisations de cette classe on peut affiner les composants mais on ne peut pas en ajouter[2].

Spécialisation de classes - Affinement des attributs composants

Dans TROPES la relation de composition établit une **dépendance exclusive et existentielle** entre le composant et l'objet composite. Cela veut dire que lorsqu'une instance I d'un concept `C` devient la valeur d'un attribut composant att d'une autre instance O d'un autre concept `D`, l'instance I ne peut apparaître nulle part ailleurs dans la base, et elle ne peut pas exister sans l'objet composite. Ainsi, pour un attribut composant, les seuls descripteurs pertinents sont le descripteur **type** qui indique la classe la plus générale à laquelle doit appartenir le composant et, dans le cas des attributs multi-valués, le descripteur **cardinalité.**

Spécialiser un attribut de nature composant consiste à restreindre son type (les classes possibles des instances qui évaluent ce composant) ou sa cardinalité (pour les attributs multi-valués). Ainsi, par exemple le composant multi-valué *portes* de type *porte*

---

[1]Nous verrons par la suite qu'une spécialisation de la décomposition est pourtant possible.

[2]En fait, logiquement on pourrait traiter la "décomposition" comme un seul attribut (on ne l'a pas fait pour ne pas alourdir l'écriture des schémas de classes).

de la classe *carrosserie* est affiné dans la classe *normale* en contraignant son type à *porte_à_poignée* et sa cardinalité à 4.

Pour les composants multi-valués, TROPES permet, de plus, un type d'affinement particulier : l'**éclatement** de l'attribut. En fait, dans plusieurs situations, un composant multi-valué décrit plusieurs composants simples que l'on n'a pas envie de séparer ; lorsque l'on descend dans la hiérarchie on peut vouloir les décrire séparément ou dans des sous-groupes ayant des restrictions différentes. Par exemple, supposons que pour les voitures classiques on veuille spécifier qu'elles ont deux roues avant petites et deux roues arrière grandes. Une première façon de décrire cet affinement est l'éclatement de la description de l'attribut. Ainsi, pour l'exemple, la description de *roues* dans *automobile* :

```
roues  ens_de  roues
```

est affinée dans la classe *classique* par :

```
roues  ens_de  roues_avant_petites  |  roues_arri re_grandes
       card 2  | 2
```

Une deuxième façon de décrire l'affinement est l'éclatement de l'attribut même (dans l'exemple on éclate *roues* en *roues_avant* et *roues_arrière*) ainsi :

```
roues
roues_avant      ens_de  roues_avant_petites
                 card 2
roues_arri re    ens_de  roues_arri re_grandes
                 card 2
```

Chacune de ces options offre des avantages. La première fournit une vision immédiate des composants de l'objet tandis que la deuxième facilite la syntaxe des éclatements ultérieurs. TROPES offre les deux options.

Décomposition multiple

Un point de vue d'un concept ayant des attributs de nature composants décrit une décomposition des objets du concept. Chaque décomposition dans TROPES a un rapport précis avec les autres objets du système, en particulier avec les objets de la base accessibles du point de vue de la décomposition, les propriétés de ce point de vue et la structure de classes correspondante.

Plusieurs points de vue d'un concept peuvent inclure des attributs de nature composant. Les objets de ce concept ont alors des décompositions multiples. Lorsque l'on traite du problème de la diffusion de valeurs entre l'objet composite et les composants, il suffit de considérer la diffusion entre les propriétés accessibles d'un point de vue et les composants pertinents dans le même point de vue. Dans la version actuelle de TROPES on permet seulement la diffusion de valeur de l'objet composite vers les parties ; dans l'exemple, la carrosserie peut partager la couleur et les dimensions de la voiture.

Dans l'exemple (Fig. 5.26) la décomposition d'une voiture en carrosserie et roues est une décomposition faite du point de vue physique ; de ce même point de vue certaines propriétés de la voiture sont accessibles : sa couleur, ses dimensions, son poids, etc. Ce sont les seules propriétés de la voiture qui peuvent apparaître aussi dans la description de la carrosserie et des roues, celles des autres points de vue n'ayant aucune pertinence pour elles (c'est le cas de l'attribut permis_de_conduire du point de vue utilisation).

De même, la hiérarchie de classes d'un point de vue est rarement indépendante des composants pertinents dans ce point de vue. Dans l'exemple, les classes *économique* et *de_luxe*, sous-classes de *automobile* du point de vue *physique*, sont en rapport direct avec les classes *carrosseries_normale* et *carrosserie_de_luxe* car une voiture de luxe a une carrosserie de luxe et une voiture normale a une carrosserie normale. ce rapport n'existe pas, par exemple, entre les voitures à "propulsion arrière" et les "carrosseries de luxe".

5.8.3. Limitations de la relation de composition dans TROPES

La version actuelle du modèle TROPES établit certaines restrictions visant à éviter (1) des descriptions récursives d'objets et (2) des ambiguïtés lors de l'éclatement d'un attribut. Ces restrictions sont :

1 Un concept composite ne peut avoir dans aucun de ses arbres de décomposition un attribut prenant ses valeurs dans ce même concept. Dans les termes de Nebel [NEB91] un concept ne peut avoir une description à restrictions circulaires. Cette restriction empêche la définition d'instances infinies. Elle est aussi valide pour les concepts composés: un concept composé ne peut apparaître parmi des concepts que référencent ses propres attributs (directement ou indirectement). Cette restriction est très forte ; en effet, si un concept composite ne peut pas être récursive, une de ses classes ne peut pas l'être non plus et encore moins une instance. Il serait intéressant de permettre la récursion des concepts composites et de ses classes en interdisant la récursion au niveau des instances, qui pose des graves problèmes pour la classification.

2 L'ordre des éléments de la valeur d'un attribut multi-valué de nature composant est pertinent. Cette interprétation d'une multi-valeur comme une liste ordonnée permet, lors de la classification d'objets composites, de faire "descendre" une instance d'une classe C vers une de ses sous-classes C' lorsque celle-ci affine un composant de C par éclatement (§ 7.3.1). Ainsi, dans l'exemple des voitures, si une instance *I* d'*automobile* a pour l'attribut *roues* la multi-valeur *{R1, R2, R3, R4}*, alors vérifier si *I* est une *automobile classique* consiste a vérifier si *R1* et *R2* (les deux premières éléments de la liste) sont des *roues_avant_petites* et *R3* et *R4* sont des *roues_arrières_grandes.*

Conclusion

TROPES est un modèle de connaissances à objets multi-points de vue fondé sur les notions de classe et d'instance. Il permet la description, selon différents points de vue, d'un concept ou famille d'objets.

Une base de connaissances TROPES est structurée en familles d'objets de natures différentes, appelés **concepts** qui établissent une partition de l'univers de discours. Dans TROPES chaque concept peut être vu selon différents **points de vue**. Un point de vue détermine un groupe d'attributs pertinents du concept et une structuration de la connaissance dans un arbre de **classes** induit par la relation de spécialisation **sorte-de**. Les points de vue permettent ainsi de représenter le caractère multidisciplinaire d'une base de connaissances (plusieurs experts regardant la même famille d'objets). Les différents points de vue d'un concept peuvent être liés par des passerelles : une **passerelle** indique une relation d'inclusion ou l'égalité ensembliste entre des classes de différents points de vue ; les passerelles établissent une communication entre les divers points de vue d'un concept. Enfin, une **instance** d'une base TROPES représente un individu particulier d'un concept, qui est rattaché, dans chaque point de vue du concept, à sa classe d'appartenance la plus spécialisée, par le lien d'appartenance **est-un**. TROPES permet la représentation et la manipulation d'**objets composites** : un objet composite est une instance d'un concept ayant des attributs de nature composant. Une instance composite peut être décrite, dans les différents points de vue de son concept, par des décompositions différentes.

Le modèle TROPES a été conçu avec deux principes : simplicité et organisation. Un modèle de connaissances doit refléter la variété, multi-disciplinarité et complexité des éléments du monde mais sous une base simple, n'incluant que des notions nettement

définies de représentation. Ainsi, la notion de concept est considérée dans toutes ses dimensions : au niveau de la base comme le critère le plus global de partition de la base ; au niveau des objets comme l'espace de définition de leur structure et identification et au niveau du raisonnement comme l'espace de travail des différents mécanismes d'inférence. De même, chaque point de vue d'un concept détermine aussi bien un groupe d'attributs du concept pertinent à ce point de vue, qu'un arbre indépendant de spécialisation de classes. Les passerelles établissent des communications entre points de vue, ponctuelles et formellement définies. Enfin, pour les objets composites, le modèle manipule une sémantique unique de la relation de composition, garantissant ainsi la validité des inférences.

Bibliographie

[CAP&93] CAPPONI C., CHAILLOT M., *Construction incrémentale d'une base de classes correcte du point de vue des types,* Actes Journée Acquisition-Validation-Apprentissage, Saint-Raphaël, 1993.

[CAR85] CARDELLI L., *Typechecking Dependent Types and Subtypes*, in Foundations of Logic and Functional Programming Workshop, LNCS, vol. 306, Springer-Verlag, pp.44-57, 1985.

[CAR&85] CARDELLI L., WEGNER P., *On Understanding Types, Data Abstraction and Polymorphism,* ACM Computing Surveys, vol. 17, n°. 4, pp.471-522, décembre 1985.

[CAR&91] CARDELLI L., MITCHELL J., *Operations on records*, Mathematical Structures in Computer Science, vol. 1, n°. 1, pp.3-48, 1991.

[MAR89] MARIÑO O., *Classification d'objets dans un modèle multi-points-de-vue*, Rapport DEA d'informatique INPG, Grenoble, 1989.

[MAR&90] MARIÑO O., RECHENMANN F., UVIETTA P. *Multiple perspectives and classification mechanism in object-oriented representation* , in Proceedings of the 9th ECAI, Stockholm, pp.425-430, 1990.

[MAR91] MARIÑO O., *Classification d'objets composites dans un système de représentation de connaissances multi-points de vue,* RFIA'91, Lyon-Villeurbanne, pp. 233-242, 1991.

[NEB91] NEBEL B., *Terminological cycles : Semantics and computational Properties,* in Principles of Semantic Networks. Explorations in the Representation of Knowledge. J.Sowa (éd.), Morgan Kaufman Publishing, chapitre.11, pp.331-361, 1991.

[ORS90] ORSIER B., *Evolution de l'attachement procédural : intégration de tâches, méthodes et procédures dans une représentation de connaissances par objets*, Rapport DEA d'informatique, INPG, Grenoble, 1990.

[PAT&91] PATEL-SCHNEIDER P., McGUINESS D., BRACHMAN R., RESNICK L., BORGIDA A., *The CLASSIC Knowledge Representation System : Guiding Principles and Implementation Rational,* SIGART Bulletin, vol.2, n°. 3, pp.108-113, 1991.

[REC92] RECHENMANN F., ROUSSEAU B., *A development shell for knowledge-based systems in scientific computing,* in Expert Systems for Numerical Computing, Houstis E.N., Rice J.R. (éd.), Elsevier Science Publishers, Amsterdam, 1992.

[WEG87] WEGNER,P.,*The Object-Oriented Classification Paradigm,* dans Research Directions in Object-Oriented Programming, Bruce Shiver, Peter Wegner (éd.), The MIT Press, Cambridge, MA, 1987.

Chapitre 6

Classification multi-points de vue dans TROPES

Chapitre 6

Classification multi-points de vue dans TROPES

Introduction

Le mécanisme de raisonnement principal de TROPES est la classification d'instances. Étant donnée une instance d'un concept d'une base de connaissances TROPES, la classification consiste à trouver, dans chaque point de vue du concept, la classe la plus spécialisée pour laquelle l'instance satisfait les contraintes. Cette classification multi-points de vue tire parti des passerelles liant des points de vue, pour optimiser la descente de l'instance. De plus, la structure multi-points de vue des concepts permet de faire une classification partielle d'une instance en ne considérant qu'un sous-ensemble des points de vue et donc d'attributs du concept.

Dans ce chapitre nous présentons la classification multi-points de vue de TROPES. Nous commençons par décrire globalement la classification d'instances dans TROPES et les différentes notions et éléments qui interviennent. Le cœur de l'algorithme de classification est une boucle qui fait descendre[1] l'instance d'un niveau dans un des points de vue du concept. Cette boucle comporte cinq parties : obtention de l'information, appariement, propagation des marques, mise à jour de l'information et choix du prochain point de vue. L'idée générale de l'algorithme est donnée dans la première section. La deuxième section présente les éléments et paramètres de la classification ainsi que les relations qui existent entre ces éléments et qui restent invariants tout au long de la classification. Dans la troisième section du chapitre nous décrivons chacune des parties de la boucle de classification ainsi que les instructions d'initialisation et les conditions de terminaison de la classification[2].

6.1. Description générale

On peut comprendre la classification d'un objet comme sa localisation dans une structure d'objets connus (la base de connaissances), organisés selon une relation d'ordre (§ 4). Localiser le nouvel objet dans cette structure revient à rendre explicites des relations existantes entre cet objet et les objets de la base.

---

[1] Par la suite, nous allons utiliser le terme "descendre" en sens figuratif pour indiquer le fait que l'on cherche des minimaux de l'arbre de classes.

[2] La parallélisation de l'algorithme de classification est présentée dans [QUI93].

Dans ce cadre, la classification d'une instance dans une représentation à objets ayant l'approche classe / instance consiste à trouver les classes d'appartenance[1] les plus spécialisées pour une nouvelle instance dans un graphe de spécialisation de classes. Lorsque l'information de l'instance est complète, la classification peut retrouver toutes les classes de la base de connaissances auquelles appartient l'instance. La classification d'une instance complète peut être vue alors comme la division des classes de la base de connaissance en deux groupes : les classes **sûres** auxquelles l'instance appartient et les classes **impossibles** auxquelles elle n'appartient pas. Les éléments minimaux du premier groupe forment le résultat de la classification. Si le système manipule des instances incomplètes, la classification d'une instance peut rendre, outre les classes sûres et les classes impossibles, un ensemble de classes **possibles** : une classe possible est une classe pour laquelle le système ne peut ni affirmer ni nier l'appartenance de l'instance à partir de l'information dont il dispose (§ 4.3.1.).

La classification d'instances dans TROPES est une classification **multi-points de vue** qui a comme espace de travail un concept de la base de connaissances. Le but de la classification multi-points de vue est de trouver, pour une instance donnée, ses classes d'appartenance les plus spécialisées dans les différents points de vue de son concept et de compléter si possible l'information de l'instance. À la fin de la classification, les classes des différents points de vue sont marquées par une étiquette sûre ou possible ou impossible [MAR89], [MAR&90].

La classification multi-points de vue de TROPES peut être décrite comme le passage d'un **état initial** (de l'instance et du concept) dans lequel on connait une partie de l'information de l'instance (au moins le concept auquel elle appartient et sa clé) et certaines de ses classes d'appartenance dans le concept (certaines classes peuvent être déjà marquées comme étant des classes sûres pour l'instance; dans le pire des cas seules les racines sont marquées), à un **état final** dans lequel on a récupéré le maximum d'information de l'instance et celle-ci est classée le plus bas possible dans les différents points de vue (toutes les classes du concept sont marquées sûre ou possible ou impossible en fonction de l'information disponible) (Fig. 6.1).

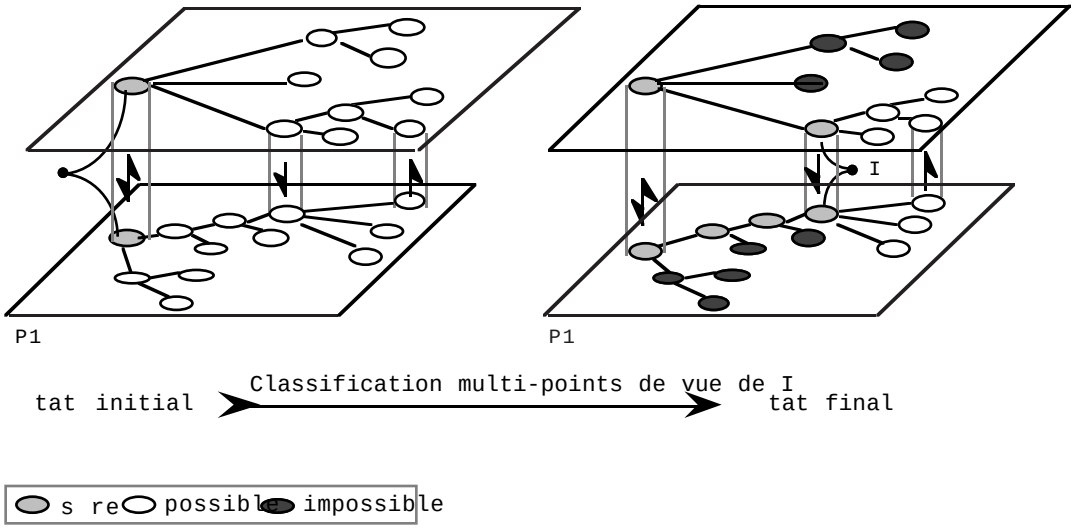


Fig. 6.1. Classification multi-points de vue dans TROPES. L'information minimale de l'instance (son concept et sa clé) permet de la rattacher aux racines des points de vue; à partir de cet état initial, la classification la fait descendre le plus bas possible dans tous les points de vue.

[1] Tout au long de ce chapitre l'affirmation : *L'instance I "appartient" à la classe C* signifie que I satisfait les contraintes de C et qu'elle est un membre de l'ensemble potentiel d'instances de C. Il faut noter que cela n'implique pas que I soit physiquement attachée à C, le rattachement étant une décision de l'utilisateur.

L'algorithme de classification cherche les classes d'appartenance les plus spécialisées pour l'instance dans les différents arbres de classification des différents points de vue de son concept. À la différence des algorithmes de classification d'instances présentés auparavant (§ 4) on ne traite pas ici d'un graphe de classification mais d'une fôret d'arbres de classification (un système de classification); le parcours de ce système de classification se fait par deux types de "descente" : la descente classique de l'instance vers une des sous-classes de sa classe d'appartenance courante et la descente vers une classe d'un autre point de vue par une passerelle.

Avant d'entrer dans les détails de l'algorithme de classification (§ 6.2), nous allons présenter l'idée générale avec un exemple. Supposons que dans une base de connaissances numériques on ait le concept "Matrice", vu des quatre points de vue : "Structure", "Forme", "Contenu" et "Remplissage" [ROU88]. Outre les passerelles triviales entre les racines des points de vue, il y a une passerelle de la classe "Matrice_diagonale/ Structure" vers la classe "Symétrique / Forme" pour indiquer que toute matrice diagonale est symétrique, et une autre passerelle entre la classe "Définie_positive / Contenu" et la classe "Matrice_carrée/Forme" indiquant qu'une matrice définie positive est toujours carrée (Fig. 6.2)

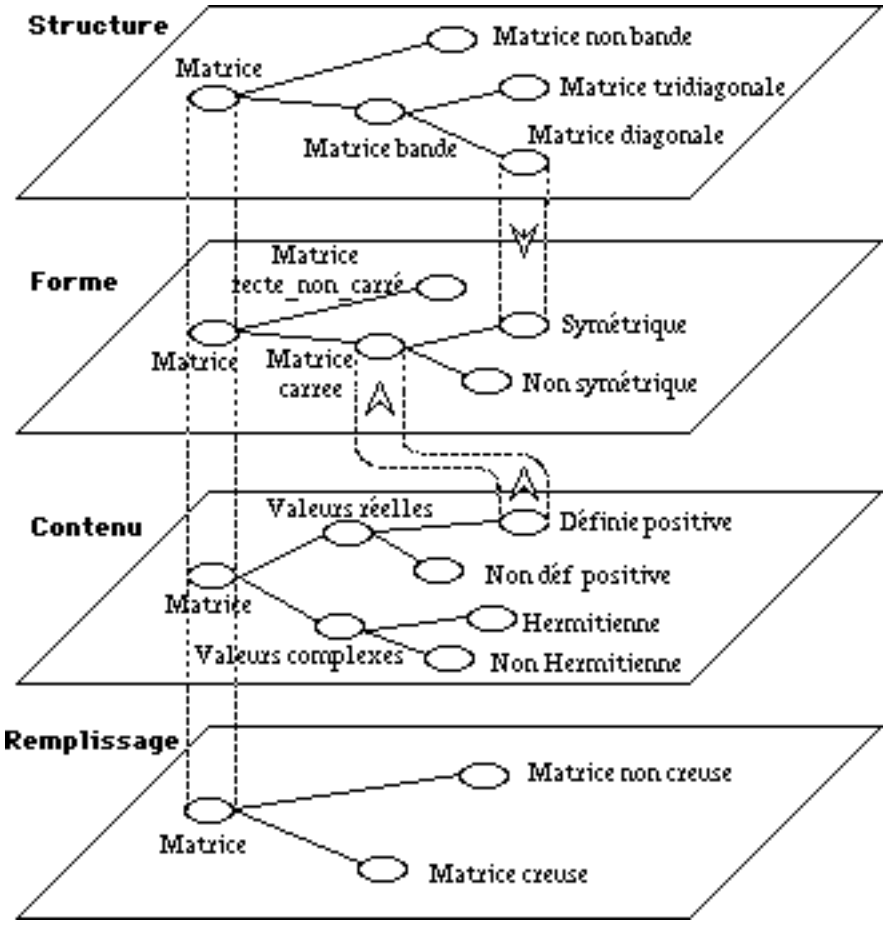


Fig. 6.2. Le concept "matrice" est décrit en TROPES par 4 points de vue, ayant, outre les passerelles entre les classes racines, deux passerelles unidirectionnelles.

Supposons maintenant que l'on veuille classer la matrice M (Fig. 6.3) dans le concept "Matrice", c'est-à-dire trouver la classe d'appartenance la plus spécialisée pour M dans chaque point de vue.

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

Fig. 6.3. Instance M du concept matrice à classer.

Pour l'état initial, on dispose déjà de l'information minimale : le fait que M est une matrice, et son identification, sa clé permettant au système de la distinguer des autres (dans ce cas son nom : M). Cette information permet de rattacher l'instance à toutes les classes racines du concept (Fig. 6.4).

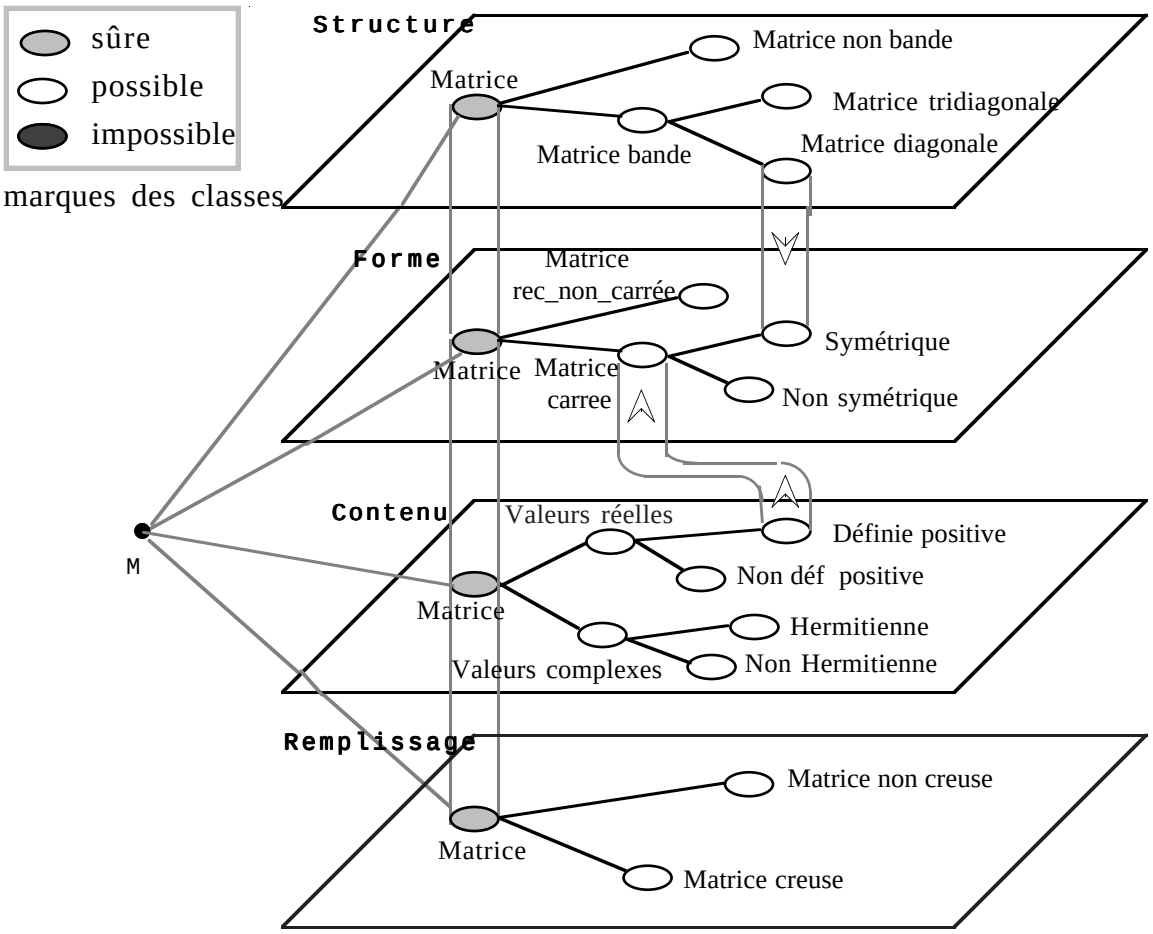


Fig. 6.4. Les racines des points de vue représentent l'univers du concept. Donc, toute instance du concept est élément de ces classes : ce sont des classes **sûres** pour M.

La classification multi-points de vue fait descendre l'instance pas à pas dans les points de vue. À chaque pas (de la boucle de classification) l'instance est descendue d'un niveau dans l'un des points de vue du concept. Cette descente appelle une procédure d'appariement qui décide vers quelle sous-classe C de la classe actuelle (plus petite classe sûre) du point de vue, doit descendre l'instance. Après l'appariement, la classe C est marquée comme étant sûre. Puis, comme par l'hypothèse des classes sœurs représente des ensembles exclusifs d'instances, les classes sœurs de C ainsi que leurs sous-classes sont marquées comme étant impossibles. Ces nouvelles marques sont propagées par les passerelles vers d'autres points de vue. Ainsi, par exemple, si du point de vue "structure" l'instance M est reconnue comme étant une matrice à bandes, puis une matrice diagonale, alors les classes Matrice_non_bande et Matrice_tridiagonale sont

marquées “impossible” et la classe Symétrique du point de vue Forme est marquée “sûre”. La classe Symétrique étant “sûre”, ses sur-classes deviennent elles aussi sûres, leurs sœurs deviennent “impossible”. Cette boucle de classification continue jusqu’à ce que l’instance soit complètement classée par rapport à l’information disponible. (Fig. 6.5). Après chaque pas de la boucle, le système choisit le point de vue sur lequel il va continuer la classification.

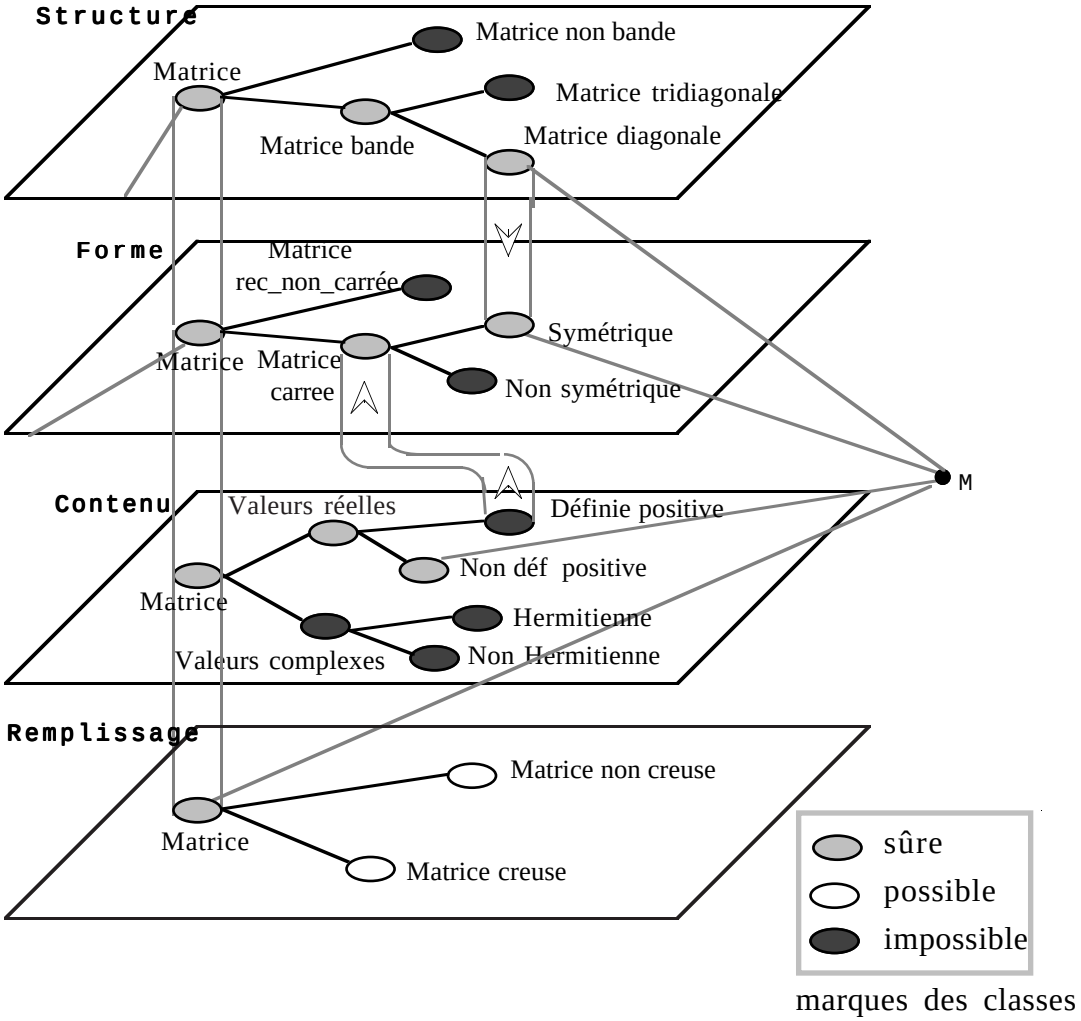


Fig. 6.5. À la fin de la classification, M est descendue le plus bas possible dans les différents points de vue (le système a pu vérifier qu’elle est une matrice diagonale, symétrique et non définie positive). De plus, toutes les classes sont marquées sûre, possible ou impossible.

6.2. Eléments de la classification

Comme nous l’avons montré dans l’exemple précédent, le cœur de l’algorithme de classification multi-points de vue est une boucle qui termine quand l’instance ne peut plus être descendue, soit parce qu’elle est déjà classée dans ses classes d’appartenance les plus spécialisées de chaque point de vue (toutes les classes sont marquées sûre ou impossible), soit parce que le système ne possède pas l’information permettant de continuer la descente dans certains points de vue (dans ces point de vue, il y aura alors des classes marquées comme “possibles”; c’est le cas du point de vue “remplissage” de l’exemple (Fig 6.5.)).

La boucle de classification fait descendre l'instance pas-à-pas dans les points de vue du concept. Un "pas" consiste à passer d'un état dont l'information est cohérente, un **état stable,** à un nouvel état stable "meilleur" (plus proche du but). Ce nouvel état stable est "meilleur" au sens où l'instance est descendue dans l'un des points de vue et qu'elle est plus bas par rapport à l'état précédent[1].

La descente de l'instance dans les différents points de vue fait évoluer les marques des classes et l'état des points de vue, en même temps qu'elle enrichit l'information de l'instance. Par la suite, nous allons définir les différents types et états des points de vue dans une classification, les différentes sortes de marques de leurs classes lors de la classification et l'information que la classification a, à tout moment, de l'instance à classer. Ces éléments vont permettre de spécifier plus rigoureusement la classification et de définir les notions d'invariant de la boucle de classification et d'état stable.

6.2.1. L'utilisateur et les points de vue

La puissance d'un modèle multi-points de vue pour la procédure de classification vient du fait qu'un utilisateur expert dans un domaine peut lancer la classification d'une instance d'un concept sur un sous-ensemble des points de vue du concept : les points de vue propres à son domaine d'expertise, les autres points de vue lui servant à inférer des connaissances additionnelles à travers les passerelles. Ainsi, avant de lancer la classification d'une instance d'un concept, l'utilisateur peut grouper les points de vue du concept en trois ensembles : les points de vue **principaux** de son domaine d'expertise, les points de vue **auxiliaires** dont il a une connaissance partielle et les points de vue **cachés** pour lesquels il n'a pas de connaissances[2].

La classification multi-points de vue de TROPES est semi-automatique : le programme pose des questions à l'utilisateur à chaque fois qu'il ne peut plus inférer d'information et qu'il ne peut pas avancer avec l'information dont il dispose. Ce dialogue commence par des questions sur les attributs des points de vue "principaux", afin de pouvoir faire descendre l'instance dans ces points de vue. Si l'information des attributs des points de vue principaux ne suffit pas à classer complètement l'instance, le programme demande à l'utilisateur des valeurs des attributs des points de vue "auxiliaires" (il ne demande jamais des informations propres aux points de vue cachés). Ainsi, on a :

Les points de vue principaux

Les points de vue principaux sont les points de vue dans lesquels l'utilisateur veut faire la classification et qu'il connaît le mieux. Ce sont des points de vue prioritaires au sens où le système va essayer à tout moment de continuer la descente de l'instance dans l'un d'entre eux. Ces points de vue seront notés [PV_{p1} , … , PV_{pn}].

Les points de vue auxiliaires

Les points de vue auxiliaires sont des points de vue moins connus par l'utilisateur. Lorsque la descente de l'instance dans les points de vue principaux s'avère bloquée, le système essaie de descendre l'instance dans l'un des points de vue auxiliaires, pour reprendre, éventuellement, par une passerelle si elle existe, l'un des points de vue principaux pour suivre la classification Étant donné que la classification dans un point de vue auxiliaire n'a pas d'intérêt en soi, l'algorithme de classification rend "inactifs" les

---

[1] Ce passage d'un état stable à un autre état stable où l'une des classes d'un point de vue est plus spécialisée garantit la terminaison de la boucle de classification (§6.3). La notion d'état stable est définie dans (§6.2.5).

[2] Si l'utilisateur ne spécifie pas ces trois ensembles, le système traite tous les points de vue comme étant des points de vue "principaux".

points de vue auxiliaires qui n'ont plus de passerelles actives (§ 6.2.4.). Ces points de vue seront notés [PV_{a1} , … , PV_{ak}].

Les points de vue cachés

Les points de vue cachés sont les arbres des domaines complètement inconnus pour l'utilisateur. Le système ne pose pas des questions à l'utilisateur sur les attributs propres à l'un de ces points de vue. Néanmoins, s'il existe des passerelles entre les points de vue "principaux" ou "auxiliaires" et les points de vue "cachés", le système peut utiliser ces points de vue comme "ponts" entre les différents points de vue pour propager des marques des classes. Ces points de vue seront notés [PV_{c1} , … , PV_{cm}].

6.2.2. L'instance et son type

Toutes les instances d'un même concept TROPES ont une même structure. Cette structure comporte deux parties : la partie **appartenance** qui décrit les classes d'appartenance les plus spécialisées de l'instance dans chaque point de vue (cette partie peut être vue comme un tuple de dimension n, n étant le nombre des points de vue du concept) et la partie **valeur** qui est un tuple de valeurs : une valeur pour chaque attribut du concept[1]. À partir de la partie appartenance de l'instance, le système peut déduire à tout moment son type (§ 5.7.3).

L'instance I à classer est distinguée par une clé qui permet de la reconnaître de façon unique dans tous les points de vue du concept. Au fur et à mesure que l'algorithme de classification fait descendre I dans les différents points de vue, la connaissance sur I s'enrichit. Cet enrichissement peut entraîner soit des modifications dans sa partie **appartenance** (lorsque l'instance est descendue dans l'un des points de vue), soit des changement dans sa partie valeur (lorsqu'un attribut inconnu reçoit une valeur).

La classification sur plusieurs points de vue entraîne une vision "parallèle" de la descente de l'instance dans les différents points de vue. En particulier, lors de l'utilisation d'une passerelle, plusieurs points de vue peuvent être modifiés "simultanément". Or, l'utilisateur ne voit qu'un seul point de vue à la fois, le point de vue courant. Donc pour ne pas le surcharger d'informations sur les autres points de vue, le système stocke dans une structure additionnelle, le **I-moule**, toutes les modifications ne concernant pas le point de vue courant. Ces modifications sont intégrées à l'instance et montrées à l'utilisateur dès que leurs points de vue deviennent le point de vue courant. Le I-moule comporte la liste des classes les plus spécialisées de l'instance et le type intersection des types de ces classes (le type de l'instance) (Fig. 6.6). Dans cette liste de classes il y a des classes qui n'ont pas encore été intégrées à l'instance et que nous allons appeler les **classes ouvertes** (§ 6.2.3)) .

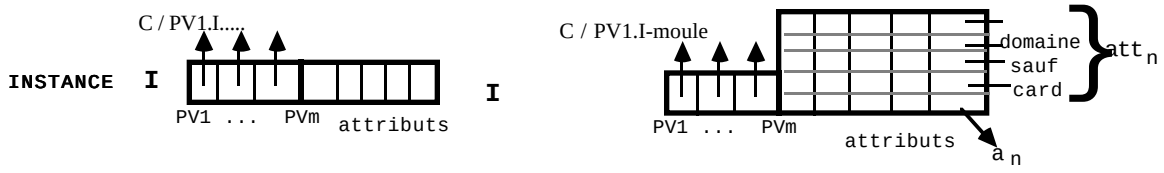


Fig. 6.6. L'instance à classer I et l'information additionnelle de son type et ses dernières classes d'appartenance calculées, le I-moule.

6.2.3. Les marques et statuts des classes

[1] Un attribut inconnu a la valeur inconnue, notée '?'.

Comme dans le système SHIRKA (§ 4.5.1), la classification dans TROPES marque les classes avec une des marques **sûre, possible** ou **impossible.** Une classe est **impossible** pour une instance I, si l'information de I est en contradiction avec les contraintes imposées par la classe, c'est-à-dire si I ne satisfait pas le type de la classe. Une classe est **possible** pour l'instance I si l'information connue de I n'est pas en contradiction avec le type de la classe, et qu'il manque des informations pour assurer l'appartenance de l'instance à la classe. Toute classe non marquée est supposée possible. Enfin, une classe est **sûre** pour une instance I si I satisfait son type.

Les classes sûres d'un point de vue peuvent être ouvertes ou fermées. De plus, elles peuvent être terminales ou non terminales.

Une classe sûre ouverte est une classe sûre dont les sous-classes n'ont pas encore été explorées[1] (en principe, une telle sous-classe n'a pas de marque, sauf si elle est une passerelle marquée impossible dans un autre point de vue). Le sous-arbre de classes non marquées dont la racine est la classe sûre ouverte est appelé **arbre actif** (Fig. 6.7)**.** Tout point de vue a au plus une classe sûre ouverte : la plus petite classe sûre de l'instance à un moment donné (Fig. 6.7).

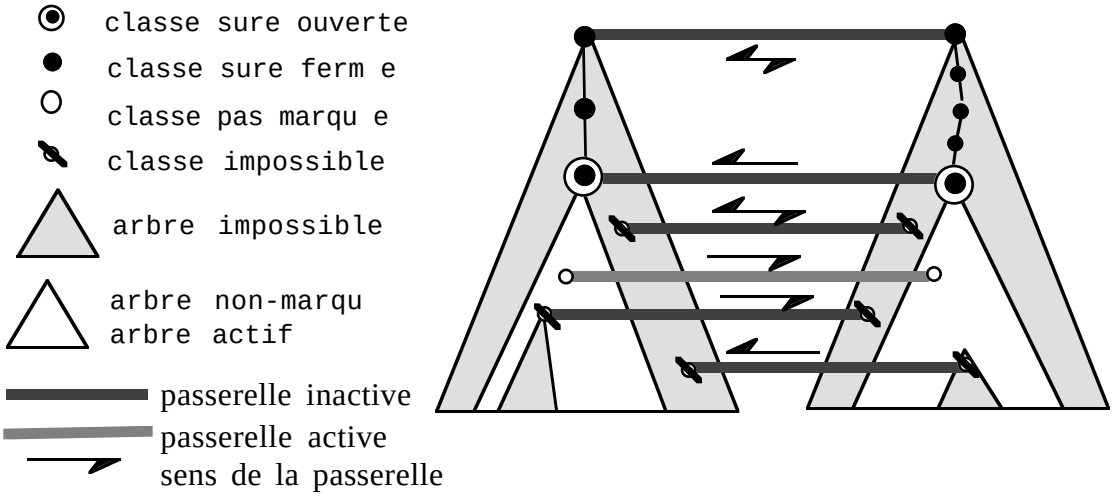


Fig. 6.7. Dans un point de vue, toutes les sur-classes de la classe la plus spécialisée de l'instance sont des classes sûres fermées. La classe la plus spécialisée peut être sûre fermée, si l'instance y est déjà descendue ou sûre ouverte sinon. Les classes sœurs de toutes les classes sûres sont des classes impossibles. Une passerelle inactive est une passerelle dont les classes extrémités sont déjà marqués sûrs ou impossibles; sinon elle est active. Un arbre actif est un arbre non marqué.

Une classe sûre fermée est une classe sûre dont les sous-classes ont déjà été explorées ; l'information de l'instance pour cette classe a été déjà complétée. Au fur et à mesure que la classification fait descendre l'instance dans un point de vue, les classes du chemin suivi par l'instance dans ce point de vue deviennent des classes sûres. Lorsque la descente continue, ces classes sont fermées au sens où on ne va plus les interroger, ni les examiner dans la classification. Toutes les sur-classes d'une classe sûre fermée sont des classes sûres fermées. Dans un état stable les sous-classes directes d'une classe sûre fermée sont toutes marquées (Fig. 6.7).

Une classe sûre terminale (Fig. 6.8) est une classe sûre à partir de laquelle l'instance ne peut plus descendre :

[1] Si la marque sûre d'une classe sûre ouverte a été obtenue en descendant l'instance dans ce point de vue, alors l'instance a toute l'information concernant cette classe. Si la marque a été obtenue par instanciation de la classe ou par une passerelle, la classe peut avoir des informations qui ne sont pas encore intégrées dans l'instance

1 soit parce qu'elle n'a pas de sous-classes (elle est une feuille de l'arbre de ce point de vue),
2 soit parce que toutes ses sous-classes sont marquées comme des classes impossibles.

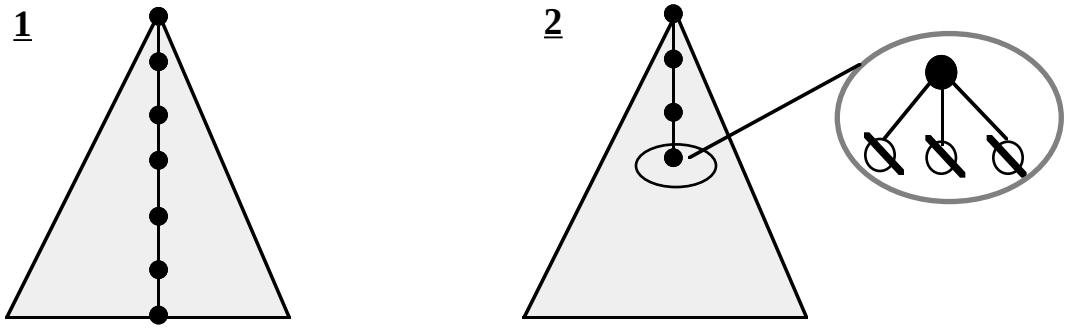


Fig. 6.8. La classe sûre terminale d'un point de vue est la plus petite classe sûre de l'instance dans ce point de vue.

6.2.4. Les statuts des points de vue

Le but de la classification étant de descendre l'instance le plus bas possible dans les points de vue principaux, les points de vue auxiliaires et cachés servent à la classification seulement s'ils permettent de déduire des informations utiles à cette descente. Par ailleurs, un point de vue principal duquel on a déjà tiré toute l'information possible, n'est plus utile à la classification non plus. Ainsi, à tout moment de la classification un point de vue peut être actif s'il sert encore au but de la classification, ou inactif sinon.

Point de vue actif

Un point de vue actif est un point de vue qui est encore utile pour la classification de l'instance. Tout point de vue principal est actif s'il a un arbre actif non vide, c'est-à-dire s'il a des classes non marquées (il n'a aucune classe marquée comme sûre terminale). Un point de vue auxiliaire ou caché est actif s'il a au moins une passerelle active : une passerelle active est une passerelle dont une des classes n'a pas encore été marquée (Fig. 6.7).

Point de vue inactif

Tout point de vue ayant une classe marquée comme sûre terminale est **inactif** (Fig. 6.8).Un point de vue auxiliaire ou caché est inactif s'il n'a pas de passerelles actives.

6.2.5. Invariants de la classification

L'invariant d'une boucle est un ensemble d'assertions sur l'état des éléments qui interviennent dans la boucle et qui doivent être valides avant chaque itération et à la fin. L'invariant de la classification peut être décrit, de façon informelle, par l'assertion "La classification se dérouler correctement", qui veut dire en terme des objets de la représentation "Les attributs de I valident les contraintes des classes auxquelles I est attachée et pour toute classe du concept, sa marque (sûre ou possible ou impossible) indique correctement la relation entre cette classe et l'instance I".

Par la suite, nous allons présenter les éléments d'un état de la classification et les relations qui doivent rester invariantes pendant toute la classification.

Etat de la classification

Un état de la classification est décrit par les éléments intervenant dans la boucle de classification :

Pour l'information générale :

° La liste ordonnée des points de vue principaux [PV_{p1} , … , PV_{pn}]

° La liste ordonnée des points de vue auxiliaires [PV_{a1} , … , PV_{ak}]

° La liste ordonnée des points de vue cachés [PV_{c1} , … , PV_{cm}]

° Le point de vue courant, sur lequel se fait la classification à ce moment : PV-actuel

° L'instance avec ses plus petites classes d'appartenance connues, et les valeurs de ses attributs.

Pour l'information concernant chaque point de vue PV_i :

° L'arbre de classes du point de vue avec ses marques : PV_i

° La plus petite classe sûre dans PV_i : C-min / PVi (pour le point de vue actuel elle sera appelée C-actuelle)

° L'état du point de vue (actif ou inactif) : état.PV_i

° Le nombre de passerelles encore actives dans ce point de vue : Npass.PV_i

Relations invariantes

Pour chaque point de vue PV_i :

° Les classes sûres forment un ordre total, dont la classe la plus grande est la racine C/PV_i et la classe la plus petite est C-min / PVi

° Toutes les classes sûres sont des classes fermées (sauf C-min/PV_i qui peut être ouverte ou fermée).

° Toutes les sous-classes d'une classe marquée impossible, sont marquées impossible.

Pour les passerelles :

° Si toutes les sources d'une passerelle sont marquées sûres, sa destination l'est aussi.

° Si la destination d'une passerelle est impossible, alors au moins une de ses sources n'est pas marquée sûre, car si toutes les sources étaient marquée sûre, alors la destination serait nécessairement sûre aussi..

° Toute classe marquée impossible, sous-classe d'une classe non-marquée, est la source d'une passerelle dont la destination est aussi marquée impossible. En effet, pour qu'une sous-classe d'une classe non marquée soit marquée, sa marque a dûe être déduite par une passerelle. Comme nous allons montrer plus loin (§ 6.3.5), si la destination d'une passerelle est impossible et toutes ses sources sauf une sont sûres, alors cette dernière est marquée impossible (et c'est la seule façon d'inférer une marque impossible par une passerelle).

Pour le point de vue actuel : PV-actuel

° C-actuelle est une classe sûre ouverte

° Le point de vue actuel est un point de vue auxiliaire seulement si la descente par les points de vue principaux est bloquée (la plus petite classe sûre pour ce point de vue est fermée) : PV-actuel = PVai $\Rightarrow$ $\forall$PVpj, C-min / PVpj est fermée

Pour l'instance

° I est attachée aux classes sûres fermées : $\forall$PVj, C / PVj.I est sûre fermée

° I satisfait les types de ces classes : $\forall$PVj, I est-un C / PVj.I

Pour le I-moule

° ∀PVj, C / PVj.I-moule est la plus petite classe sûre de PV_j c'est-à-dire :
∀PVj, C / PVj.I-moule = C-min /PV_j

° Les contraintes exprimées dans le I-moule pour chaque attribut a_x : att_x, sont l'union des contraintes de cet attribut pour toutes les classes sûres de I; ils décrivent l'intersection des types de cet attribut pour les différentes classes sûres.

Relations entre I et I-moule

° Les classes auxquelles le I-moule est attaché sont plus petites ou égales à celles de I
Le I-moule descend plus vite que l'instance parce que sa descente se fait dès qu'une classe sûre est identifiée alors que la descente de l'instance se fait seulement quand cette classe sûre est examinée :
∀PVj, C / PVj.I-moule ≤ C / PVj.I (Fig. 6.9)

° Si I n'est pas descendue à une classe sûre déjà reconnue par le I-moule, alors cette classe est une classe sûre ouverte : C / PVj.I-moule < C / PVj.I ⇒ C / PVj.I-moule est sûre ouverte

° I satisfait toutes les contraintes du I-moule :
∀ att_x .I-moule, v_x valide att_x c'est-à-dire : ∀PVj, I est-un C / PVj.I-moule

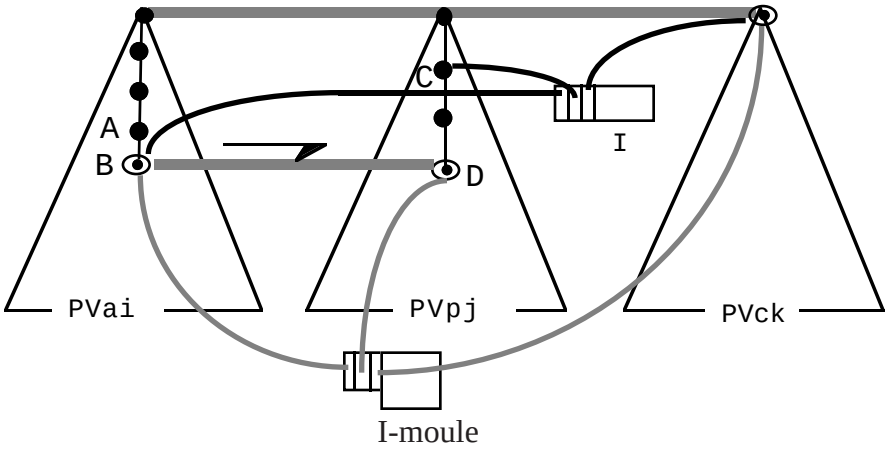


Fig. 6.9. Lorsque I descend de A à B dans le point de vue courant PVai, la marque sûre de B est propagée à D. Cette nouvelle classe sûre dans PVpj est mémorisée dans le I-moule. I reste attachée à C jusqu'à ce que PVpj devienne le point de vue courant.

Un état dont les éléments satisfont les relations invariantes est appelé un **état stable**. L'algorithme de classification va garantir qu'après chaque pas de la boucle de classification, la base de connaissances soit dans un état stable.

6.3. L'algorithme de classification

L'algorithme de classification repose sur une boucle ; l'instance commence dans un état initial stable et chaque pas de la boucle de classification passe la base de connaissances d'un état stable Ek à un état stable E_{k+1} plus proche du but (dans E_{k+1} l'instance est rattachée aux classes plus spécialisées que dans E_k). Le corps de la boucle comporte cinq pas: obtention d'information, appariement, propagation des marques, mise à jour de l'information et choix du prochain point de vue. L'algorithme termine (fini) quand l'instance est complètement classée par rapport à l'information disponible, c'est-à-dire quand tous les points de vue sont bloquées, ou bien si l'utilisateur décide de l'arrêter (Fig. 6.10).

```
construire_ tat_initial
```

```
tant_que not ( fini )
  obtenir_information
  faire_appariement
  propager_marques
  mettre_ _jour_l_information
  choisir_prochain_point_de_vue
fin_tant_que
```

Fig. 6.10. L'algorithme de classification est une boucle dont chaque itération fait descendre l'instance dans l'un des points de vue.

Construire l'état initial consiste à créer à partir des informations initiales sur l'instance et ses classes d'appartenance de départ, un état stable. Une fois l'état initial construit, la classification consiste à répéter la boucle de classification : obtention de l'information, appariement, propagation des marques, mise à jour de l'information et choix du prochain point de vue.

Par la procédure **obtenir_information,** le système obtient de l'utilisateur l'information manquante sur la classe sûre ouverte du point de vue courant (C-actuelle). Etant donné que cette classe est déjà une classe sûre pour I, l'utilisateur n'est autorisé à donner que des valeurs valides pour les attributs dans cette classe (et qui valident aussi les contraintes établies dans le I-moule), ou bien la valeur inconnue.

Une fois l'information de la classe actuelle complétée, l'algorithme cherche à faire descendre l'instance dans ce point de vue (vers une des sous-classes directes de C-actuelle). La procédure **faire_appariement** réalise une comparaison entre les valeurs des attributs de I et les contraintes des différentes sous-classes de C-actuelle. Pour faire cette comparaison, elle demande à l'utilisateur les valeurs des attributs de I dont elle a besoin. Comme résultat de la comparaison, l'appariement marque chacune des classes révisées avec une marque "sûre", "possible", ou "impossible".

Les nouvelles marques peuvent générer des incohérences au niveau des points de vue (vis-à-vis des relations invariantes entre les classes marquées). La procédure **propager_marques** rétablit cette cohérence en marquant comme "impossibles" les sous-classes des nouvelles classes "impossibles" et en propageant les marques "sûre" et "impossible" correctement par les passerelles. Enfin, cette étape met à jour l'état actif ou inactif des points de vue (en fonction du type de point de vue et du nombre de passerelles qui restent actives).

La fonction **mettre_à_jour_l_information** met à jour l'information de la plus petite classe sûre de chaque point de vue. Enfin, la procédure, **choix_prochain_point_de_vue**, choisit, en fonction d'un critère préétabli, le point de vue à prendre comme point de vue actuel pour la prochaine itération du cycle. C'est aussi dans cette partie que le système vérifie les différentes conditions de sortie du cycle (y compris la décision de l'utilisateur d'arrêter la classification).

6.3.1. Construction de l'état initial

Avant de lancer la classification, il faut donner des valeurs initiales aux éléments de façon à avoir un état initial stable.

Paramètres de départ

Avant commencer la classification, l'utilisateur doit indiquer la liste des points de vue principaux, celle des points de vue auxiliaires et celle des points de vue cachés (par défaut, tous les points de vue sont des points de vue principaux).

Information sur l'instance et le I-moule

Dans l'état initial de la classification, l'instance a seulement l'information initiale donnée par l'utilisateur : la liste des valeurs d'attributs v_{ai} et de classes d'appartenance connues par l'utilisateur dans les différents points de vue C_i / PV_i (si pour un point de vue l'utilisateur n'a pas d'information, la classe d'appartenance de l'instance est la classe racine du point de vue). Les classes d'appartenance de départ de l'instance sont des classes sûres ouvertes, c'est à dire des classes sûres sur lesquelles on peut encore poser des questions. Pour garder les assertions de l'invariant, le I-moule est rattaché à ces classes d'appartenance C_i / PV_i et l'instance est rattachée à leur sur-classes directes (dans le cas de la racine, l'instance n'est rattachée à aucune classe). De plus, le I-moule a comme type l'intersection des types des classes Ci / PVi. L'instance a dans son tuple de valeurs les valeurs initiales données par l'utilisateur.

Information sur chaque point de vue et sur les passerelles

Toutes les sur-classes de la classe initiale de rattachement Ci sont marquées comme sûres. Toutes les sœurs de la classe initiale sont marquées comme impossibles, car une instance ne peut appartenir qu'à une classe par niveau (par l'hypothèse d'exclusivité des classes sœurs). Si parmi les classes marquées, il y a des extrémités de passerelles, les marques sont propagées aux autres points de vue (par le mécanisme de propagation de marques présenté dans § 6.3.5). La classe Ci est marquée comme sûre et est reconnue comme la plus petite classe sûre du point de vue (C-min / PVi).

Si PVi est un point de vue principal, il est au début en état actif; pour les points de vue auxiliaires ou cachés, le système compte le nombre de passerelles actives Npass.PVi ; si Npass.PVi = 0, l'état de PVi est inactif, sinon il est actif. Comme point de vue courant (PV-actuel) le système prend au départ le premier point de vue de la liste des points de vue principaux, PVp1, Cp1 devient la classe courante, C-actuelle (Fig. 6.11).

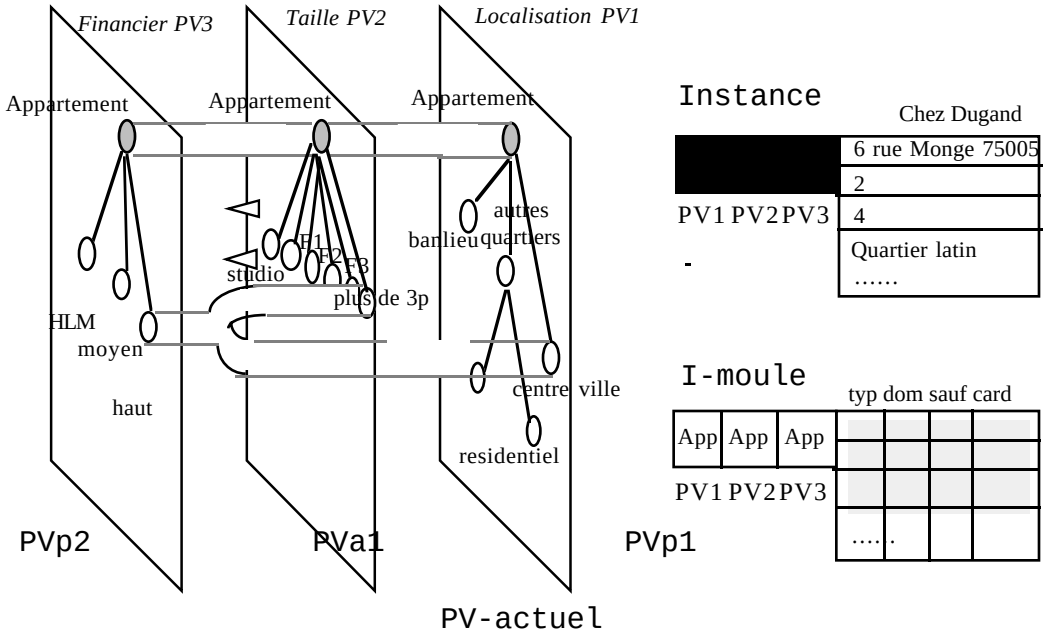


Fig. 6.11. Supposons par exemple, que l'on souhait classer l'instance *Chez Dugand* du concept Appartement pour laquelle on ne connait que la clé, et que les points de vue *Localisation* et *Financier* sont des points de vue principaux, et *Taille* est un point de vue auxiliaire. La classification commence par le point de vue *Localisation* et l'instance et le I-moule sont initialisés de façon à satisfaire l'invariant.

6.3.2. Condition de terminaison de la boucle de classification

Le programme peut terminer pour l'une des trois raisons suivantes :

- Le but est atteint : l'instance I est classée le plus bas possible dans chacun des points de vue principaux. Ceci est reconnu par le fait que tous les points de vue principaux sont "inactifs", leur plus petite classe sûre est fermée terminale.
- L'algorithme ne trouve plus de chemins à suivre. Ce cas arrive lorsque, même s'il existe encore des points de vue principaux actifs (avec des classes non marquées), il n'y a plus de classes sûres ouvertes dans aucun des points de vue.
- L'utilisateur veut arrêter la classification.

Si l'on groupe ces conditions dans un prédicat : **fini = C1 & C2 & C3**, alors le cycle continue tant que **fini** reste faux : tant_que **not (fini)**.

6.3.3. Obtention d'information

La procédure d'obtention d'information complète l'information de la classe actuelle du point de vue courant, C-actuelle, qui est une classe sûre ouverte. Pour compléter cette information soit le système fait des inférences de valeurs manquantes d'attributs de cette classe, soit il demande ces valeurs à l'utilisateur. Dans ce dernier cas, celui-ci doit donner des valeurs valides, c'est à dire des valeurs satisfaisant le type calculé pour cet attribut (stocké dans le Imoule). À la fin, l'instance est descendue vers C-actuelle ; C-actuelle est marquée comme étant sûre fermée (Fig. 6.12).

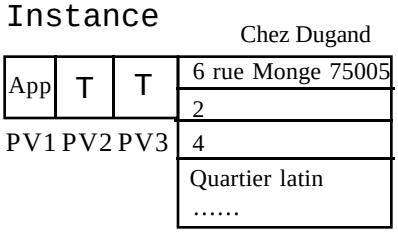


Fig. 6.12. Si, dans l'exemple de l'appartement, la classe *Appartement / Localisation* a, outre les attributs clés, l'attribut *quartier*, le système demande sa valeur à l'utilisateur, puis rattache l'instance à cette classe, qui devient sûre fermée.

6.3.4. Appariement

Le but de l'étape d'appariement est de trouver une classe sûre parmi les sous-classes de C_actuelle qui ne sont pas encore marquées[1] {S1, S2…, Sn} c'est-à-dire trouver une classe pour laquelle l'instance I satisfait toutes les contraintes, pour pouvoir descendre I d'un niveau dans l'arbre de classes de ce point de vue. Une classe Sj est une classe sûre pour I si I satisfait son type, c'est-à-dire si I a une valeur valide pour tous les attributs de la classe Sj ou si, pour tout attribut non valué dans l'instance on peut affirmer que lorsqu'elle aura une valeur, cette valeur sera valide. Ce deuxième cas est vrai pour un attribut si son type pour l'instance (calculé et stocké dans le I-moule) est un sous-type de son type pour la classe Sj.

La procédure d'appariement suit deux principes :

[1]Une des sous-classes a pu être marquée comme impossible par une passerelle : dans ce cas elle n'est évidement pas examinée.

1 Faire le maximum d'inférences avant de commencer à poser des questions à l'utilisateur.

2 Ordonner les questions à poser à l'utilisateur de façon à tirer le maximum d'information de chacune de ses réponses.

Réduire le nombre de classes à tester

Pour le premier principe, la procédure d'appariement essaye de réduire le nombre de classes à tester soit en identifiant dès le début la seule classe sûre de l'ensemble {S1,…, Sn}, **premier pas**, soit, si cela n'est pas possible, en identifiant les classes impossibles pour les enlever de l'ensemble à considérer, **deuxième pas**.

Premier pas

Nous avons dit auparavant que pour vérifier si une classe Sj est sûre pour I il faut, soit comparer la valeur soit le type de chaque attribut de I avec le type de cet attribut dans Sj. Comme la deuxième vérification peut être coûteuse, seule la première sera considérée dans ce premier pas. Ainsi, le premier pas de la procédure prend parmi l'ensemble {S1,…,Sn} les classes n'ayant que des attributs qui sont valués dans l'instance et pour chacun d'eux, elle teste si la valeur dans l'instance satisfait les contraintes. Si en faisant cette vérification, elle détermine une classe sûre, l'appariement s'arrête car par l'hypothèse d'exclusivité des classes sœurs, toutes les autres classes de ce niveau sont impossibles ; si aucune classes sûre n'est trouvée, l'appariement continue avec le deuxième pas.

Deuxième pas

De même que pour le premier pas, l'algorithme ne fait ici que des comparaisons entre les valeurs des attributs dans l'instance et leur type dans la classe. Il examine toutes les classes de {S1,…,Sn}. Pour chaque classe, il compare chacun des attributs de son schéma avec sa valeur dans l'instance. Si pour un de ces attributs la valeur dans I ne valide pas le type, la classe est marquée comme impossible.

Si, après ce deuxième pas, toutes les classes ont été marquées impossibles, alors l'appariement s'arrête (la classe actuelle est devenue une classe terminale et le point de de vue sera marqué inactif par la suite).

Ordonner les questions

Si après ces deux pas aucune classe de l'ensemble n'a été marquée sûre et qu'il demeure des classes sans marque, le système interroge l'utilisateur[1]. La réponse à chacune de ces questions peut être une valeur v ou la valeur inconnue, '?'.

Pour faciliter l'interaction avec l'utilisateur, la procédure ordonne les questions à poser à l'utilisateur de façon à optimiser les inférences que le système peut tirer des réponses données. L'idée générale de cet ordonnancement est d'examiner en premier les classes ayant le plus petit nombre d'attributs non valués. Ainsi, le système commence par poser des questions sur le premier attribut non valué de la plus petite classe selon l'ordre précédent.

Au fur et à mesure que l'utilisateur donne les valeurs des attributs (valides pour le type actuel de I), le système met à jour les marques des classes qui possèdent de ces attributs. Les questions s'arrêtent quand le système arrive à marquer une classe sûre ou quand il n'y a plus de classes non marquées dans l'ensemble ou encore quand, même s'il y a encore des classes non marquées, il n'y a plus de questions à poser (l'utilisateur n'a

[1]Le modèle prévoit l'utilisation de méthodes de calcul des valeurs des attributs attachées au schéma de l'attribut au niveau du concept (qui devraient se déclencher avant le recours à l'utilisateur) mais cette option n'est pas encore prise en compte par la classification.

pas su répondre à toutes les questions posées. Si aucune classe n'a été marquée comme sûre, les sous-classes non marquées sont marquées comme possibles (Fig. 6.13).

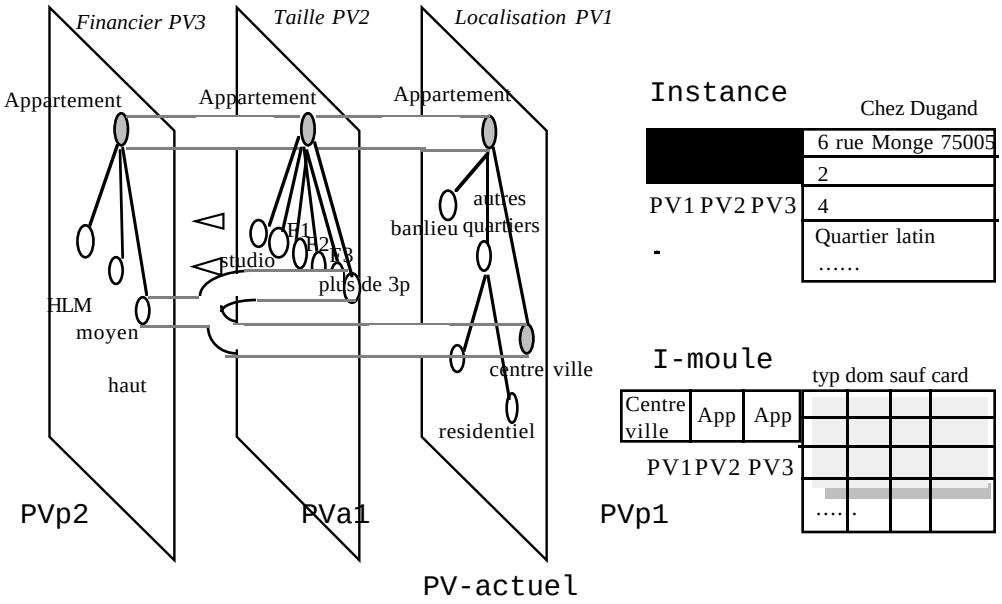


Fig. 6.13. Reprenons l'exemple de l'appartement. Les sous-classes de la classe courante sont les classes *Banlieue*, *Centre_Ville* et *Autre_Quartiers*. Si le seul attribut introduit dans ce niveau est l'attribut *Quartier*, qui a déjà une valeur dans l'instance, l'algorithme d'appariement identifie la classe sûre dès son premier pas. Supposons encore que le quartier latin fasse partie du centre de Paris (!), dans ce cas la procédure d'appariement marque comme sûre la classe Centre_ville, qui dévient la plus petite classe sûre du point de vue *Localisation* et qui est mémorisée dans le I-moule.

6.3.5. Propagation des marques

Les nouvelles marques inférées par l'appariement peuvent créer des inconsistances vis-à-vis des relations invariantes établies pour les points de vue. L'algorithme de propagation de marques rétablit l'état d'équilibre.

Règles de propagation de marques

L'algorithme gère la propagation des marques en fonction de six règles de cohérence déduites de la sémantique accordée aux classes, aux passerelles et aux marques. Les trois premières concernent les propagations à l'intérieur d'un point de vue et les trois dernières aux propagations par des passerelles :

1. Dans tout point de vue, les classes sœurs décrivent des ensembles mutuellement exclusifs. Ainsi, si une instance appartient à une de ces classes, elle ne peut pas appartenir à aucune autre classe (Fig. 6.15).

⇒ Propagation de la marque **impossible** aux classes sœurs d'une nouvelle classe sûre.

2. La spécialisation de classes dans TROPES corresponde l'inclusion ensembliste. Ainsi, si une instance appartient à une classe (à l'ensemble potentiel décrit par la classe), alors elle appartient à toutes ses sur-classes.

⇒ Propagation de la marque **sûre** à toutes les sur-classes d'une nouvelle classe sûre.

3. Par la correspondance entre l'inclusion ensembliste et la spécialisation, si une instance n'appartient pas à l'extension d'une classe, elle ne peut pas appartenir à l'extension d'aucune de ses sous-classes.

⇒ Propagation de la marque **impossible** aux sous-arbres du point de vue ayant comme racine une classe **impossible** (Fig. 6.15).

4. Une passerelle unidirectionnelle reflète aussi une inclusion ensembliste. Pour les passerelles à une seule source, si l'instance appartient à la classe source de la passerelle, alors elle appartient à la classe destination. Cette relation ensembliste peut se généraliser aux passerelles ayant plusieurs sources : si une instance appartient a TOUTES les classes sources, alors elle appartient à la classe destination.

⇒ Propagation de la marque **sure** à la classe destination d'une passerelle dont TOUTES les sources ont la marque sûre (Fig. 6.14).

5. Vu dans le sens inverse : la classe source d'une passerelle représente un sous-ensemble des instances de la classe destination de la passerelle. Ainsi, si l'instance n'est pas membre de l'ensemble potentiel d'instances de la classe destination, alors elle ne peut pas appartenir à la classe source, dont l'ensemble potentiel est un sous-ensemble de celui de la classe destination.

⇒ Propagation de la marque **impossible** à la classe source d'une passerelle ayant UNE seule classe source et dont la classe destination est marquée impossible (Fig. 6.14.).

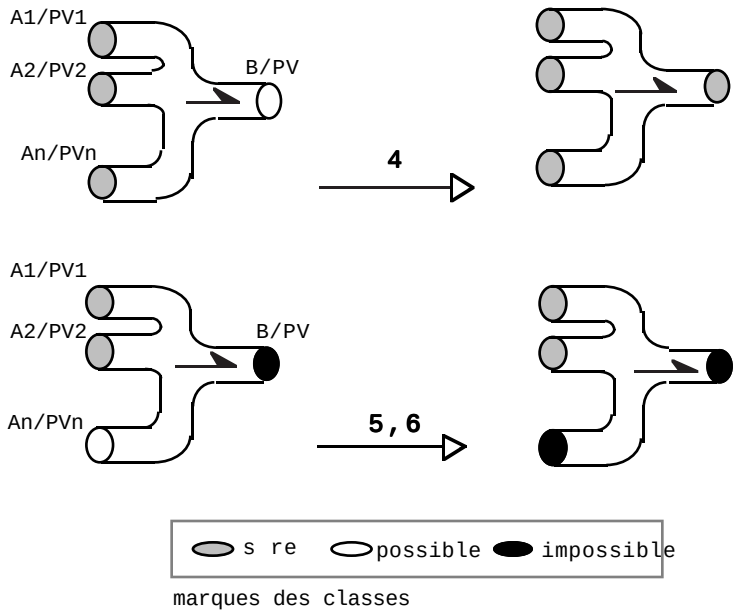


Fig. 6.14. Propagation de la marque sûre à la destination d'une passerelle dont les sources sont sûres (**4**). Propagation (dans les sens inverse) de la marque sûre à la seule source non marquée d'une passerelle dont les autres sources (s'il y en a) sont sûres et la destination est impossible (**5,6**).

6. Le cas précédent peut être généralisé aux passerelles ayant plusieurs sources.

   Soient A_1/PV_1, A_2/PV_2,…, A_x/PV_x des classes sources d'une passerelle et B/PV sa classe destination. Par la définition de passerelle : une instance I qui appartient a TOUTES les sources de la passerelle, appartient aussi à sa destination :

   $$x \in A_1/PV_1 \wedge x \in A_2/PV_2 \wedge \ldots \wedge x \in A_n/PV_n \Rightarrow x \in B/PV.$$

   La contraposée de cette formule logique est :

   $$\neg (x \in B/PV) \Rightarrow \neg (x \in A_1/PV_1 \wedge x \in A_2/PV_2 \wedge \ldots \wedge x \in A_n/PV_n)$$

   qui peut être lue comme : si x n'appartient pas a B/PV alors il ne peut pas appartenir à TOUTES les A_i/PV_i, i :1…n .

Autrement dit, si la destination de la passerelle est marquée impossible et toutes les sources sauf une sont marquées sûre, alors celle qui n'est pas marquée doit être nécessairement une classe impossible.

⇒ Propagation de la marque **impossible** à la seule classe source non_marquée d'une passerelle dont la destination est marquée impossible et toutes les autres classes sources sont marquées sûre (Fig. 6.14), (Fig. 6.15).

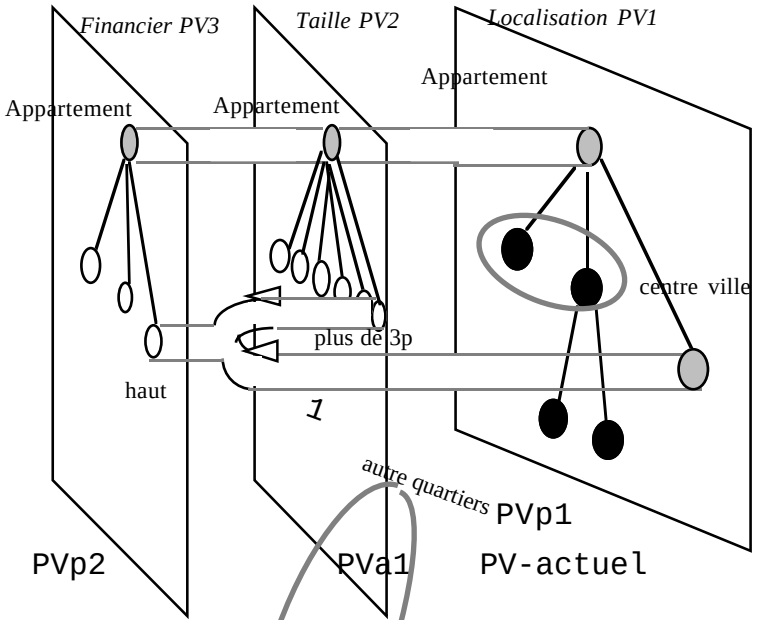


Fig. 6.15. En reprenant l'exemple de l'appartement : la classe *Centre_ville* du point de vue actuel étant marquée sûre, ses sœurs deviennent impossibles ainsi que les sous-arbres de celles-ci. La marque sûre de *Centre_ville* n'est pas propagée par la passerelle car l'autre source (*Plus_de_3p)* n'est pas encore marquée.

Mise à jour des passerelles et points de vue actifs

À part la propagation des marques, cette étape de classification doit mettre à jour le nombre de passerelles actives, Npass et l'état (actif ou inactif) de chaque point de vue. Npass est décrémenté (de 1) à chaque fois qu'une passerelle qui était encore active devient inactive, c'est-à-dire lorsqu'elle ne sert plus à faire des inférences.

Une passerelle devient inactive si :

- toutes ses classes (sources et destination) ont une marque sûre ou impossible.
- la destination d'une passerelle est marquée sûre :
- une des sources d'une passerelle est marquée impossible

Pour voir la validité de ces affirmations, il suffit de vérifier, avec les cas de propagation décrits auparavant, qu'aucune nouvelle marque pour une des classes non marquées de ces passerelles ne permet de faire de nouvelles inférences.

Un point de vue auxiliaire ou caché devient inactif quand le nombre de ses passerelles actives (Npass) est 0. Un point de vue principal passe à un état inactif quand la plus petite classe sûre de ce point de vue pour l'instance à classer est terminale.

Dans l'exemple de l'appartement, le point de vue *Localisation*, qui est le point de vue actuel, devient inactif car toutes ses classes sont déjà marquées. Comme nous allons montrer dans § 6.3.7, la prochaine itération de la boucle de classification se fera pour un autre point de vue (le point de vue actuel change).

Algorithmes de propagation de marques

Par la suite nous allons montrer le schéma général de parcours des points de vue (système de classification) pour faire la propagation des marques.

- [<S1,E1>,…,<Sn,En>,<D,E>] indique une passerelle ayant comme sources les classes S1, S2,…, Sn et comme destination la classe D (ces classes appartenant toutes à des points de vue différents : la source S1 a l'étiquette E1, S2 l'étiquette E2, …, et la destination D l'étiquette E (non marqué est noté '').
- La fonction *marque* (C) rend la marque de la classe C.
- La procédure *marquer* (C,E), marque la classe C avec l'étiquette E
- Le prédicat *sur-classe* (S,C) rend TRUE si S est une sur-classe de C, FALSE sinon
- Le prédicat *sous-classe* (S,C) rend TRUE si S est une sous-classe de C, FALSE sinon
- Le prédicat *sœur* (S,C) rend TRUE si S est sous-classe directe de la sur-classe directe de C, FALSE sinon
- Lorsque toutes les classes d'une passerelle sont marquées, celle-ci est enlevée de la liste des passerelles actives du concept (par la fonction désactiver(pass)). À la fin de la propagation, la liste de passerelles encore actives est parcourue pour identifier les points de vue inactifs.

Propagation de la marque sûre

propsûre (C)

{marque la classe C puis propage "sûre" aux sur-classes de C, propage "imp" aux sœurs de C. Si C est une source d'une passerelle avec destination impossible et dont après marquage de C seule une source est non marquée, les autres étant sûres, alors propage "imp" à cette source. Enfin si S est la source d'une passerelle dont toutes les autres sources sont sûre, propage "sûre" à la classe destination de la passerelle}

si marque (C) = ''

alors

marquer (C, SURE)

pour_tout S telle que sur_classe(S,C) -> propsûre (S)

pour_tout S telle que sœur (S,C) -> propimp (S)

si ∃ pass = [<S1,SURE> ,… ,<C,SURE>, …, <S_i,''>, …, <Sn,SURE>, <D,IMP>]

alors désactiver (pass)
propimp (Si)

si ∃ pass = [<S1,SURE>,…, <C,SURE>,…,<Sn,SURE>, <D,''>]

alors désactiver (pass)
propsûre (D)

propimp (C)

{ marque la classe C "imp"; propage "imp" à toutes ses sous-classes. Si C est la destination d'une passerelle dont toutes les sources sont "sûre" sauf une, alors propage "imp" à cette source non marquée }

si marque (C) = ''

alors

marquer (C,IMP)

pour_tout S telle que sous-classe (S,C) -> propimp (S)

si ∃ pass = [<S1,SURE> ,… , <S_j,''>, …, <Sn,SURE>, <C,IMP>]

alors désactiver (pass)
propimp (Si)

6.3.6. Mise à jour de l'information

La propagation des marques peut produire une spécialisation de l'instance dans un (ou plusieurs) point de vue PVi autre que le point de vue actuel, quand une passerelle est utilisée. Ce cas est reconnue par le fait que la classe sûre la plus petite de ce point de vue (C-Min / PVi) est une sous-classe de la classe indiquée dans le I-moule.

Pour rétablir complètement la cohérence au niveau de chacun des points de vues modifiés, il faut mettre à jour le type de l'instance (gardé dans le I-moule). Cette étape de la classification calcule le type des nouvelles plus petites classes sûres des différents points de vue et met à jour le type de l'instance, qui va être le type intersection du type de l'instance à ce moment et des types de ces nouvelles classes sûres.

Ainsi, par exemple, si le système ne peut plus rien déduire des points de vue principaux dans la classification de l'appartement *chez_Dugand*, et si en regardant le point de vue auxiliaire *Taille* il arrive à spécialiser l'instance I vers la classe *Plus_de_3P* alors, par propagation de marques, la classe *Haut* du point de vue *Financier* est marquée sûre (c'est la classe destination d'une passerelle dont toutes les sources sont sûres) (Fig. 6.15) et elle devient la nouvelle plus petite classe sûre de I dans ce point de vue. Comme cette classe est devenue une classe sûre pour l'instance, celle-ci valide son type. La procédure de **mise à jour de l'information** inclut alors le type de cette classe dans le type de l'instance (stocké dans le I-moule) - cette inclusion peut être vue logiquement comme l'intersection des ensembles des valeurs décrites par les deux types (ou simplement l'intersection des types).

6.3.7. Choix du prochain point de vue

Après la mise à jour de l'information, le système choisit le nouveau point de vue à considérer. Ce choix dépend du type du point de vue actuel (principal ou auxiliaire) et de l'état (actif ou inactif) de ce point de vue et des autres points de vue du concept.

En toute généralité, le choix du prochain point de vue vise à suivre la classification dans les points de vue principaux lorsque dans ceux-ci il y a encore des informations à déduire. Lorsqu'il y a encore des points de vue principaux actifs (avec une classe ouverte), la procédure choisit l'un d'entre-eux selon le principe suivant : si le point de vue actuel est un de ces points de vue principaux actif, il est gardé comme le point de vue actuel, sinon le nouveau point de vue est le premier point de vue principal actif selon l'ordre donné initialement par l'utilisateur dans la liste des points de vue principaux.

S'il n'y a pas de points de vue principaux actifs, alors, si le point de vue actuel est un point de vue auxiliaire actif dont la plus petite classe sûre est ouverte, il reste le point de vue actuel, sinon un point de vue auxiliaire actif ayant une classe sûre ouverte, qui est le premier de la liste fournie au début par l'utilisateur est choisi. Il faut noter que ce critère de choix est complètement arbitraire.

Si tous les points de vue principaux et auxiliaires sont inactifs, la classification est terminée.

Une alternative au choix du prochain point de vue présenté précédemment est d'avoir une mesure d'utilité des points de vue auxiliaires vis-à-vis de la classification et choisir le "meilleur" selon cette mesure. Cette mesure dépend de la connaissance que l'utilisateur a des différents points de vue.

Par exemple, dans la situation suivante (Fig. 6.16), lorsque la descente de I est bloquée dans le point de vue principal, le système peut utiliser deux critères distincts amenant à des sélections différentes du point de vue à prendre.

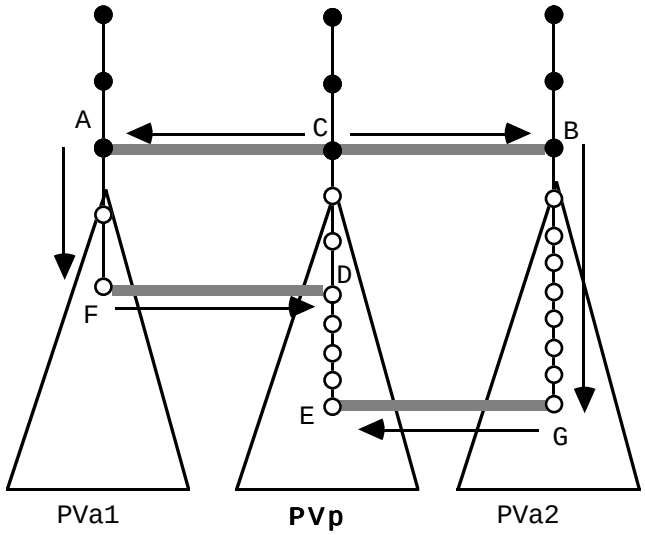


Fig. 6.16. Lorsque la descente de I s'arrête dans le points de vue principal PVp au niveau de la classe C, source de deux passerelles C->A et C->B, le système peut, soit continuer par PVa1 soit par PVa2.

Critère 1 : Avoir la passerelle vers un point de vue principal qui arrive le plus bas possible (à une classe très eloignée de la racine)

Pour un utilisateur ayant quelques connaissances du point de vue auxiliaire PVa2, il est plus utile de continuer la descente vers ce point de vue car, si le système parvient à classer l'instance dans la classe G, elle aura descendu 7 niveaux dans le point de vue principal; néanmoins ce choix va demander à l'utilisateur un grand nombre d'informations concernant le point de vue PVa2 pour vérifier l'appartenance de I aux classes qui se trouvent entre B et G.

Critère 2 : reprendre le plus vite possible un point de vue principal pour suivre la classification

Le critère opposé consiste à dire que, si l'utilisateur a choisi le point de vue PVp comme point de vue principal, c'est parce qu'il connaît la plupart des informations de l'instance I pour les attributs de ce point de vue. Donc, le système vise à reprendre le plus vite possible ce point de vue pour avoir plus d'information de la part de l'utilisateur. Avec ce critère, le point de vue choisi est PVa1.

6.4. Classification d'objets composés

À la différence des objets simples dont toutes les propriétés prennent leurs valeurs dans des concepts primitifs (entier, chaîne, réel, etc.), les objets composés ont des propriétés qui prennent leurs valeurs dans des concepts abstraits (définis par l'application) de la base de connaissances (§ 5.2.1). L'algorithme de classification que nous venons de présenter permet de classer aussi bien les instances simples que les instances composées. La seule différence entre la classification d'instances simples et la classification d'instances composées se trouve au niveau de la procédure de validation du type, qui est la procédure en charge de vérifier si une valeur donnée satisfait un type donné d'un attribut. Cette validation est à la base des procédures d'obtention d'information et d'appariement car les deux doivent faire des comparaisons entre les valeurs données par l'utilisateur et les contraintes connues (pour l'obtention d'information) ou en train d'être testées (pour l'appariement). Par la suite nous allons décrire la procédure de validation de

type lors de la classification d'une instance composée en indiquant son intégration à la procédure d'appariement.

6.4.1. Validation du descripteur de type

Un attribut est décrit dans un schéma de classe par son nom et éventuellement par des descripteurs *type*, *domaine*, *sauf*, *card*, *défaut* et *valeur*. Le quatres premiers descripteurs établissent l'affinement du type de l'attribut. La normalisation du type se fait en intégrant cet affinement au type hérité. Un type normalisé d'un attribut est décrit dans TROPES par trois éléments: les types de base issus de la normalisation du descripteur *type*, le domaine qui est la normalisation des descripteurs *domaine* et *sauf*, et la cardinalité normalisée pour les attributs multi-valués. La validation des deux derniers se fait par des comparaisons directes entre la valeur à valider et les valeurs, sa taille et les tailles admises[1].

Le premier élément, le descripteur de type normalisé, n'a de sens que pour les attributs abstraits (prenant leurs valeurs dans un concept défini) ; le type de base des attributs primitifs ne peut pas être spécialisé. Par la suite nous allons décrire la procédure de validation du descripteur de type d'un attribut défini ; cette validation peut déclencher toute une série de validations en cascade faisant intervenir des instances de concepts différents.

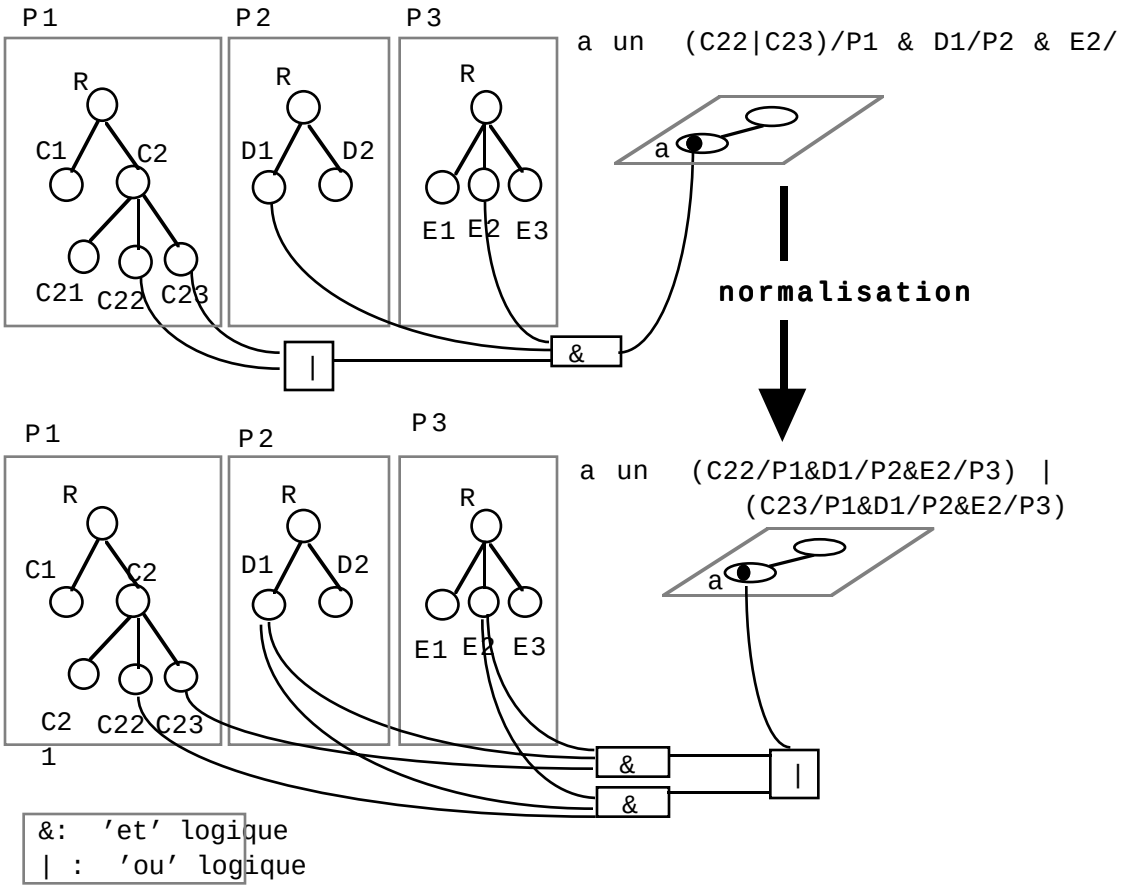


Fig. 6.17. Le résultat de la normalisation du descripteur de type est une liste de listes qui représente une disjonction de conjonction de classes. Chaque conjonction comporte une série de classes de points de vue différents auxquelles doit appartenir l'instance pour satisfaire le type.

[1]Dans le cas des attributs définis, la comparaison entre la ou les instance(s) qui correspond(ent) à la valeur de l'attribut et le domaine de celui-ci se fait en comparant leurs clés avec l'ensemble de clés admises.

Le descripteur de type d'un attribut **a** qui prend ses valeurs dans un concept T de la base est décrit par des disjonctions de classes de T d'un même point de vue ou par la conjonction de classes de T de points de vue différents (§ 5.6.1). La normalisation de ce descripteur produit une disjonction des conjonctions de classes de T représentée sous forme d'une liste de listes de classes (Fig. 6.17).

Valider la valeur **v** d'un attribut **a** d'une instance composée, pour une classe Cx, consiste à vérifier que v (qui est en fait une instance) appartient à toutes les classes d'une des listes-conjonctions $L_1 = \{C_1, C_2,\ldots,C_n\}$ de la liste-disjonction $\{ L_1, L_2,\ldots, L_m\}$ obtenue après la normalisation du descripteur du type de base de a dans Cx.

Dans la plupart des cas, le descripteur de type comporte une seule classe d'affinement. La vérification se réduit alors à vérifier que la valeur v appartient à cette classe, C_1. Cette vérification prend en compte trois cas possibles :

a) v est rattachée à une sous-classe de C_1 => v appartient aussi à C_1 (sûre) et elle valide donc le descripteur de type.

b) v est rattachée à une classe qui n'est pas comparable avec C_1 => v n'appartient pas à C_1 (impossible) et la validation échoue.

c) v est rattachée à une sur-classe D de C_1=> v peut appartenir à C_1 (possible). Il faut vérifier que v satisfait le type de la classe C_1. Cette vérification prend en compte aussi bien les valeurs des attributs de l'instance v que la connaissance déduite de son appartenance à D.

La procédure de validation proposée dans le dernier cas consiste à essayer de descendre une instance dans un point de vue, de sa classe d'appartenance, vers une sous-classe (pas nécessairement directe) de celle-ci, en vérifiant éventuellement des connaissances additionnelles sur les valeurs des attributs de v, pour les valider contre le type de ces attributs dans la classe C_1.

Si l'un des attributs de v, visible dans C_1, est lui-même un attribut défini , c'est-à-dire si v est, lui aussi, une instance composée, alors la vérification du type de v peut déclencher des vérifications sur ses attributs, puis sur les attributs de ceux-ci et ainsi de suite. Pour garantir la terminaison de cette suite, le modèle interdit la description de concepts récursifs, c'est-à-dire des concepts qui apparaissent plus d'une fois dans une suite d'attributs (§ 5.8.3).

Par la suite nous allons illustrer cette validation de type par un exemple de la procédure d'appariement d'une instance. Dans cet exemple, l'appariement entreprend de localiser l'instance dans l'une des sous-classes de sa classe courante du point de vue courant.

6.4.2. Validation de type pour l'appariement

Le problème de la validation du type d'un attribut d'un objet composé est le suivant : étant donné un objet composé *Ic* d'une classe *C* (d'un point de vue *PVi* d'un concept *C*) ayant un attribut *a* qui prend ses valeurs dans la classe *T* (d'un point de vue *PVy* du concept *A*), déterminer si *Ic* peut appartenir à la classe *Cx*, sous-classe de *C*, qui contraint les valeurs de *a* à être des instances de la classe *T1*, sous-classe de *T*.

Alors les trois cas décrits précédemment correspondent ici aux cas suivants :

Soit *Id* la valeur de l'attribut *a* pour l'instance *Ic*.

- *Id* appartient déjà à la classe *T1*, ou à une de ses sous-classes, alors elle satisfait bien le type de *T1*. *Cx* est marquée comme sûre pour *Ic* et elle devient la nouvelle plus petite classe sûre de *Ic* dans ce point de vue (Fig. 6.18).

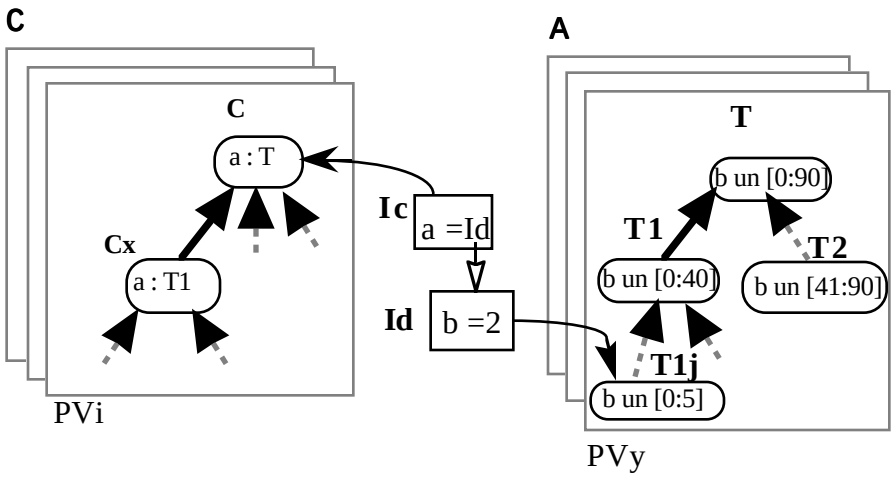


Fig. 6.18. L'attribut composant *a* de *Ic* est valué par l'instance *Id*. Celle-ci appartient à la classe *T1j*, sous-classe de *T1*, alors *Ic* peut être descendue à *Cx*.

- *Id* appartient à une classe *T2*, sous-classe de *T*, qui n'est pas comparable (par la relation de spécialisation) avec *T1*. Alors l'instance *Id* ne peut pas être attachée à *T1* et donc l'instance *Ic* ne peut pas être attachée à *Cx*. Cette dernière est marquée comme impossible pour *Ic* (Fig. 6.19).

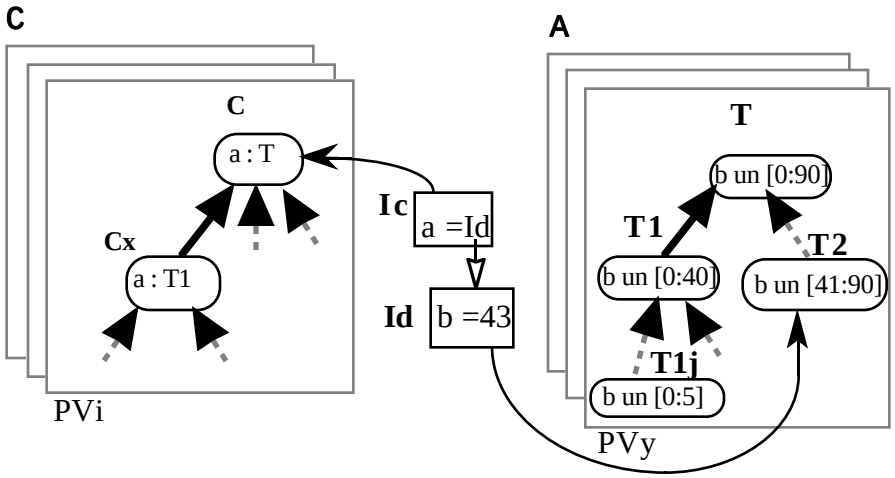


Fig. 6.19. L'instance *Id* appartient à la classe *T2*, disjointes de *T1*d. L'attribut *a* de *Ic* ne satisfait pas le type de *Cx*. *Cx* devient une classe *impossible* pour *Ic*.

- *Id* appartient à une classe *Tx*, sur-classe de *T1*. Dans ce cas pour savoir si *Id* peut appartenir à *T1*, il faut valider le type de *Id* par rapport à *T1* (Fig. 6.20).

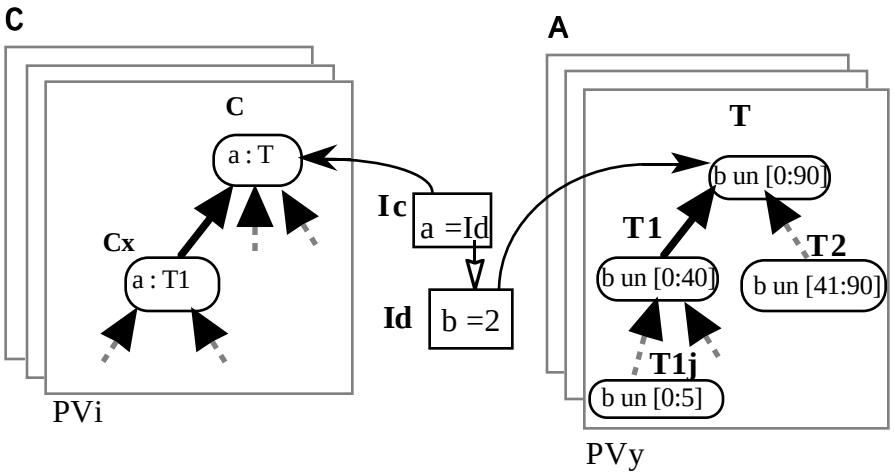


Fig. 6.20. *Id* appartient à *T*. Pour savoir si *Ic* peut être attachée à *Cx*, il faut vérifier si *Id* peut être attachée'à *T1*, c'est-à-dire vérifier si *Id* satisfait le type de *T1*.

Lorsqu'un attribut défini **a** d'une instance composée Ic est lui aussi une instance composée, Id, la vérification des types est realisée de la même façon sur les attributs abstraits de Id. Un processus récursif de vérification est donc lancé. Ce processus s'achève quand toutes les instances impliquées dans l'instance composée Ic, par exemple Id, vérifient les types spécifiés par les attributs abstraits dont elles sont les valeurs respectives, par exemple T1 pour Id (Fig. 6.21).

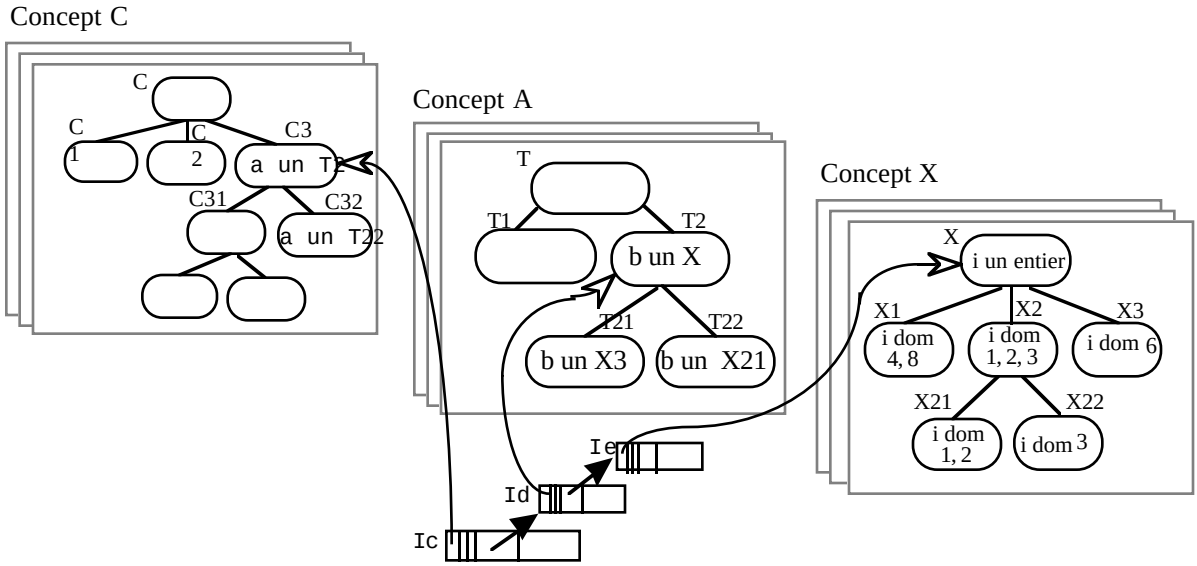


Fig. 6.21. Pour pouvoir attacher Ic, instance de C3, à C32, sa valeur Id pour l'attribut **a** doit valider le type de la classe T22 (en être une instance). L'algorithme essaie d'attacher Id, instance de T2, à T22. Id peut ëtre attachée à T22 si sa valeur Ie pour b est-un X21. Ainsi, si la valeur de i pour Ie est 1 ou 2, alors Ic est-un C32.

6.4.3. Les cycles dans les objets composés

Comme nous l'avons dit auparavant, la version actuelle de TROPES interdit la description d'objets circulaires (§ 5.8.3). Un objet composé dans TROPES ne peut pas faire partie de sa propre description. Cependant, la prochaine version du système va inclure des instances cycliques [EUZ93] issues de classes ayant une description de restriction circulaire [NEB91] ; une classe a une description de restriction circulaire si elle peut être atteinte par une chaîne d'attributs qui commence par sa définition.

L'extension du modèle pour inclure des instances cycliques nécessite une modification de l'algorithme de classification pour traiter ou au moins prévenir des inter-blocages éventuels. En effet la classification d'instances cycliques peut produire des *inter-blocages*. C'est un cas possible lorsque pour classer une instance O il faut vérifier que la valeur I d'un de ses attributs soit une instance d'une classe particulière. Si cette vérification déclenche la vérification de sa valeur pour un attribut b, et que cette valeur est exactement l'instance O, alors la classification est bloquée (Fig. 6.22).

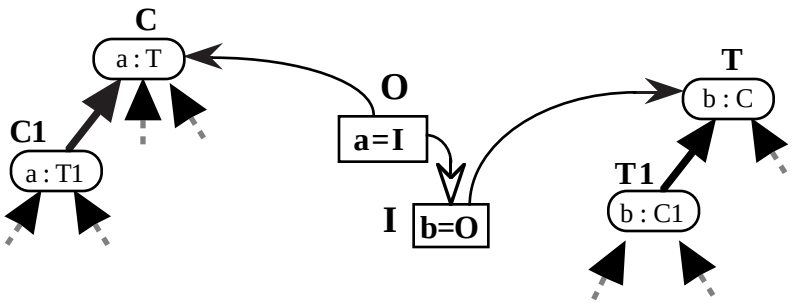


Fig 6.22. La classification essaie d'attacher O, instance de C, à C1. Pour ce faire, elle doit vérifier le type T1 pour I. Or I appartient à T1 seulement si son attribut b appartient à C1 ; dans ce cas, la valeur de l'attribut b de I est l'instance O. Donc, pour déduire que O appartient à C1, le système doit déjà savoir qu'elle lui appartient, ce qui crée un inter-blocage.

Conclusion

Dans ce chapitre nous avons présenté l'algorithme de classification multi-points de vue de TROPES. Cet algorithme de classification d'instances tire partie de la division en concepts d'une base TROPES et de la description multi-points de vue de chacun de ces concepts. De plus la propagation des marques à travers les passerelles permet d'accélérer la descente de l'instance et d'utiliser les connaissances sur un point de vue pour continuer la descente dans un autre.

L'algorithme de classification est paramétrable et modulaire. En effet, l'utilisateur peut configurer son environnement lors de la classification en organisant les points de vue d'un concept en trois catégories selon la connaissance qu'il a de chacun. Cette organisation est prise en compte par l'algorithme pour déterminer la navigation dans le concept et les attributs à demander à l'utilisateur. L'algorithme de classification TROPES est décrit par un cycle qui comporte cinq étapes (modules) bien définies : obtention d'information, appariement, propagation de marques, mise à jour de l'information et choix du prochain point de vue. Dans ce chapitre nous avons présenté la fonctionnalité de chacune de ces étapes.

Dans le chapitre 4 nous avons montré que les étapes principales de la classification sont l'appariement et le parcours du graphe de classes. Dans TROPES chaque point de vue d'un concept manipule un sous-ensemble des attributs du concept et un arbre simplifié des classes du concept. Donc, l'appariement dans un point de vue est en rapport avec un nombre plus réduit d'attributs et de classes que si cette même connaissance avait été représentée par une base mono-point de vue. De plus le programme ordonne les questions à poser à l'utilisateur lors de l'appariement, de façon à tirer le maximum d'informations de chacune de ses réponses. Enfin, la limitation du langage d'expression, aux aspects statiques des classes de la version courante de TROPES, réduit l'appariement à une vérification de type d'attribut. Dans le cas des objets composés, cette vérification de type peut entraîner une suite d'appels à une procédure de spécialisation qui valide

l'appartenance d'une instance à une classe. Les restrictions établies par le modèle pour la description d'objets composés garantissent la terminaison de cette suite de spécialisations.

La propagation de marques est la procédure qui profite au maximum des originalités de TROPES, à savoir, les passerelles entre les points de vue et l'hypothèse d'exclusivité mutuelle de classes sœurs. En effet, grâce à cette hypothèse l'identification d'une classes sûre pour l'instance dans un point de vue permet d'éliminer toute une partie de l'arbre du point de vue, en marquant les classes de cette partie de l'arbre comme des classes impossibles pour l'instance. Les passerelles offrent un mécanisme d'inférence de marques entre points de vue qui permet d'optimiser la classification de l'instance.

L'algorithme de classification de TROPES classe l'instance de façon incrémentale (au fur et à mesure que l'utilisateur donne des informations, le système produit des résultats partiels) et interactive qui s'appuie sur la normalisation de types et d'arbres de types des attributs pour optimiser la procédure d'appariement. Cette approche est également utilisée par [CAP&93] pour la classification de classe. Le parcours du graphe d'un point de vue correspond à peu près à la classification dans les logiques terminologiques [McGRE91] ou dans SHIRKA [REC88]. Le parcours du concept à travers des passerelles est tout à fait originale.

Un prototype de l'algorithme de classification multi-points de vue ainsi que du modèle de base, TROPES ont été développés en CAML [COU&90]. Bien que testée avec des exemples artificiels, cette réalisation est une première validation du modèle et de l'algorithme de classification associé.

Bibliographie

[CAP&93] CAPPONI C., CHAILLOT M., *Construction incrémentale d'une base de classes correcte du point de vue des types,* Actes Journée Acquisition-Validation-Apprentissage, Saint-Raphael, 1993.

[COU&90] COUSINEAU G., HUET G., *The CAML Reference Manual, V.2.6.1.* INRIA-ENS, 1990.

[EUZ93] EUZENAT J., *Classification dans les représentations par objets : produits de systèmes classificatoires,* Rapport interne, Equipe SHERPA, INRIA, 1993.

[McGRE&91] MAC GREGOR R.M., BURSTEIN M.H. *Using a Description Classifier to Enhance Knowledge Representation,* IEEE Expert Intelligent Systems and Applications, juin, 1991.

[MAR89] MARIÑO O., *Classification d'objets dans un modèle multi-points de vue,* Rapport de DEA d'informatique, INPG, Grenoble, 1989.

[MAR&90] MARIÑO O., RECHENMANN F., UVIETTA P. *Multiple perspectives and classification mechanism in object-oriented representation* , 9th ECAI, pp.425-430, Stockholm 1990.

[NEB91] NEBEL B., *Terminological cycles : Semantics and computational Properties,* in Principles of Semantic Networks. Explorations in the Representation of Knowledge. J. Sowa (éd.), Morgan Kaufman Publishing, chapitre.11, pp.331-361, 1991.

[QUI93] QUINTERO A., *Parallélisation de la classification d'objets dans un modèle de connaissances multi-points de vue,* Thèse d'informatique, Université Joseph Fourier, Grenoble, juin 1993.

[REC88] RECHENMANN F., *SHIRKA : système de gestion de bases de connaissances centrées-objet,* Manuel d'utilisation 1988.

[ROU88] ROUSSEAU B., *Vers un environnement de résolution de problèmes en biométrie : Apport des techniques de l'intelligence artificielle et de l'interaction graphique*, Thèse de doctorat, Université Claude Bernard Lyon 1, Lyon, 1988.

Chapitre 7

Correction et complexité

Chapitre 7

Correction et complexité

Introduction

Dans le chapitre précédent nous avons présenté l'algorithme de classification multi-points de vue de TROPES [MAR89], [MAR&90]. Dans ce chapitre nous présentons une preuve de la correction de l'algorithme et le calcul de sa complexité. La première partie du chapitre présente la preuve de la correction : nous prouvons en particulier la terminaison et la convergence de l'algorithme. Le formalisme utilisé pour décrire l'algorithme dans cette partie ainsi que les différents théorèmes démontrés sont le résultat d'un travail d'équipe développé avec Rodrigo Cardoso et Alejandro Quintero [QUI93] [CAR&92].

La deuxième partie du chapitre concerne le calcul de la complexité de l'algorithme. Ce calcul est fait avec le modèle RAM (random acces machine) proposé par [AHO&74] pour calculer la complexité maximale de l'algorithme. Les procédures d'optimisation, comme par exemple le tri des attributs à demander à l'utilisateur dans l'étape d'appariement, devraient réduire cette complexité dans le cas moyen.

7.1. Correction de l'algorithme

Dans le chapitre précédent nous avons présenté l'algorithme de classification multi-points de vue de TROPES. Dans cette partie nous allons montrer la correction de cet algorithme. En particulier nous allons prouver sa convergence : le résultat de la classification d'une instance est unique, indépendamment de l'ordre dans lequel sont visités les points de vue du concept. Cette preuve est triviale lorsqu'il n'y a pas de passerelles entre des points de vue du concept de l'instance, car dans ce cas, la classification d'une instance peut être vue comme plusieurs classifications indépendantes, une sur chaque point de vue. Nous allons montrer que c'est aussi le cas lorsque le concept possède des passerelles.

Pour la démonstration nous allons définir les notions de **forme de classification**, **arbre de classification**, **trace de classification**, **système de classification**, **état accessible** et **S-classification**.

7.1.1. Forme de classification

Le triplet FC = (X,C,f) est une **forme de classification** pour X ssi :

- X est un ensemble d'individus, appelé l'univers de FC.
- C est une partition de X', $X' \subseteq X$, $C = \{X_1,\ldots,X_h\}$ avec $h \geq 1$, et telle que $\forall i \in [1..h]$, $Xi \subseteq X'$, et $Xi \cap Xj = \emptyset$ pour tout $i \neq j$.

- $f : X \rightarrow \{0,1,\ldots,h\}$ et telle que $\forall i \in [1..h]$, $f(x)=i \Leftrightarrow x \in X_i$ et $f(x)=0 \Leftrightarrow (\forall i \in [1..h], x \notin X_i)$. f est appelée la fonction de classification et $X_0 = X$.

Dans la définition précédente, X représente une classe, C représente l'ensemble des sous-classes de X (ces classes sont numérotées à partir de 1) et f représente la fonction d'appariement qui permet de décider à quelle sous-classe de X appartient une instance de la classe X. Dans le cas d'une taxonomie, l'union des sous-classes de X, X' est toute la classe X.

Il est important de rappeler que l'hypothèse de départ de notre algorithme est que des classes sœurs représentent des ensembles disjoints d'instances ($Xi \cap Xj = \emptyset$ pour tout $i \neq j$). C'est grâce à cette hypothèse que l'appariement peut être formalisé par une fonction (une instance d'une classe appartient au plus à une des sous-classes de la classe).

La fonction de classification rend 0 ou le numéro de la sous-classe vers laquelle doit descendre l'instance. La fonction rend 0 quand l'instance ne peut être descendue vers aucune des sous-classes de X. Ce cas peut signifier ou bien que la division en sous-classes n'est pas exhaustive et que la classe d'appartenance de l'instance la plus spécialisée est X, ou bien que l'instance est incomplète et qu'avec l'information disponible le système ne peut pas poursuivre la descente dans ce point de vue.

Par exemple (Fig. 7.1), la spécialisation de la classe matrice carrée en des classes Symétrique et Non_Symétrique correspond à la forme de classification FC = $<X,\{X_1,X_2\},f>$ où X = matrice carrée, X_1 = Symétrique, X_2 = Non_Symétrique et f est la fonction définie par

f : X -> {0,1,2}

f(M) = 1 si Mij = Mji, i,j :1…m,

2 sinon (m est la dimension de M et Mij sa valeur dans la position (i,j))

La partition étant exhaustive, $X = X_1 \cup X_2$, f ne rend jamais 0.

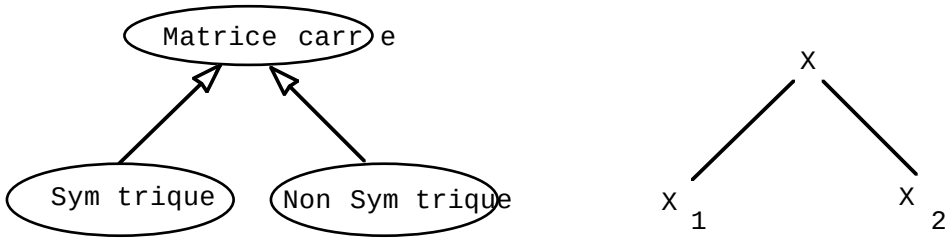


Fig. 7.1. La spécialisation de la classe Matrice carrée en symétrique et Non_Symétrique peut être décrite par la forme de classification FC= < X, $\{X_1,X_2\}$, f>, avec X= matrice carrée, X1 = Symétrique, X2 = Non_Symétrique et f(x)=1 si x est symétrique, 2 sinon.

Le cas particulier où h=0 est une forme de classification dégénerée : la classe X n'a pas de sous-classes (C = { }) et $\forall x \in X$, f(x)=0.

Un élément x de X est **classifié** pour la forme de classification FC = (X,C,f) lorsque l'appartenance de x à l'un des ensembles de la partition C de X est déterminée en accord avec la valeur de f pour x. Si $h \geq 1$ et f(x)=0 alors x ne peut pas être classifié dans l'un des sous-ensembles de X donnés par C, il est alors classifié dans X.

Une forme de classification est définie pour un niveau donné. Par la suite nous allons introduire la notion d'arbre de classification qui étend les idées de forme de classification à tout l'arbre de classes.

7.1.2. Arbre de classification

Un **arbre de classification** A(X), noté AC, est un arbre fini dont les nœuds sont des formes de classification FC et tel que :

- La racine est une FC = (X,C,f).
- Si un nœud de l'arbre est une FC=(Y,D,g) et qu'il a comme successeurs les FC (Y_1,D_1,g_1) ,…, (Y_h,D_h,g_h)

  alors la partition D est donnée par les Y_i : D = $\{Y_1,\ldots,Y_h\}$.

Dans la définition précédente, A(X) représente un point de vue (la classe racine est X) et D est l'ensemble des sous-classes d'une classe.

Par exemple (Fig. 7.2), l'arbre de classification correspondant au point de vue "forme" du concept "matrices" possède une première FC, la racine, qui a comme univers l'ensemble de matrices; comme partition de cet ensemble les ensembles "matrice carrée" et "matrice rect_non_carrée". Le nœud "matrice carrée" est la FC présentée auparavant (Fig. 7.1) qui a comme univers l'ensemble de matrices carrées. Le nœud "matrice rect_non_carrée" est une feuille de l'arbre qui correspond à une FC dégénerée, n'ayant pas de partition (sa fonction de classification rend toujours 0).

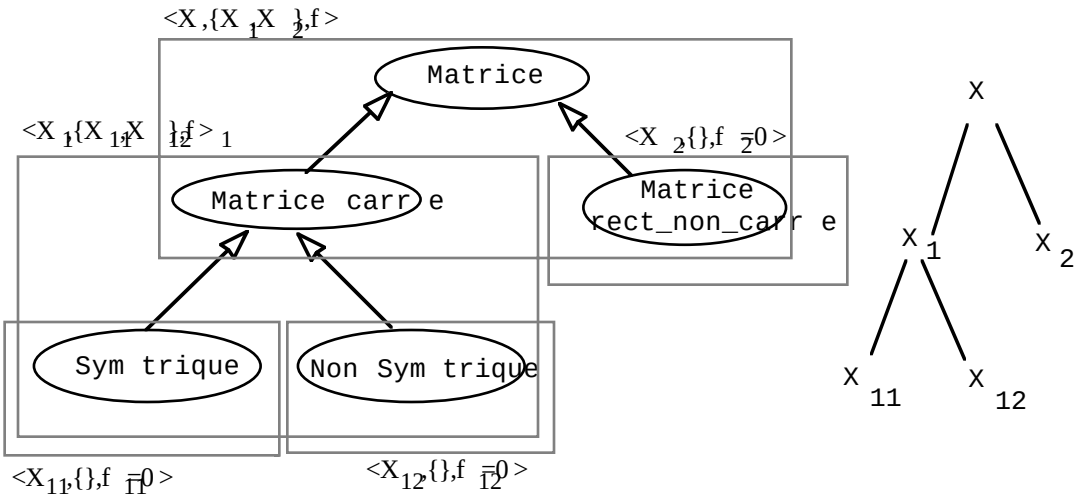


Fig. 7.2. Un arbre de classification est un arbre fini dont tout nœud est une forme de classification FC; l'univers de la FC d'un nœud (autre que la racine) est un des éléments de la partition de son nœud père dans l'arbre. Le point de vue "forme" du concept "matrice" est décrit par l'arbre de classification A(X) = {<X,{X1,X2},f>, <X1,{X11,X12},f1>, <X2,{},0>, <X11,{},0>, <X12,{},0>}, où X = matrice, X1 = Matrice carrée, X2 = matrice rect_non_carrée, X11 = symétrique, X12 = Non_Symétrique.

Un AC peut être étiqueté en marquant chacun de ses nœuds par sa position (suite d'entiers qui indique le chemin de la racine vers ce nœud). Ainsi

- la racine est dénotée par (X,C,f) (ou bien $(X_\lambda ,C_\lambda ,f_\lambda)$[1])
- pour une FC dénotée par (Xα,Cα,fα) sa i-ème fille est dénotée par (Xαi,Cαi,fαi) .

L'ensemble de positions de A(X), noté Pos(A), est donné par l'ensemble des suites finies d'entiers non nuls des nœuds de l'arbre A. Les positions sont ordonnées par ordre lexicographique $<_p$: pour deux positions $\alpha,\beta \in Pos(A)$, $\alpha <_p \beta :\Leftrightarrow \alpha$ est un préfixe de β.

[1] λ est la suite vide de numéros.

Ainsi, l'arbre étiqueté de l'exemple précédent (Fig. 7.2) possède l'ensemble de positions : Pos(A) = {λ, 1, 2, 1.1, 1.2} où $\lambda <_p 1$; $\lambda <_p 2$, $1 <_p 1.1$; $1 <_p 1.2$.

$U_{AC} = \{X_\alpha \mid \alpha \in Pos(A)\}$ est l'ensemble des univers[1] de A

Le but de la classification est de faire descendre un objet le plus bas possible dans l'arbre de classes. Si l'on considère le cas d'une instance x de la racine ($x \in X$), alors la classification revient à trouver un $\alpha \in Pos(A)$ tel que $x \in X_\alpha$, et que X_α soit maximal par rapport à l'ordre $<_p$ des positions.

7.1.3. Trace de classification

Soit A(X) un AC, et $x \in X$.

La **classification** de x par rapport à A(X), notée $C_A(x)$, est l'univers $X_{\lambda n_1 \ldots n_r}$ tel que :

- Pour $0 \le i < r$: $f_{n_0 \ldots n_i}(x) = n_{i+1}$, $n_0 = \lambda$
- $f_{n_r}(x) = 0$

Ainsi, la classification $C_A(x)$, est le plus petit univers $X_{\lambda n_1 \ldots n_r} = X_\alpha$ pour lequel on peut affirmer $x \in X_\alpha$, c'est-à-dire le nœud le plus bas dans l'arbre (celui avec la position maximal).

La **trace de classification** de x par rapport à A(X), notée $A_t(x)$, est le chemin qui mène de X à $C_A(x)$, uni au sous-arbre dont la racine est $C_A(x)$: $A(C_A(x))$ (Fig. 7.3)

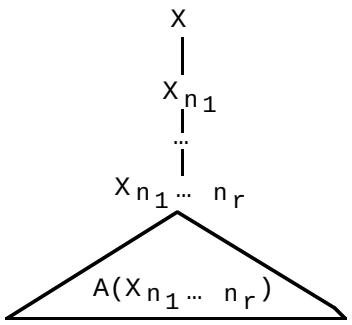


Fig. 7.3. Trace de classification de x par rapport a A(x) : $A_t(x)$; $A(X_{n1} \ldots nx)$ est le sous-arbre ayant comme racine $X_{n1} \ldots nx$.

Ainsi, dans l'exemple des matrices, la classification de la matrice identité I par rapport au point de vue "forme" est la classe "symétrique"= X_{11}. Dans ce cas $f_0(I)=1$, $f_1(I)=1.1$ et $f_{1.1}(I)=0$ et la trace de classification de I par rapport à "forme" est $A_t(x) = \{X, X_1, X_{1.1}\}$.

7.1.4. Système de classification

Un **système de classification** S(X), noté SC, est une paire <B,P> telle que :

- $B = \{A^1(X), \ldots, A^m(X)\}$ est un ensemble fini de ACs de l'ensemble X.
- P = ensemble de passerelles défini par une relation binaire sur l'ensemble

[1]Par univers d'un nœud X, on entend l'ensemble des instances décrites par X, la classe X.

$U_B = U_{\backslash s(;A1)} \cup \ldots \cup U_{\backslash s(;Am)}$, ne mettant pas en rapport deux éléments du même ensemble U_{Ai}, [1]

Ainsi, un couple $<x^i_\alpha, x^j_\beta> \in P$, i≠j, et β n'est pas un prefixe[2] de α, est une *passerelle* allant de x^i_α vers x^j_β . [3]

Un système de classification correspond à un ensemble d'arbres de classification et des passerelles, donc à un concept de la base de connaissances. Ainsi, par exemple le concept "matrice", (Fig. 6.2) peut être décrit par le système de classification <B,P> où

B= { $A^1(X)$, $A^2(X)$, $A^3(X)$, $A^4(X)$}

avec X = matrice, A^1=Structure, A^2=Forme, A^3= Contenu et A^4= Remplissage, et

P= { <Matrice^1,Matrice^2>, <Matrice^2,Matrice^1>, <Matrice^1,Matrice^3>,
< Matrice^3,Matrice^1>, <Matrice^1,Matrice^4>, <Matrice^4,Matrice^1>,
<Matrice^2,Matrice^3>, <Matrice^3,Matrice^2>, <Matrice^2,Matrice^4>,
<Matrice^4,Matrice^2>, <Matrice^3,Matrice^4>, <Matrice^4,Matrice^3>,
<Matrice_diag^1,Symétrique^2>, <Définie_positive^3,Matrice_carrée^2>}

La classification multi-points de vue consiste alors à étendre la classification d'un arbre de classification à toute la forêt d'arbres décrite par un système de classification. Le résultat de la classification multi-points de vue n'est plus un univers d'un arbre, mais un tuple d'univers (un pour chaque arbre ou point de vue). La classification multi-points de vue est la combinaison de plusieurs classification simples et des passerelles entre les arbres. En effet, si par classification de x dans un arbre A^i on déduit $x \in x^i_\alpha$ et qu'il y a une passerelle $<x^i_\alpha, x^j_\beta> \in P$, alors on a aussi $x \in x^j_\beta$.

Ainsi, si la matrice identité, I, est reconnue comme étant une matrice diagonale dans l'arbre de classification A^1 (structure), alors en utilisant la passerelle < Matrice_diag^1, Symétrique^2>, le système peut inférer que la matrice identité est une matrice symétrique, dans l'arbre de classification A^2 (structure) : il fait ainsi un raccourci dans la descente de I dans A^2.

7.1.5. S-classification

Nous allons définir par la suite le terme S-classification, qui généralise la classification d'un arbre à tout le système de classification. Pour cela nous allons utiliser la notion d'état de classification et d'état accessible. Informellement, un état de classification est une combinaison d'univers des différents arbres (un tuple de classes, dont une de chaque point de vue) et un état accessible pour une instance x à partir d'un état e est un état par lequel peut passer x lors de sa classification à partir de e. Un état est accessible depuis un autre s'il y a une chaîne d'états directement accessible entre les deux. Un état e' est *directement accessible* par un état e s'ils sont identiques sauf pour un des composants du tuple ; pour ce composant, e' a une classe plus basse dans l'arbre, obtenue par une descente directe ou par une passerelle.

---

[1] Pour simplifier les explications, nous ne traitons ici que les passerelles ayant une seule classe source. Pour les passerelles avec plusieurs sources, il suffit d'étendre la définition à une relation n-aire, n<m.

[2] β est un prefix de α si la suite de numeros de α contient celle de β, α = βx, x ≠ λ

[3] X^j_β signifie la classe Xβ du point de vue (arbre) j.

Soit $S = <B,P>$ un SC, avec $B = \{A^1,\ldots,A^m\}$, $x \in X$.

- L'ensemble des **états de la classification** pour x, noté EC, est le produit cartésien :

  $U_{A^1} \times \ldots \times U_{A^m}$

- L'ensemble E des **états "accessibles"** (pour x) à partir d'un état initial, noté ECA, est un sous-ensemble de EC défini par induction par :

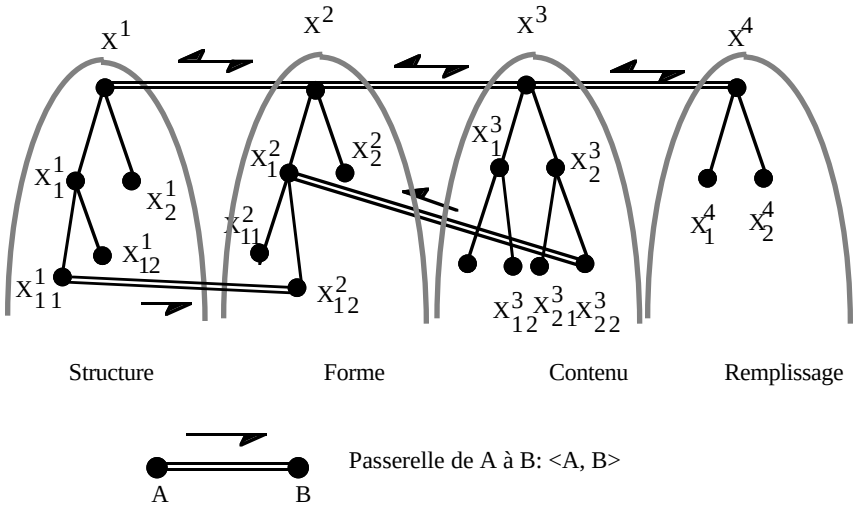
  - L'état initial (les racines) est un état accessible (toute instance appartient aux racines de son concept) : <X,…,X> ∈ ECA

  - Soit $e = <x^1_{\alpha_1},\ldots, x^m_{\alpha_m}> \in ECA$, alors $e' = <x^1_{\beta_1},\ldots, x^m_{\beta_m}> \in ECA$ si e' a les mêmes composants que e, excepté un ensemble de composants. Il existe au moins un composant different. Soit i un de ces composants différents, alors, ou bien :

    - $\beta_i = \alpha_i f_{\alpha_i}(x)$, ou bien
    - il existe une passerelle $<x^j_{\alpha_j}, x^i_{\beta_i}> \in P$, pour un j, $1 \le j \le m$.

  Dans ce cas on dit que e' est **"accessible" à partir de** e, noté $e \rightarrow^*_R e'$.

- Une **S-classification** pour x est un état accessible à partir duquel il n'existe aucun autre état accessible, c'est-à-dire, un état terminal.



Soit $e = <x^1_1, x^2, x^3_2, x^4>$. Les états $e' = <x^1_{11}, x^2, x^3_2, x^4>$, $e'' = <x^1_{11}, x^2, x^3_2, x^4_1>$, $e''' = <x^1_{11}, x^2_{12}, x^3_2, x^4_1>$, $e'''' = <x^1_{11}, x^2_{12}, x^3_{22}, x^4_1>$ sont des états successivement atteignables pour une instance x à partir de e.

Fig. 7.4. : Le concept "matrice" a 4 points de vue. Formellement, le système de classification de matrice a 4 arbres de classification et 8 passerelles (dont 6 triviales); La descente d'une instance lors de la classification est décrite en terme des états (tuples de classe) par lesquels elle peut passer.

Ainsi, par exemple, pour une instance x du concept matrice (Fig. 6.2) en partant d'un état initial <Matrice_bande, Matrice, Valeurs_réelles, Matrice> on peut avoir la suite d'états accessibles : <Matrice_diagonale, Matrice, Valeurs_réelles, Matrice>, en descendant dans le point de vue structure ; <Matrice_diagonale, Matrice, Valeurs_réelles, Matrice_noncreuse>, dans remplissage ; <Matrice_diagonale, Symétrique, Valeurs_réelles, Matrice_non_creuse>, en utilisant la passerelle de Matrice_diagonale à Symétrique; et enfin, en descendant dans le point de vue Contenu, on arrive à l'état final <Matrice_diagonale, Symétrique, Définie_positive, Matrice_noncreuse>. Cet état est une **S-classification** si à partir de celui-ci il n'y a plus d'état accessible pour x (Fig. 7.4).

Dans un système de classification, on veut interpréter le passage d'un état à un état suivant comme un gain d'informations dans le processus de classification, comme un affinement de la connaissance que l'on a de l'instance. Cela est bien le cas lorsque le passage d'un état à un autre se fait en utilisant les FC (en descendant par un point de vue). En ce qui concerne les passerelles, il faut garantir que leur utilisation ne fasse pas "reculer" (monter dans l'arbre) la classification et n'entraîne des inconsistances. Par la suite nous allons montrer les cas problématiques. Ces cas sont contrôlés par le système lors de la création d'une base.

7.1.6. Consistance d'un système de classification

Soit $S = <B,P>$ un SC.

- S est **redondant** s'il existe deux passerelles $<X^{i}_{\alpha\beta}, X^{j}_{\gamma}>$ et $<X^{j}_{\gamma\delta}, X^{i}_{\alpha}> \in P$ (Fig. 7.5) ou s'il existe une chaîne de passerelles $<X^{i_1}_{\alpha\beta}, X^{i_2}_{\gamma}>$, ..., $<X^{i_k}_{\gamma\delta}, X^{i_l}_{\gamma^{\alpha}}>$,..., $<X^{i_j}_{\gamma^{\varphi}}, X^{i_1}_{\alpha}> \in P$.

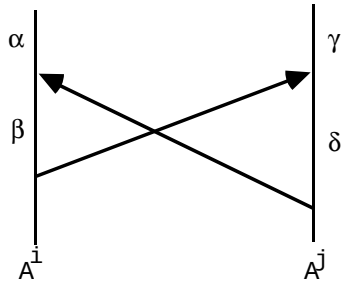


Fig. 7.5. Schéma d'un SC redondant. Lors de la classification l'une des deux passerelles ne servira pas à la descente de l'instance.

- S est **inconsistant** s'il existe un $x \in X$ pour lequel il y a deux états accessibles e et e' différant dans un seul composant, disons le k-ième :

  $e = <X^{1}_{\alpha_1},\ldots,X^{k}_{\alpha_\kappa},\ldots, X^{m}_{\alpha_m}>$, $e' = <X^{1}_{\alpha_1},\ldots,X^{k}_{\beta_\kappa},\ldots, X^{m}_{\alpha_m}>$ et α_k n'est pas un préfixe de β_k ni l'invers.

  Un SC **consistant** est un SC qui n'est pas inconsistant.

Supposons, par exemple, qu'une instance I du concept matrice de la base soit classée dans la classe Matrice_diagonale du point de vue Structure : elle arrive alors à l'état e1=<Matrice_diagonale,.,.,.,.>.

Supposons aussi que du point de vue Forme elle soit classée dans la classe Non_Symétrique : elle arrive à l'état e2 = <…,Non_Symétrique,…,…>.

Alors à partir de e1, en utilisant la passerelle <Matrice_diagonale,Symétrique> la classification peut atteindre l'état e3 = <Matrice_diagonale,Symétrique,..,..,.. >.

Or cet état est inconsistant avec l'état e2, également atteint par l'instance, car les classes Symétrique et Non_Symétrique du point de vue Forme ne sont pas comparables par la relation d'ordre entre classes (l'étiquette de l'une n'est pas préfixe de l'étiquette de l'autre)

- S est complet par rapport aux passerelles, noté **(Pas-)complet** si toutes les sous-classes d'une source d'une passerelle, sont aussi des passerelles vers la même destination, c'est-à-dire si l'implication suivante est satisfaite :

$<X^i_\alpha, X^j_\beta> \in P \wedge \quad \alpha\gamma \in Pos(A^i) \Rightarrow \quad <X^i_{\alpha\gamma}, X^j_\beta> \in P$

`S` est (`Pas-`)incomplet , si `S` n'est pas complet.

- Le système de classification `cmp(S) = <B,P1>` est la **complétion**[1] de `S`, si

  $P1 = P \cup \{<X^i_{\alpha\gamma}, X^j_\beta> \mid <X^i_\alpha, X^j_\beta> \in P \wedge \alpha\gamma \in Pos(A^i)\}$ (Fig. 7.6)

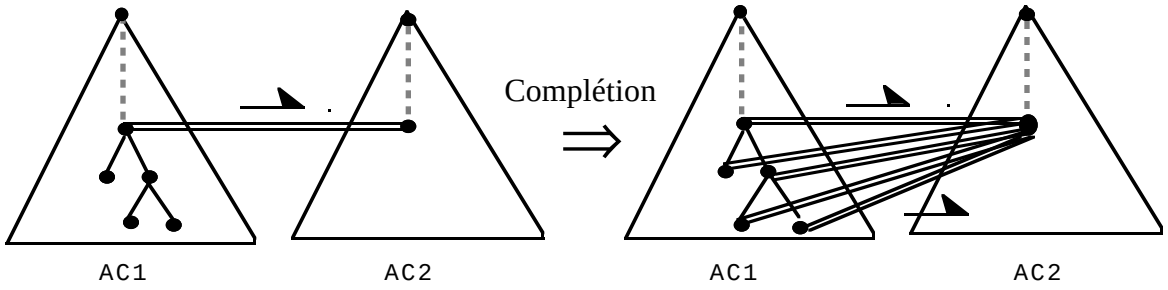


Fig. 7.6. Le système de classification de droite est la complétion du système de gauche

L'utilisation des passerelles dans un SC redondant peut entraîner des cycles pendant le processus de classification. Néanmoins, ces cycles sont interdits par la définition d'état accessible ($\rightarrow_R^*$).

En effet,

si le processus de classification se trouve dans l'état $e = <\ldots,X^i_{\alpha\beta},\ldots,X^j_{\gamma\delta},\ldots>$

et qu'il existe les passerelles $<X^i_{\alpha\beta}, X^j_\gamma>, <X^j_{\gamma\delta}, X^i_\alpha> \in P$,

alors les états $e' = <\ldots,X^i_\alpha,\ldots,X^j_{\gamma\delta},\ldots>$, $e'' = <\ldots,X^i_\alpha,\ldots,X^j_\gamma,\ldots>$ et $e''' = <\ldots,X^i_{\alpha\beta},\ldots,X^j_\gamma,\ldots>$

ne sont pas accessibles (par la définition d'état accessible).

Il est intéressant de noter que les états inconsistants peuvent se présenter seulement lors de la présence d'une passerelle (à l'intérieur de la classification dans un arbre il n'y a pas d'incohérences).

De plus, il faut noter que

- Si l'ensemble de passerelles d'un SC est vide, alors SC est trivialement complet.
- La complétion d'un SC consistant est aussi consistante.
- la complétion d'un SC est toujours un SC complet, car le fait d'ajouter de nouvelles passerelles pour la complétion n'ajoute pas de nouvelles connaissances au SC.

7.1.7. Convergence de la classification

Par la suite, nous allons utiliser les résultats de consistance et de complétion montrés précédemment, pour prouver la confluence du système de classification. Cette confluence garantit que le résultat final de la classification d'une instance dans un concept est unique, indépendamment de l'ordre du parcours des passerelles et le choix du prochain point de vue dans chaque étape de la classification .

Théorème 1
Soit `S = <B,P>` un SC consistant et complet, alors $\forall x \in X$ il n'existe qu'une `S`-classification.

[1] L'idée de complétion est donnée informellement dans (§5.4.).

Preuve

Soit $x \in X$; un état e accessible est représenté pour un m-tuple d'univers (de classes) des FCs de chaque AC du SC. Un "prochain" état est accessible à partir de l'état e :

- soit en descendant verticalement dans un ou plusieurs composants du tuple. La descente verticale veut dire que dans un AC l'instance est descendue à une sous-classe de la classe actuelle.
- soit en utilisant une passerelle ou plusieurs pour changer les composants du tuple. Dans un SC consistant, l'utilisation d'une passerelle entraîne un changement d'un des composants du tuple.
- soit en changeant simultanement les composants par la descente verticale et par l'utilisation des passerelles.

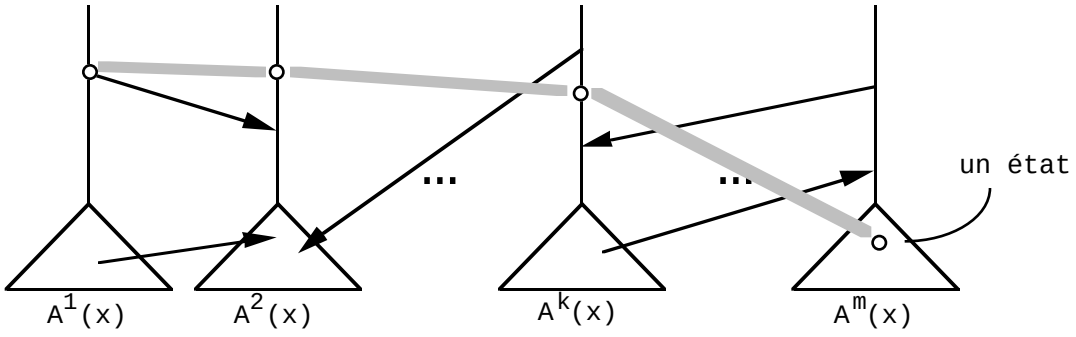


Fig. 7.7. Pour chaque AC^k est présentée la trace de classification de x, $A^k_\tau(x)$; les flèches représentent les passerelles.

Soit $Q = \{<\alpha_1,\ldots,\alpha_m> | <X_{\alpha_1},\ldots,X_{\alpha_m}> \in ECA\}$, et soit le graphe $G\ (Q, .\rightarrow.)$, tel que :

1. $<\alpha_1,\ldots,\alpha_m> \rightarrow <\beta_1,\ldots,\beta_m> \Leftrightarrow <X_{\beta_1},\ldots,X_{\beta_m}>$ est accessible à partir de $<X_{\alpha_1},\ldots,X_{\alpha_m}>$.
2. La relation $.\rightarrow.$ est définie comme :

   $<\alpha_1,..,\alpha_m> \rightarrow <\beta_1,..,\beta_m> \Rightarrow <\alpha_1,..,\alpha_m> <_p <\beta_1,..,\beta_m>$. ($\forall i$, α_i est un préfixe de β_i).

Cela signifie que le graphe G ne possède pas de cycle[1], donc chaque séquence d'états accessible est de longueur finie. Les séquences maximales en partant de l'état $<X,\ldots,X>$ finissent toujours en S-classifications.

Par ailleurs, si nous considérons le graphe G comme un graphe de réduction, la confluence locale entraîne la confluence et la canonicité, car pour un système de réécriture R noethérien : R est confluent si et seulement si il est localement confluent. Donc, l'ensemble de S-classifications doit être unitaire, car il correspond aux formes normales de la réduction. La confluence locale d'un système S peut être montrée par la convergence des paires critiques de S. D'après le théorème de G. Huet, un système de réécriture est localement confluent si et seulement si toutes ses paires critiques convergent.

Soit $a, b, c \in Q$, tels que $a \rightarrow b$, $a \rightarrow c$. Il faut montrer que b et c sont convergents, c'est-à-dire qu'il existe un w tel que $b \rightarrow^*_R w$ et $c \rightarrow^*_R w$. La figure suivante (Fig. 7.8) montre les différents cas qui peuvent se présenter :

[1]Il est important de distinguer entre le graphe G et le système de classification. Le premier ne comporte que la suite d' états accessibles pour l'instance et l'accessibilité est définie de façon à ne pas avoir des cycles. Un SC peut avoir des cycles (il est alors redondant).

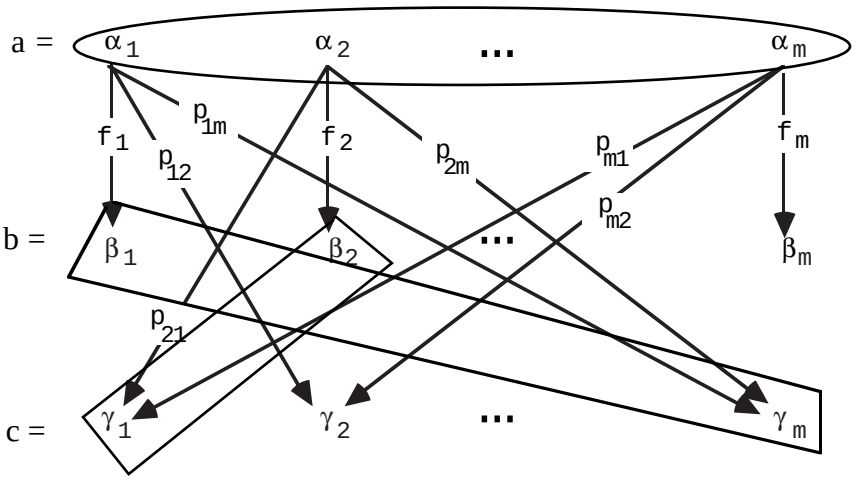


Fig. 7.8. Paires critiques obtenues pendant la classification. Par exemple a → b et a → c.

En partant de $a = <\alpha_1, \ldots, \alpha_m>$, les réductions possibles de a,b et c sont décrites par les m-tuples dont le composant α_k est changé par β_k ou par γ_k, $\forall k=1,\ldots,m$. Le composant α_k est changé en β_k ($\alpha_k \overset{f_k}{\rightarrow} \beta_k$) si l'on utilise la fonction de classification f_k (plus précisément, la fonction f_{α_k}). Le changement de γ_k peut être justifié par l'existence d'une passerelle p_{ik}, avec $i \neq k$ (qu'il s'agisse de passerelles initiales ou de celles ajoutées lors de la complétion) (Fig.7.9). Finalement, s'il n'y a pas de changement dans le composant k, la justification est la fonction identité id_k.

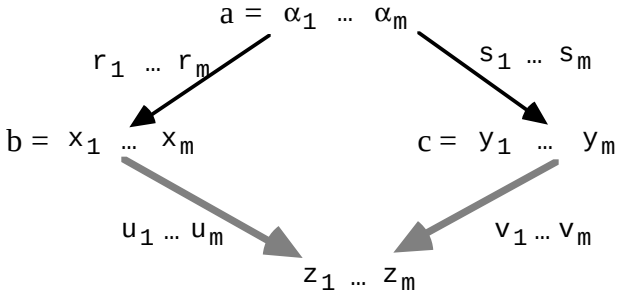


Fig. 7.9. Convergence des paires critiques b et c. b est égal à $<x_1, \ldots, x_m>$ et c est égal à $<y_1, \ldots, y_m>$.

Ainsi, b est égal à $<x_1, \ldots, x_m>$ et c est égal à $<y_1, \ldots, y_m>$. Les étiquettes $<r_1, \ldots, r_m>$ qui nomment les arcs reliant a et b sont les justifications des changements dans les composants. Elles sont définies comme :

r_k = id_k , si $x_k = \alpha_k$
= f_k , si $x_k = \beta_k$
= p_{jk} , si $x_k = \gamma_k$. Dans ce dernier cas, la passerelle p_{jk} doit exister.

De la même façon sont définies les étiquettes $<s_1, \ldots, s_m>$, qui justifient le passage de a à c. Maintenant, il faut trouver un m-tuple $d=<z_1, \ldots, z_m>$ pour lequel b et c convergent. Pour cela, nous allons expliquer comment doivent se transformer les composants des deux m-tuples à l'aide des transformations disponibles. Les étiquettes $<u_1, \ldots, u_m>$ et $<v_1, \ldots, v_m>$ seront utilisées pour indiquer les transformations correspondantes (Fig. 7.10).

x_k	y_k	u_k	v_k
α_k	α_k	id_k	id_k
α_k	β_k	f_k	id_k
α_k	γ_k	p_{jk}	id_k
β_k	α_k	id_k	f_k
β_k	β_k	id_k	id_k
β_k	γ_k	p_{jk}	id_k
γ_k	α_k	id_k	p_{jk}
γ_k	β_k	id_k	p_{jk}
γ_k	γ_k	id_k	id_k

Fig. 7.10. Définition des u_k et v_k, pour $k=1,\ldots,m$, en fonction des valeurs possibles de x_k et y_k.

Il faut noter l'utilisation de passerelles qui n'apparaissent pas dans le graphe (Fig. 7.9). L'ajout de ces passerelles est justifié par la complétion du SC. C'est le cas de la passerelle p_{1m}, de β_2 vers γ_m.

Il est facile de vérifier que $b \rightarrow_R^* d$ y $c \rightarrow_R^* d$. Donc, G est localement confluent, et comme le SC est noethérien, il est confluent (d'après le lemme de Newmann pour les systèmes de réécriture). **[]**

7.1.8. Algorithme de classification

Nous avons analysé la correction et la performance des algorithmes qui font la classification des instances dans un SC. Etant donné un SC, un algorithme de classification doit partir d'un $x \in X$ et produire comme résultat une S-classification pour x. La définition suivante concerne les algorithmes qui produisent une telle classification en utilisant seulement les fonctions de classification définies pour le SC.

Soit S(X) un SC et le graphe $G(ECA, .\rightarrow_R^* .)$.

- Pour $x \in X$, une **S-trace de classification** pour x, notée S(x), est un chemin de longueur maximal en G et dont l'état initial est <X,…,X>. Si S(x) est fini, alors S(x) finit en une S-classification (état terminal).
- Soit H un ensemble de S-traces : H est un **algorithme de classification** pour S, ssi $\forall x \in X$, il existe une S-trace $S(x) \in H$.
- Un **résultat** H(x) est une S-classification quelconque de x qui est un état terminal d'une S-trace $S(x) \in H$.
- Un algorithme de classification H est **déterministe** si pour chaque $x \in X$, il n'existe qu'une S-trace $S(x) \in H$.

Corollaire
La complexité d'un algorithme de classification H est le maximum des longueurs des traces qui appartiennent à H : complexité(H) = maximum{ | t | tq. $t \in H$, t est une trace}.

Preuve :

C'est une conséquence du Théorème 1. **[]**

Cette complexité est la complexité minimale possible. Dans la section suivante nous allons montrer que notre algorithme de classification a cette complexité.

Théorème 2

Pour un SC consistant et complet, un algorithme de classification ne produit qu'une seule S-classification.

Preuve :

C'est une conséquence du Théorème 1. □

Ce théorème garantit que le résultat de la classification d'une instance dans une base de connaissances TROPES est unique et ne dépend pas de l'algorithme de classification utilisé.

Théorème 3

Pour un SC consistant, il existe un algorithme de classification qui a pour résultat la S-classification qui serait obtenue pour tout algorithme de classification appliqué sur le SC complet correspondant.

Preuve :

L'algorithme cherché doit utiliser les passerelles dès qu'il peut le faire. Le choix d'utiliser les passerelles avant les autres transformations équivaut à simuler la complétion. □

Le résultat le plus important de cette formalisation est la convergence ; c'est-à-dire la garantie que tout algorithme de classification d'instances en TROPES donne un même résultat. Ainsi, le choix du point de vue (dernière étape de la classification) n'a aucune incidence sur le résultat de la classification. Cependant, le choix correct d'un point de vue peut modifier la performance de l'algorithme. De plus, lorsque la classification est menée par des spécialistes qui n'ont pas de connaissances complètes pour tous les points de vue, l'organisation la distinction entre points de vue principaux, auxiliaires et cachés permet d'inférer de grande quantité de connaissances avec la connaissance partielle de l'utilisateur. Nous avons aussi montré que, lorsque l'on travaille avec des instances complètes (ou pouvant être complétées par l'algorithme de classification ou par l'utilisateur lors de la classification), l'utilisation de passerelles n'entraîne pas un changement de la classification en elle-même.

7.2. Complexité de l'algorithme

Dans la section § 6.3 nous avons donné l'algorithme de classification multi-points de vue de TROPES et dans la section § 7.1 nous avons montré la correction de cet algorithme. Dans cette partie, nous allons faire l'analyse de la complexité de l'algorithme selon le modèle RAM de [AHO&74]. Comme nous l'avons montré, l'algorithme comporte, après une étape d'initialisation, une boucle composée de cinq parties : obtention d'information, appariement, propagation de marques, mise à jour de l'information et choix du prochain point de vue. Nous allons calculer la complexité de chacune de ces parties, puis la complexité de la boucle et enfin celle de l'algorithme tout entier.

À première vue, les procédures les plus coûteuses de la classification sont l'appariement et le parcours du graphe pour la propagation des marques. La complexité de la procédure d'appariement ainsi que de celle d'obtention d'information est déterminée par celle de la procédure de validation du type d'un attribut, qui vérifie si une valeur donnée **v** pour un attribut **a** satisfait un type **t** particulier de cet attribut. Par ailleurs, la propagation des marques fait un parcours des arbres de classification dont le coût est déterminé par la structure du concept (hauteur des arbres des points de vue, nombre de passerelles du concept, etc.).

Avant de faire le calcul de la complexité de chacun des pas de la boucle de classification, nous allons calculer la complexité de la validation du type d'un attribut**.**

7.2.1. Complexité de la validation du type d'un attribut

La validation du type d'un attribut comporte la validation de son domaine (normalisation des descripteurs *domaine* et *sauf*), et pour les attributs définis la validation de son descripteur de *type* normalisé. Pour les attributs multi-valués la validation inclut la vérification de la cardinalité, descripteur *card* normalisé. La complexité de la procédure de validation de type dépend, pour les attributs primitifs de celle de la procédure de validation du domaine, et pour les attributs définis de la validation du domaine et de la validation du descripteur *type*.

Validation du descripteur de type

Dans le chapitre précédent nous avons décrit la façon de normaliser et valider la satisfaction du descripteur de *type* pour un attribut défini (d'une instance composée) (§ 6.4). Par la suite nous allons calculer la complexité de cette procédure.

Le descripteur de type d'un attribut **att** est (après une normalisation faite par l'analyseur syntaxique du système) une disjonction des conjonctions de classes du concept de base de l'attribut. Cette disjonction de conjonctions est représentée sous forme d'une liste de listes de classes. Soit **b** le nombre des disjonctions de cette liste, alors pour faire la validation du type de **att** pour une valeur **v**, le système doit parcourir, dans le pire des cas, toute la liste de **b** listes-conjonction et regarder pour chaque liste-conjonction si ses classes sont des classes sûres pour **v**. Une liste-conjonction a au plus **p** classes (p étant le nombre de points de vue du concept). Ainsi, la complexité de la vérification du descripteur de type est

O(descr_type) = O(b * p * tt),

où **b** = lnombre de listes-conjonction

p = nombre de points de vue du concept

tt = coût pour vérifier si une instance appartient à une classe

Dans la plupart des cas le descripteur de type affine une seule classe dans un seul des points de vue du concept; dans ces cas la complexité est O(tt). Par la suite, nous allons calculer la complexité, tt, de vérification d'appartenance d'une instance **v** à une classe **C**.

Dans le chapitre précédent (§ 6.4.1) nous avons décrit la procédure de vérification de l'appartenance d'une instance à une classe. La première étape dans cette validation est l'obtention de la marque de la classe. Cette procédure a un coût de **O(h)** où h est la hauteur du point de vue (§ 7.2.4)[1].Si la classe a une marque impossible ou sûre, la validation termine. Si la classe a une marque possible alors le système doit valider la valeur de v par rapport au type de la classe C, c'est à dire valider la valeur de chaque attribut de v par rapport au type de cet attribut dans C.

La complexité pour vérifier si une instance satisfait un type peut être obtenue à partir de l'équation suivante:

descr_1_type (v) = h + a * valider_type (va)

[1]La complexitée présentée dans §7.2.4 inclut un facteur z correspondant au nombre de sous-arbres impossibles à l'intérieur de l'arbre actif. Cette information est conservée seulement pour le concept de l'instance à classer.

où **v** est la valeur de l'attribut défini de l'instance à classer, **a** est le nombre maximal d'attributs d'un concept et **h** est la hauteur d'un point de vue et **va** est la valeur d'un attribut de v.

La validation de type de **va** peut se limiter à la validation de domaine et de cardinalité (si l'attribut est primitif) ou bien elle peut faire appele à une nouvelle procédure de validation des attributs de **va** (si l'attribut est défini), puis des attributs des attributs de **va** et ainsi de suite. Dans cette séquence d'attributs chaque concept ne peut apparaître qu'une seule fois, alors la séquence a une taille maximale de **m**, où **m** est le nombre de concepts. Dans le pire de cas, tous les concepts apparaissent dans une séquence et à chaque niveau d'imbrication tous les attributs sont des attributs définis. La complexité dans ce cas est :

$$O(\text{descr\_1\_type}) = m * O(h) + O(a^m) * \text{fdom}$$

$$\mathbf{tt = O(descr\_1\_type) = O(a^m * fdom)}$$

où **a** =nombre maximal d'attributs d'un concept et

m = nombre des concepts de la base

fdom = coût pour valider la satisfaction du domaine de l'attribut

Lorsque l'attribut att de l'instance est un attribut multi-valué, cette vérification de type doit être faite dans le pire des cas pour tous les éléments de la multi-valeur. Si la classe n'impose pas de contraintes de cardinalité la valeur multiple peut comporter toutes les instances du concept. Dans ce très peu probable cas, la complexité est:

$$\mathbf{tt = O(descr\_1\_type) = O((s * a)^m * fdom)}$$

où **s** = nombre maximal d'instances d'un concept

a =nombre maximal d'attributs d'un concept et

m = nombre des concepts de la base

fdom = coût de valider la satisfaction du domaine d'un l'attribut

Validation du domaine

Par la suite nous allons calculer **fdom** c'est-à-dire la complexité de la validation des contraintes de domaine d'un attribut dans une classe. Pour faire le calcul de cette complexité nous allons expliquer, pour les différents types d'attributs et de concepts, la procédure de normalisation des descripteurs de domaine utilisée dans TROPES, puis la procédure de validation du domaine et enfin la complexité de cette validation.

Le domaine des valeurs possibles d'un attribut est déterminé, pour les attributs mono-valués par deux descripteurs (*dom* et *sauf*). Les attributs multi-valués utilisent aussi le descripteur *card* (§ 5.6.1). Par la suite nous allons présenter la structure d'un domaine normalisé et la fonction de validation pour des attributs mono-valués (des concepts ordonnés et non ordonnés), puis la normalisation et la validation des attributs multi-valués (aussi bien pour les concepts ordonnés que pour les concepts non ordonnés).

Attribut mono-valué prenant ses valeurs dans un concept ordonné

TROPES distingue deux catégories de concepts selon leur domaine de base : les concepts ordonnés et les concepts non-ordonnés. Lorsqu'un attribut prend ses valeurs dans un concept ordonné, ces valeurs sont comparables par une relation d'ordre total : c'est le cas du concept primitif *entier* . Le domaine de valeurs possibles (aussi bien le descripteur *dom* que le descripteur *sauf*) peut être décrit par des valeurs simples ou par des intervalles de valeurs. La normalisation d'un tel domaine consiste à créer les intervalles décrivant toutes les valeurs possibles pour l'attribut. Par exemple, l'attribut

Rendez_vous décrit dans (Fig. 7.12a) est normalisé par une liste d'intervalles ordonnés (Fig. 7.12b).

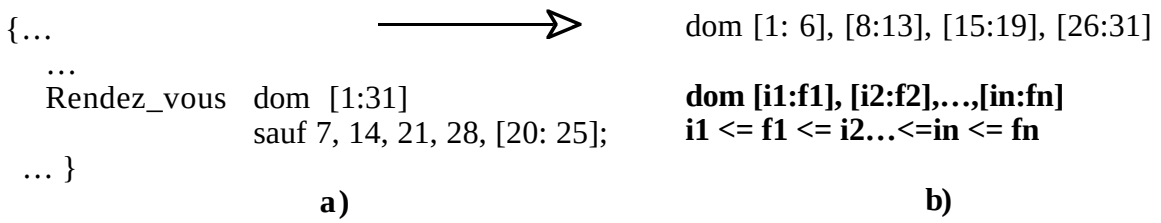


Fig. 7.12. Les descripteurs *dom* et *sauf* sont normalisées dans une liste d'intervalles ordonnés correspondant au même ensemble de valeurs

Vérifier qu'une valeur **v** est dans le domaine d'un attribut d'un concept ordonné consiste à trouver un intervalle [i :f] dans le domaine, tel que i <= v <= f. L'algorithme doit trouver dans cet ordre l'intervalle dont i <= v. Comme les intervalles sont ordonnés, cette recherche dans une liste ordonnée a un coût logarithmique en fonction du nombre d'intervalles. Dans ce cas, la fonction **valider** a une complexité de l'ordre :

O(dom_mono_ordonne) = $\log_2$(in) * k, k=constante

O(dom_mono_ordonne) = O($\log_2$(in))

où **in** = nombre d'intervalle composant le domaine normalisé

Attribut mono-valué prenant ses valeurs dans un concept non ordonné

Pour les concepts non_ordonnés, le domaine est donné explicitement par l'ensemble des valeurs permises et l'ensemble des valeurs exclues. De plus, la valeur TOUT permet de ne pas avoir à donner explicitement toutes les valeurs du domaine. La normalisation consiste à enlever du domaine les éléments décrits par le descripteur sauf (Fig. 7.13)

L'algorithme de **validation** doit comparer ici, dans le pire des cas, la valeur de l'instance avec toutes les valeurs de l'ensemble donné (qu'il représente les valeurs valides ou les valeurs exclues). Donc,

O(dom_mono_non_ordonne) = O(t)

où **t** = nombre d'éléments (explicitement permis ou exclus) du domaine

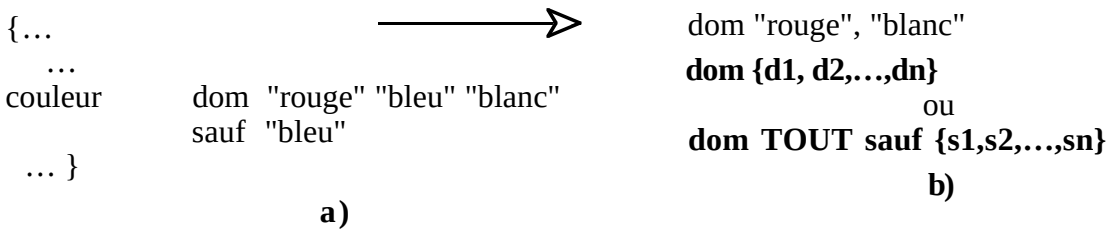


Fig. 7.13. Pour les attributs prenant des valeurs dans les concepts non ordonnés, la normalisation construit l'ensemble des valeurs permises.

Attribut multi-valué prenant ses valeurs dans un concept ordonné

Les cas précédents correspondent aux attributs mono-valués. Pour les attributs multi-valués, l'ensemble de multi-valeurs (listes de valeurs) du type peut être donné, soit par sa cardinalité (taille maximale d'une de ces multi-valeurs) et par l'ensemble de valeurs de base qui peuvent être présentes dans une de ces listes, soit par la présentation explicite des valeurs multiples permises et exclues, ou encore par une combinaison des deux.

- Dans le premier cas, le domaine est exprimé sous la même forme que pour les attributs mono-valués. Vérifier qu'une valeur multiple donnée peut être construite à partir d'un domaine de valeurs de base consiste, d'une part, à vérifier que la valeur multiple a une

taille autorisée, et, d'autre part, à vérifier, pour chacune de ses valeurs simples, qu'elle appartient au domaine de base.

- Pour les concepts ordonnés, la valeur **v** est une multi-valeur dont les éléments sont ordonnés. Il suffit donc de parcourir la liste d'intervalles une seule fois pour déterminer l'appartenance de ces éléments au domaine; de plus la taille de la multi-valeur se calcule en même temps, ce qui donne une complexité constante à la vérification de la cardinalité. Ainsi, la complexité obtenue est :

  O (dom_multi_ord) = O (in)

  où **in** = nombre d'intervalles composant le domaine normalisé.

- Pour les concepts non_ordonnés, le système doit vérifier, pour chaque élément de la multi-valeur, s'il est dans le domaine donné, ce qui coûte :

  O (dom_multi_non_ord) = O (t * c)

  où **t** = nombre d'éléments, explicitement permis ou exclus du domaine de base ;

  c = taille maximale permise pour une multi-valeur de l'attribut (descripteur card).

• Dans le deuxième cas, le domaine, après normalisation, est l'ensemble explicite des valeurs multiples permises ; soit **l** le nombre d'éléments de cet ensemble et **c** la cardinalité maximale permise pour une valeur multiple.

- Si l'attribut prend ses valeurs dans un concept ordonné, chaque multi-valeur a ses éléments ordonnés. Ainsi, comparer la valeur **v** avec une multi-valeur du domaine revient à parcourir en ordre les deux listes, donc à faire c comparaisons. Ce processus doit se répéter pour les l listes :

  O (dom_multi_ord) = O (l * c)

  où **l** = nombre de multi-valeurs du domaine ;

  c = taille maximale permise pour une multi-valeur de l'attribut (descripteur card).

- Si l'attribut prend ses valeurs dans un concept non_ordonné, pour comparer **v** avec une autre multi-valeur, le système doit chercher chaque élément de v dans la liste décrivant l'autre multi-valeur, ce qui dans le pire des cas peut coûter c * c. Cette comparaison se fait, dans le pire des cas **l** fois. Ainsi, comparer la valeur **v** avec une multi-valeur du domaine, revient à parcourir en ordre les deux, donc faire c comparaisons, où c est la taille maximale des deux listes (c'est-à-dire la taille maximale permise). Ce processus doit se répéter pour toutes les listes :

  O (dom_multi_non_ord) = O (l * c * c)

  où **l** = nombre de multi_valeurs du domaine,

  c = taille maximale permise pour une multi-valeur de l'attribut (descripteur card).

• Le troisième cas, qui serait le plus coûteux, n'est pas pris en compte car il ne permet pas, à un prix raisonnable, de faire une normalisation unique du domaine. Par exemple, les descriptions suivantes (Fig. 7.14) de la multi-valeur m donnent le même ensemble mais leur normalisation demanderait la génération de l'ensemble explicite de valeurs.

```
{                                   {
  m  dom  [4 : 5] {4,5,6} {6}        m  dom [4 : 6]
                                        sauf  {5,6},{4
}                                   }
```

Fig. 7.14. Deux schémas de description d'un attribut qui correspondent au même type, les multi-valeurs : {4}, {5}, {6}, {4,5}, {4,5,6}.

TROPES gère les deux premiers types de descriptions de domaines multi-valués présentés précédemment; ainsi, la complexité de la validation d'un type pour une valeur

donnée lorsque celle-ci est une multi-valeur, est la complexité maximale de ces deux formes de description. En résumé, la complexité de la fonction **valider** est la somme de la complexité de la vérification de la classe de la valeur et du coût maximal de la vérification du domaine :

COMPLEXITÉ POUR VALIDER UN TYPE POUR UNE VALEUR D'ATTRIBUT

fa = O(valider) = fat + fdom

où **fdom** = **max (dmo, dmno, dmmo, dmmno)**,

fat = O (valider descripteur type) = O (b * p * tt),

tt = O (descr_1_type) = O (a^m * fdom)

dmo = O (dom_mono_ordonne) = O(log(in)),

dmno = O (dom_mono_non_ordonne) = O (t),

dmmo = O (dom_multi_ord) = maximum (O (in),O (l * c))

dmmno= O (dom_multi_non_ord) = maximum (O (t * c),O (l * c * c)),

où

b = nombre de disjonctions des classes d'appartenance pour l'attribut

p = nombre des points de vue du concept de l'attribut

a = nombre d'attributs d'un concept

m = nombre de concepts de la base

in = nombre d'intervalles composant le domaine ordonné

t = nombre d'éléments permis (ou exclus) du domaine de base

l = nombre de multi-valeurs du domaine

c = taille maximale permise pour une multi-valeur (descripteur card)

Pour la classification d'instances simples, **fat = 0**. Pour la classification d'instances composées, dans le cas moyen, le descripteur type d'un attribut comporte une seule classe d'appartenance pour les instances qui valident cet attribut, auquel cas **fat = O (tt)**.

Si l'on suppose que les attributs multi-valués sont décrits en général par le domaine de base (sauf quand le nombre de multi_valeurs est très faible), alors on a, pour les attributs prenant leurs valeurs dans un concept ordonné T , une complexité qui peut être généralisée à :

fa = O (tt + in)

où **tt** = coût de vérification de l'appartenance d'une instance à une classe

i**n** = nombre d'intervalles du domaine

Pour les attributs de concepts non ordonnés, une complexité de :

fa = O (tt + t * c)

où

tt = coût de vérification de l'appartenance d'une instance à une classe

t = nombre d'éléments permis (ou exclus) du domaine de base

c = cardinalité maximale pour une multi-valeur

Enfin, pour les attributs prenant des valeurs dans les concepts primitifs, tt = s, où s est le bas coût d'une vérification simple et la complexité est O(in) et O(t * c) respectivement.

Par la suite nous allons faire l'analyse de la complexité de la procédure de classification en calculant la complexité pour chacun de ses pas.

7.2.2. Complexité de la procédure d'obtention d'information

Dans la procédure d'obtention d'information, le système complète l'information de la classe actuelle en demandant à l'utilisateur des valeurs d'attribut pour les attributs non valués de l'instance. La valeur d'un attribut doit satisfaire le type de cet attribut dans la classe.

L'algorithme d'obtention d'information est :

pour_tout ai de C-actuelle tel que v(ai) = ⊥

-> demander (a_i, $ta_{c\text{-actuelle}}$)

valider (v_i, a_i, $ta_{c\text{-actuelle}}$)

Le nombre maximal d'attributs à demander est le nombre total d'attributs de la classe **ac**. Si **a** est le nombre total d'attributs du concept, alors **ac** <= **a**, et la complexité de la procédure d'obtention d'information est :

COMPLEXITÉ DE LA PROCÉDURE D'OBTENTION D'INFORMATION

O(d'obtention) = O (a * fa).

où **a** = nombre d'attributs du concept

fa = complexité de valider un type pour une valeur d'attribut

7.2.3. Complexité de la procédure d'appariement

La procédure d'appariement fait le choix d'une classe sûre parmi les sous-classes {S1,S2,…,Sn} de la classe actuelle. Pour faire ce choix, la procédure demande à l'utilisateur les valeurs inconnues des attributs de ces sous-classes. Lorsque l'utilisateur donne une valeur pour un attribut, le système le valide par rapport à chacun des types de cet attribut dans les classes de l'ensemble {S1,S2,…,Sn} où il apparaît. La fonction de validation est la même fonction valider de la procédure d'obtention d'information qui a un coût maximal **fa**.

Comme nous l'avons montré dans § 6.3.4, la procédure d'appariement d'information cherche à faire le plus grand nombre d'inférences avant de poser des questions à

l'utilisateur, puis ordonne ces questions de façon à optimiser les inférences tirées des réponses.

Le premier raccourci de la procédure consiste à marquer les **n¡** classes pour lesquelles tous les attributs sont valués dans l'instance (ni <= n, où **n** est le nombre maximal de sous-classes d'une classe du concept). Pour marquer chacune de ces classes, le système détermine si l'instance satisfait son type (si les valeurs de ses attributs satisfont les types correspondant dans la classe). Dans le pire des cas, la procédure vérifie les **a** attributs du concept pour chacune de ces **ni** classes, ce qui donne une complexité de :

O (ni * a * fa).

Pour les classes qui n'ont pas encore été marquées **(n - ni)**, la procédure d'appariement va chercher à compléter l'information en demandant à l'utilisateur les valeurs d'attributs manquantes. Le deuxième raccourci de l'algorithme consiste à choisir les attributs à demander dans le bon ordre. Pour cela, la procédure construit une structure temporaire ayant, pour toute classe, le nombre d'attributs inconnus de l'utilisateur et le nombre d'attributs non valués. Le remplissage de cette structure demande le parcours de tous les attributs de toutes les classes non marquées, ce qui coûte :

O (a * (n-ni)).

Après ces étapes préliminaires, le système commence à poser les questions à l'utilisateur en faisant les pas suivants :

1. Prendre la plus petite classe non-marquée C_{min}. C_{min} est, parmi les classes ayant le plus petit nombre d'attributs inconnus, celle ayant le plus petit nombre d'attributs non valués. Cette classe est trouvée en parcourant la structure temporaire, ce qui coûte **O (n-ni)**
2. Demander la valeur du premier attribut non valué de Cmin et valider cette valeur par rapport aux types des classes où il apparaît (dans le pire des cas, toutes les **ni** classes font référence à cet attribut), ce qui a une complexité de **O ((n-ni) * fa)**. Au fur et à mesure qu'il fait ces validations, le programme mémorise dans la structure temporaire le nombre d'attributs qui restent à valuer dans chaque classe, ainsi que les marques pour les classes n'ayant plus d'attributs inconnus (ces deux processus ont une complexité constante).
3. Une fois le premier attribut valué et validé, le système cherche le prochain attribut non valué de la même classe. S'il n'en reste plus, il recalcule la plus petite classe non marquée en révisant les éléments de la structure temporaire : **O (n-ni)**, puis il prend son premier attribut non valué et demande sa valeur à l'utilisateur.

Ainsi, les pas 1,2,3 suivis pour valider un attribut du concept ont une complexité de :

O ((n-ni) + (n-ni) + ((n-ni) * fa))

Dans le pire des cas, le système doit faire cette suite de pas pour toutes les classes, pour tous les attributs du concept, ce qui donne :

O(a * (2 * (n-ni) + (n-ni) * fa)) = O(a * (n-ni) + a * (n-ni) * fa).

La complexité de la procédure d'appariement est la somme de la complexité des étapes préliminaires et la complexité du cours de l'appariement :

$$O(\text{appariement}) = O(ni * a * fa + a * (n-ni) + a * (n-ni) * fa)$$
$$= O(n * a * fa + 2a * (n-ni))$$
$$= O(a * n * (fa + 1))$$
$$= O(a * n * fa)$$

COMPLEXITÉ DE LA PROCÉDURE D'APPARIEMENT
O(appariement) = O (n * a * fa).
où **n** = nombre maximal de sous-classes directes d'une classe
a = nombre d'attributs du concept
fa = complexité de valider un type pour une valeur d'attribut

7.2.4. Complexité de la procédure de propagation de marques

Pour le calcul de la complexité de la procédure de propagation de marques, nous allons prendre en compte les paramètres suivants :

j = niveau dans le point de vue de la nouvelle classe actuelle.

h = hauteur maximal de l'arbre de classes (point de vue).

pa = nombre de passerelles actives du concept

z = nombre maximal d'arbres impossibles, sous-arbres de l'arbre dont la racine est la plus petite classe sûre d'un point de vue (taille de L_imp Fig. 7.15)

p = nombre de points de vue du concept.

La procédure de propagation de marque entraîne des propagations à l'intérieur d'un point de vue et des propagations entre différents points de vues. L'algorithme de propagation, tel qu'il a été présenté dans § 6.3.5 marque explicitement toutes les classes impossibles et possibles des différents points de vue. Nous avons fait une amélioration à la procédure de propagation de marques, pour éviter ce marquage exhaustif des classes. Pour cela, nous avons défini des structures de représentation simples permettant de récupérer et de mettre à jour toute l'information des étiquettes d'un point de vue lors d'une classification.

Dans un état stable de la classification, tout point de vue a une plus petite classe sûre pour l'instance, appelé C_min (Fig. 7.15a)). À partir de C_min, le système peut récupérer toutes les classes sûres de l'instance pour ce point de vue : ce sont toutes les sur-classes de C_min (Fig. 7.15.b)). De plus, les seules classes qui "peuvent" ne pas être marquées sont les sous-classes de C_min. Toutes les classes sœurs d'une classe sûre et toutes les sous-classes de celles-ci sont des classes impossibles (Fig. 7.15c)).

Bien qu'en principe toutes les sous-classes de la plus petite classe sûre sont des classes non marquées, il peut y avoir, dans ce sous-arbre, des classes pour lesquelles on ait déduit la marque impossible par une passerelle; les sous-classes de celles-ci ont, elles aussi, la marque impossible (Fig. 7.15d)).

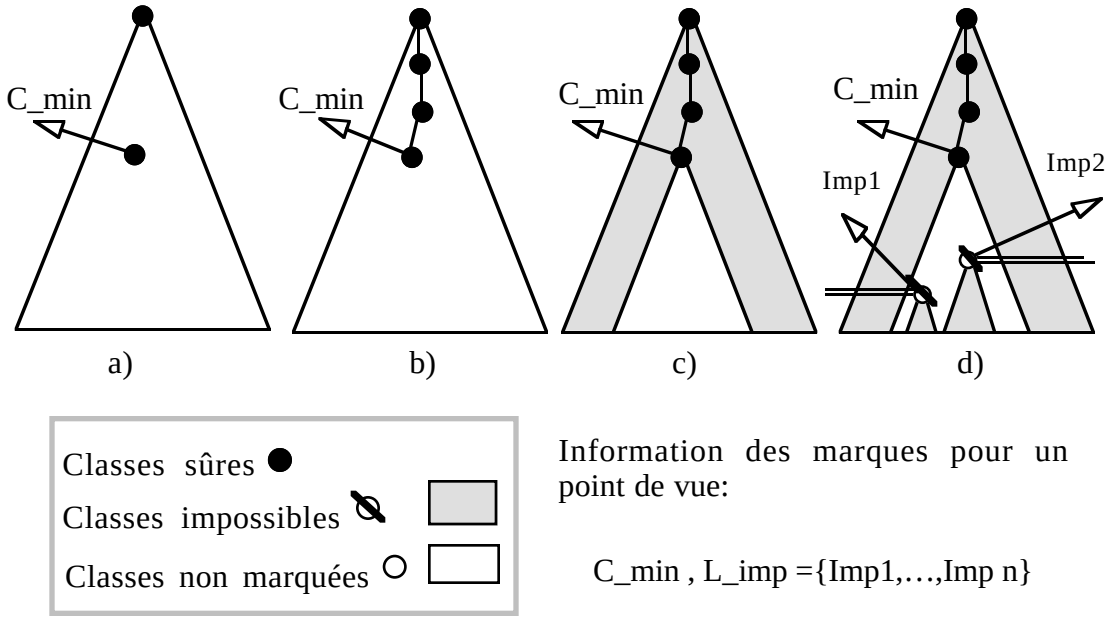


Fig. 7.15. Avec l'information de la plus petite classe sûre C_min pour l'instance et les racines des sous-arbres impossibles de C_min, le système peut retrouver la marque correcte de toute classe du point de vue.

Ainsi, à partir de la plus petite classe sûre d'un point de vue **C_min** et de la liste des racines des arbres impossibles qui sont sous-arbres de C_min, **L_imp**, on peut retrouver la marque de toute autre classe du point de vue.

En effet, la fonction marque(C, C_min, L_imp) peut être définie par :

marque(C, C_min, L_imp) =

si sur_classe_egal (C, C_min) -> SURE
sinon si ∃ i, i ∈ L_imp tel que sur_classe_egal (i,C) -> IMP
sinon si sur_classe_str (C_min, C) -> *non marqué*
sinon -> IMP

où

- sur_classe_egal (C,S) est un prédicat qui rend TRUE si S<=C
- sur_classe_str (C,S) est un prédicat qui rend TRUE si S<=C et S ≠ C

Pour résoudre chacun de ces prédicats il faut remonter par les liens des sur-classes de S jusqu'à C ou jusqu'à la racine.

Ainsi, si

j = niveau dans le point de vue de la nouvelle classe actuelle.

h = hauteur maximale de l'arbre de classes (point de vue).

z = nombre maximal de sous-arbres de C_min marqués impossibles

alors

° sur_classe_egal (C,C_min) doit parcourir au maximum **j** niveaux;

° ∃ i, i ∈ L_imp tel que sur_classe_egal (i,C). Pour résoudre cette partie, le système examine C puis toutes ses sur-classes (au plus **h** classes). Pour chacune de ces classes, le programme vérifie qu'elle appartient à l'ensemble L_imp, ce qui demande de regarder, au maximum, toute la liste L_imp, donc ses **z** éléments.

La liste L_imp est formée des racines des sous-arbres impossibles de C_min. Un tel sous-arbre peut exister seulement s'il y a eu une passerelle, déjà activée, qui a marqué sa racine comme étant impossible. Ainsi, le nombre **z** d'éléments de L_imp est limité

par le nombre de passerelles du concept dans lequel intervient un même point de vue et donc limité par **le** nombre de passerelles du concept, **pa**.

° sur_classe_str (C_min,C) doit parcourir le chemin entre C et C_min ou, dans le pire des cas, le chemin entre C et la racine, donc **h** niveaux.

Ainsi, on a :

O (marque) = maximum (j, h * z, h)

O(marque) = m = **O (h * z)** où **z** est le nombre de sous-arbres impossibles de C_min

La procédure de propagation de marques doit garantir que pour chaque point de vue du concept, ces deux éléments C_min et L_imp sont bien mis à jour pour prendre en compte les nouvelles marques (pour que la fonction marque rende la bonne réponse pour toute classe du concept). Nous allons montrer par la suite que toutes les propagations de marques à l'intérieur d'un point de vue et entre points de vue peuvent être faites en modifiant correctement les C_min et L_imp des différents points de vue et en mettant à jour les passerelles actives du concept.

Propagation à l'intérieur d'un point de vue

Lorsque dans un point de vue, une nouvelle classe C est marquée comme **sûre**, elle devient la nouvelle plus petite classe sûre de ce point de vue. En effet, l'algorithme ne peut marquer comme sûres que les classes non marquées, les marques impossibles étant non modifiables, si C était non marquée, alors elle serait nécessairement plus petite que la plus petite classe sûre courante. D'après l'analyse faite dans la section précédente, il suffit de mettre à jour la valeur de C_min pour que la fonction marque rende pour toutes ses sur-classes la marque sûre (règle de propagation 1), pour toutes les sœurs des classes sûres la marque imp (règle de propagation 2) et pour toute sous-classe d'une classe impossible la marque imp.

C_min := C coût constant.

Ainsi :

O (prop 1 PV) = O (1)

Propagation par les passerelles

L'algorithme de propagation doit prendre en compte les passerelles actives du concept. La liste de passerelles actives comporte, pour chaque passerelle, la liste de ses sources et la classe destination. Le nombre maximal de classes intervenant dans une passerelle (nombre de sources + 1) est **p** (nombre de points de vue).

Pour faire la propagation de marques, l'algorithme parcourt toutes les passerelles actives **pa** et pour chaque passerelle, il exécute les pas suivants (Fig. 7.16) :

- Calculer les marques courantes de chacune des classes intervenant dans la passerelle.

  Au maximum il doit calculer **p** marques. Alors la complexité de ce calcul est **p * h * z .**
- Vérifier que la combinaison des marques de la passerelle permet de déduire la marque d'une de ses classes (non marquée) et mettre à jour les structures C_min et L_imp du point de vue de cette classe pour prendre en compte sa nouvelle marque.

  Les deux cas qui déclenchent une propagation sont (règles 4, 5 et 6 de § 6.3.5) :

° Toutes les sources de la passerelle ont la marque sûre, alors la destination D/PVx est marquée sûre. Marquer comme sûre la destination D/PVx revient à mettre à jour la classe C_min de PVx (C_min := D). Cette action a un coût constant, **k** (Fig. 7.16a)).

° Toutes les sources sauf une, C/PVy (non marquée), ont la marque sûre et la destination a la marque impossible, alors C/PVy est marquée impossible, ce qui revient à l'ajouter à la liste des racines des sous-arbres impossibles de C_min de son point de vue, L_imp/PVy. Cet ajout a un coût constant, **k** (Fig. 7.16b).

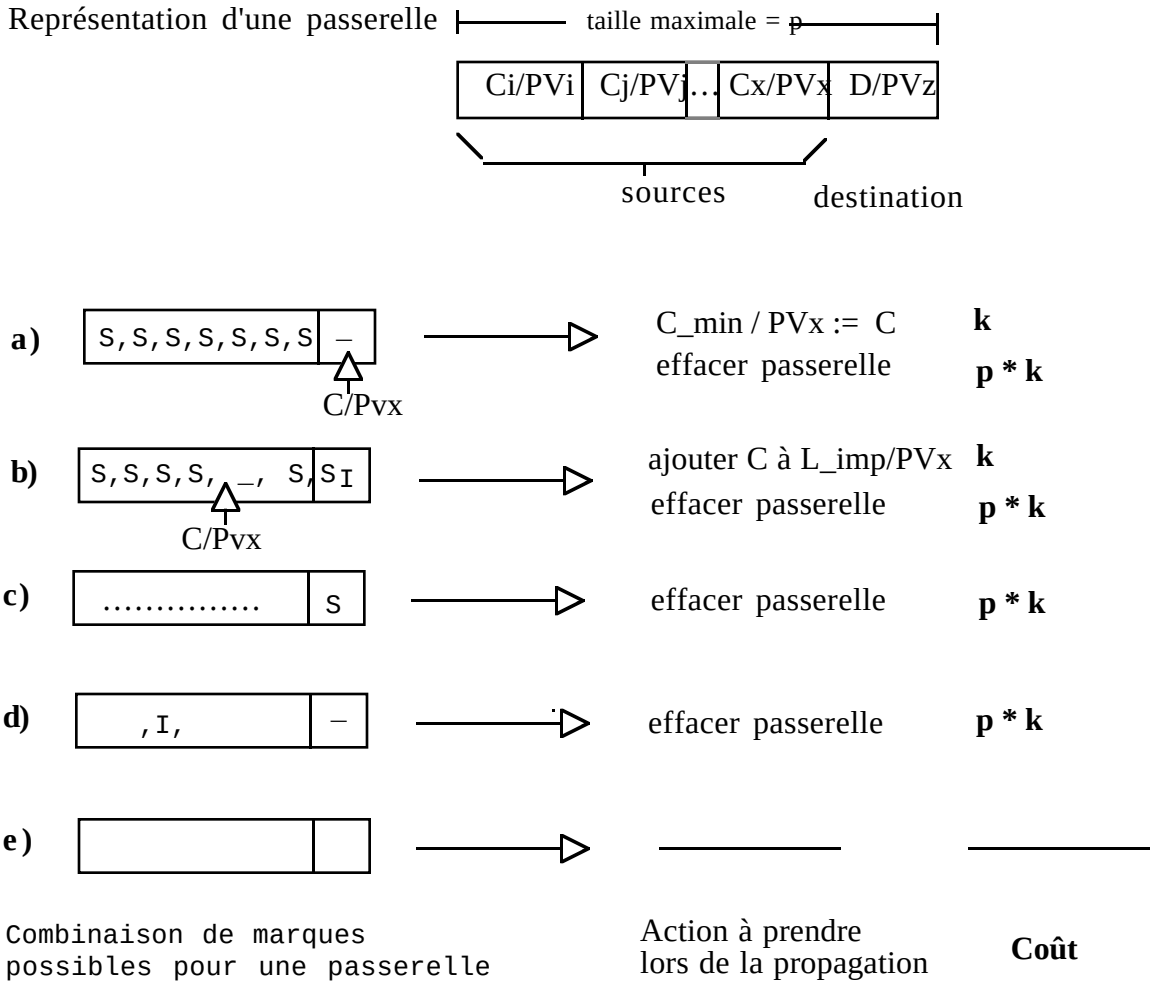


Fig. 7.16. Pour chaque passerelle, l'algorithme calcule les marques de ses classes, regarde si la passerelle peut produire une nouvelle marque, propage cette marque, puis efface la passerelle. Il efface aussi les passerelles qui ne permettrons jamais de déduire des marques.

- Si la passerelle ne sert plus à déduire des marques, la procédure l'efface. Une passerelle ne sert plus à déduire des marques si toutes ses classes sont marquées (c'est le cas si la combinaison des marques correspondait à l'un de deux types précédents), ou bien si la destination de la passerelle est marquée sûre ou une de ses sources est marquée impossible. Dans ces quatre cas (qui sont identifiés lors du marquage) la passerelle doit être effacée.

  Avant d'effacer une passerelle, la procédure met à jour l'information du nombre de passerelles actives de chaque point de vue intervenant dans la passerelle et l'état du point de vue :

  pour tout PV de la passerelle -> Npass/PV := Npass/PV - 1;
  si Npass/PV = 0 -> état.PV := inactif

Le coût de cette opération est **p * k**

Le marquage, la propagation et la mise à jour d'une passerelle ont une complexité de :

O(une_passerelle) = O (p * h * z) + maximum (k+ p, k + p, p, p)

= O (p * h * z) + O (p)

O (une_passerelle) = O (p * h * z)

La mise à jour de toutes les passerelles actives est au maximum :

O (passerelles) = O (pa * (p * h * z))

La propagation des marques par les passerelles continue tant qu'il y a des passerelles actives qui permettent d'inférer de nouvelles marques. Ainsi, la procédure de révision, de propagation et de désactivation des passerelles continue jusqu'à ce qu'il n'y ait plus de passerelles actives ou jusqu'à ce qu'il n'y ait plus de changement dans la configuration de marques (un parcours complet de la liste de passerelles ne modifie les marques d'aucune passerelle).

Ainsi, l'algorithme de propagation de passerelles parcourt une première fois toutes les passerelles actives en regardant celles qui peuvent être activées **O (passerelles)**. S'il n'y a pas eu de passerelles actives, la procédure s'arrête. S'il y a eu au moins une passerelle P activée, la procédure parcours une deuxième fois toutes celles qui restent (au plus pa - 1), car l'activation de P a pu entraîner des modifications sur les marques d'autres passerelles.

° Dans le cas extrême où l'activation d'une passerelle entraîne en cascade celle de toutes les autres passerelles actives, l'algorithme parcourt **pa** fois les passerelles actives. Donc il fait (pa * (pa-1) / 2) révisions de passerelles, ce qui coûte :

O (prop_toute_pass) = O (pa * passerelles)

O (prop_toute_pass) = O (pa * (pa * p * h * z))

° La situation extrême opposée est produit lorsqu'aucune passerelle n'est activée lors du premier parcours de la liste de passerelles actives. Dans ce cas, le parcours des passerelles n'est fait qu'une fois avec un coût :

O (prop_no_pass) = O (pa * p * h * z)

Le premier de ces deux cas entraîne une première propagation des passerelles assez coûteuse (de l'ordre de pa^2), mais ce coût est compensé au niveau de l'algorithme total de classification par le fait qu'après cette première propagation il n'y a plus de passerelles actives, ce qui entraîne un taux constant pour les itérations suivantes du cycle de classification, dans cette étape (§ 7.2.7).

Donc, on peut dire déjà que le coût total de la propagation de marques pour toute la classification est, dans ce cas extrême :

O (prop_marques_totale) = O (pa * (pa * p * h * z))

L'autre extrême se présente lorsque toutes les passerelles lient des feuilles des points de vue. Ainsi, seront déclenchées seulement à la fin. À chaque itération de la boucle, la procédure parcourt toutes les **pa** passerelles une seule fois (car aucune n'est activable). Ce parcour va se faire tout au long de la classification; en fait celle-ci ne va pas tirer parti des passerelles pour accélérer la descente. Nous allons montrer plus loin (§ 7.2.7) que dans ce cas le nombre maximal d'itérations de la boucle de classification est p * h car les points de vue peuvent être vus comme des arbres de classification indépendants, et pour chacun d'eux, l'instance descend de h niveaux (de la racine aux feuilles). Donc, pour ce deuxième cas la propagation totale de marques coûte :

O (prop_marques_totale) = O (p * h * (pa * p * h * z))

Enfin, la procédure de propagation de marques est aussi chargée de désactiver les points de vue principaux dont la plus petite classe sûre est une classe terminale. Pour cela elle parcourt la liste des points de vue principaux actifs Pai et pour chacun elle vérifie que

- C_min/Pai est une feuille (elle n'a pas de sous-classes) : coût constant
- si les sous-classes de C_min sont dans la liste L_imp /Pai : coût **n * z**, **n** étant le nombre maximal de sous-classes d'une classe et **z** la taille maximale de la liste L_imp

Ainsi la complexité de mettre à jour l'état actif ou inactif des points de vue est :

O (état_PV) = O (n * z)

La complexité totale de la procédure de propagation de marques est donc :

♦ Pour le premier cas de propagation

O (propagation_toutes) = O (prop 1 PV) + O (prop_tout_pass) + O (état_PV)
= O (1) + O (pa * (pa * p * h * z)) + O (n * z)

Comme on peut supposer que p * h * pa > n, on a :

O (propagation_toutes) = O (pa * (pa * p * h * z))

♦ Pour le deuxième cas de la propagation

O (propagation_non) = O (prop 1 PV) + O (prop_no_pass) + O (état_PV)
O (propagation_non) = O (1) + O (pa * p * h * z) + O (n * z)

Comme on peut supposer que p * h * pa > n, on a :

O (propagation_non) = O (pa * p * h * z)

COMPLEXITÉ DE LA PROCÉDURE DE PROPAGATION DE MARQUES

O (propagation) = **O (propagation toutes)** ou
= **O (propagation_non)**

O (propagation_toutes) = O (pa^2 * p * h * z)

O (propagation_non) = O (pa * p * h * z)

où

pa = nombre de passerelles actives du concept.

n = nombre maximal de sous-classes directes d'une classe

h = hauteur maximale de l'arbre de classes (point de vue).

z = nombre de sous_arbres IMP de C_min

p = nombre de points de vue du concept.

7.2.5. Complexité de la procédure de mise à jour

Le quatrième pas de l'algorithme de classification est la procédure de mise à jour de l'information liée à l'instance par rapport à sa nouvelle classe d'appartenance. La complexité de cette procédure est le coût de l'intersection entre deux types, t1 et t2. En effet, cette procédure calcule pour tous les attributs présents dans le schéma de la classe actuelle C_actuelle l'intersection de son type avec le type conservé dans le I-moule pour cet attribut.

Dans la section § 7.2.1, nous avons montré la normalisation du type d'un attribut. Le calcul d'intersection de types part des types normalisés. Ainsi, pour un attribut **a** qui prend des valeurs dans un concept T on a :

- descripteur type :

si T est un concept primitif, le type dans le descripteur type ne change pas. Si le concept est un concept défini, alors l'intersection des descripteurs type de deux types consiste à garder la plus petite classe sûre des deux (en supposant que chaque type ne corresponde qu'à une classe). Les systèmes étant cohérents, ces classes doivent être comparables, alors la complexité de ce calcul est **O (h)** où h est la hauteur maximale des points de vue du concept T.

- descripteurs dom et sauf :

- si le concept T est ordonné[1], alors calculer l'intersection des domaines de deux types est la conformation d'une liste ordonnée d'intervalles à partir de deux listes ordonnées d'intervalles. La complexité de cette procédure est **O(in)** où **in** est le nombre d'intervalles du type ayant le plus grand nombre d'intervalles dans son domaine.
- Si le concept n'est pas ordonné, alors l'intersection, dans le cas le plus coûteux (pour un attribut multi-valué) consiste à créer l'ensemble intersection des éléments des ensembles des deux types, ce qui revient à vérifier, pour chaque élément d'un ensemble, s'il est présent dans l'autre. Cette opération est assez coûteuse: si **c** est la cardinalité maximale la plus petite de deux types pour un attribut multi-valué, **t1** le nombre d'éléments (permis ou exclus) d'un des types et **t2** le nombre d'éléments permis ou exclus de l'autre type, alors la complexité dans le pire des cas est **O (t1 * t2 * c)**.

Ainsi, la complexité de la mise à jour d'information pour une nouvelle classe sûre d'un point de vue est : **O (a * max (in, t1 * t2 * c))**, où **a** est le nombre d'attributs du concept. Cette mise à jour est faite pour tous les points de vue pour lesquels la propagation a produit une nouvelle classe sûre. Dans le cas extrême, le nombre total de points de vue ayant une nouvelle classe sûre est **p-1**, **p** étant le nombre total de points de vue du concept. La complexité totale de la procédure de mise à jour est :

COMPLEXITÉ DE LA MISE À JOUR D'INFORMATION

O (mise_a_jour) = O (p * a * ft)

ft = **max (in, t1 * t2 * c)**

Où **p** = nombre des points de vue du concept

a = nombre d'attributs du concept

in = nombre maximal d'intervalles du domaine (de T1 et T2)

t 1= taille du domaine de base (de T1)

t 2= taille du domaine de base (de T2)

c = min (card maximale de T1 , card maximale de T2)

[1]Nous supposons que pour les attributs multi-valués le domaine est donné à partir du domaine de base (option 2 de §7.2.1. normalisation et validation du domaine).

7.2.6. Complexité de la procédure du choix du prochain point de vue

Le dernier pas de l'algorithme séquentiel de classification concerne le choix du prochain point de vue à considérer . Le critère à considérer pour ce choix est de prendre le premier point de vue principal actif. S'il n'y a pas de points de vue principaux actifs, l'algorithme prend le premier point de vue auxiliaire actif. Comme la procédure de propagation de marques met à jour la liste des points de vue principales actifs L_Pp et la liste des points de vue auxiliaires actifs L_pa, le choix du prochain point de vue est :

si L_Pp = {PVpi,…} -> PVi
sinon si L_Pa = {PVax,…} -> PVax
sinon -> fini := TRUE (il n'y a plus de points de vue actifs)
fsi
si ∀ Pv, I.C/PV == C_min /PV -> fini := TRUE (il n'y a plus de classes ouvertes)
si interruption_finir(utilisateur) -> finir := TRUE

Cette procédure a une complexité O (1) + O (p) où **p** est le nombre de points de vue du concept:

COMPLEXITÉ DU CHOIX DU PROCHAIN POINT DE VUE
O (choix du prochain point de vue) = O (p)
où **p** = nombre des points de vue du concept

7.2.7. Complexité totale de l'algorithme de classification

Après avoir calculé la complexité de chacun des pas de la boucle de classification, nous terminons par le calcul de la complexité totale de l'algorithme de classification :

construire_état_initial
tant_que not (fini)
obtenir_information
faire_appariement
propager_marques
mettre_à_jour_l_information
choisir_prochain_point_de_vue
fin_tant_que

Soient
p = nombre des points de vue du concept
h = hauteur maximale des points de vue du concept
n = nombre maximal de sous-classes directes d'une classe
z = nombre maximal de sous-arbres de C_min marqués impossible
a = nombre d'attributs d'un concept
fa = coût pour valider un type pour une valeur d'attribut
pa = nombre de passerelles du concept.
ft = coût pour calculer l'intersection de deux types d'un attribut

La complexité de chacun des pas est reportée dans le cycle de classification :

tant que not (fini)

O(obtention d'information) = O (a * fa)

O(appariement) = O (n ∗ a * fa).

O(propagation de marques) = O (pa^2 * p * h * z) ou

O (pa * p * h * z)

O(mise à jour de l'information) = O (p * a * ft)

O(choix du point de vue) = O (p)

fin_tant_que

La complexité de l'algorithme total de classification peut être calculée à partir de ses deux comportements extrêmes de propagation des passerelles.

Propagation de toutes les passerelles dès le début

Si les passerelles sont toutes activées lors de la première itération de la classification, alors la procédure de propagation de marques coûte dans sa première itération O (pa^2 * p * h * z) alors que son coût est constant dans les itérations suivantes (§ 7.2.3). Le résultat final de la complexité de l'algorithme de classification d'une instance dans tous les points de vue du concept est, dans ce cas :

$$O(\text{classification}) = O\ (pa^2*p*h*z) + \sum_{j=0}^{h} p*(((n+1)*a*f_a)+(p*a*ft)+1+p)$$

O(classification) = O (pa^2 *p*h*z) + (p*h* ((n*a*f_a) + (p*a*ft) +1 +p))

O(classification) = O (p * h * ((n * a * f_a) + (p * a * ft) + (pa^2 * z)+p))

où

p = nombre des points de vue du concept

h = hauteur maximale des points de vue du concept

n = nombre maximal de sous-classes directes d'une classe

a = nombre d'attributs d'un concept

fa = coût pour valider le type pour une valeur d'attribut

ft = coût pour valider l'intersection de deux types T1, T2

pa = nombre de passerelles du concept.

z = nombre maximal de sous-arbres de C_min marqués impossible

Propagation de toutes les passerelles à la fin

Si toutes les passerelles unissent des classes feuilles des points de vue, alors la propagation se fait pour toutes ces passerelles à la fin. Dans chaque itération, l'algorithme de propagation de marques fait UN parcours de la liste de passerelles actives. Dans ce cas, la classification ne tire pas parti des passerelles, et elle se déroule par une descente (dans le cas extrême, l'instance est descendue de la racine jusqu'à une feuille, donc h niveaux) dans chacun des points de vue du concept (p descentes). La complexité de l'algorithme de classification pour ce deuxième cas est :

$$O(\text{classification}) = \sum_{j=0}^{h} p*(((n+1)*a*f_a)+(p*a*ft)+(pa*p*h*z)+p)$$

O(classification) = O (p * h * ((n*a*fa) + (p*a*ft) + (pa*p*h*z)+p))

où

p = nombre des points de vue du concept

h = hauteur maximale des points de vue du concept

n = nombre maximal de sous-classes directes d'une classe

a = nombre d'attributs d'un concept

fa = coût pour valider le type pour une valeur d'attribut

ft = coût pour valider l'intersection de deux types T1, T2

pa = nombre de passerelles du concept.

z = nombre maximal de sous-arbres de C_min marqués impossible

À partir de ces résultats nous pouvons conclure :

° Si le nombre de passerelles du concept est petit, notamment, s'il est plus petit que le nombre de points de vue multiplié par la hauteur maximale de ces points de vue (pa < p * h), alors l'algorithme utilisant la propagation par passerelles est moins coûteux que l'algorithme sans passerelles. Cela est en fait, le cas le plus probable. Il suffit de constater que si pa >= p * h, alors chaque descente d'un niveau de l'instance entraîne le déclenchement d'une passerelle (dû aux marques du point de vue sur lequel s'est effectuée la descente), ce qui implique qu'il existe dans chaque point de vue une chaîne de classes (liées par la relation de spécialisation) allant de la racine à une feuille, pour laquelle toute classe a une passerelle ; cette structure de points de vue n'est pas très probable.

Par ailleurs, nous pouvons conclure que la complexité de l'algorithme est déterminée principalement par deux termes **(n*a*fa) + (p*a*ft)** et **(pa*p*h*z).**

° Le facteur principal du terme (n*a*fa) + (p*a*ft) est fa (et ft). Donc, le coût principal de cette partie de l'algorithme est proportionnel au coût de la fonction de validation du type d'une valeur, valider (dont le principe est utilisée aussi bien pour fa que pour ft). Enfin, pour les instances simples, la complexité de la fonction valider dépend **du nombre d'éléments du domaine de l'attribut pour la classe, t** (dans le cas des concepts non ordonnés) et du **nombre d'intervalles du domaine normalisé in** (dans le cas des concepts ordonnés). Dans TROPES, la procédure de normalisation du domaine du type d'un attribut d'une classe (§ 7.2.1) vise à réduire au maximum les valeurs **in** et **t**. Pour les instances composées, la complexité de la fonction valider est affectée d'un facteur $\mathbf{a^m}$, où **a** est le nombre d'attributs du concept et **m** le nombre de concepts. Elle dépend donc fortement du nombre d'attributs du concept.

° Le deuxième facteur intervenant dans la complexité, (pa*p*h*z), montre simplement que le nombre d'éléments de la structure taxinomique du concept a une incidence directe sur la complexité de l'algorithme. La valeur de z est normalement petite, c'est le nombre de passerelles déjà activées (en moyenne la moitié = pa/2) qui ont produit une marque impossible (1 sur 2) pour une classe du point de vue en question (en moyenne 1 de p, p étant le nombre des points de vue) et qui sont sous-classes de C_min. Donc, ce deuxième terme de la complexité est déterminé par **le nombre de points de vue p et leur hauteur h.**

Cette complexité de la classification est donc O(h*p*ft). En reprennant les conclusions de la première partie du chapitre, nous avons que la fonction de classification a un coût proportionnel à ft, que la taille d'un état est p et que la trace maximal d'une classification est h. Donc, la complexité de notre algorithme de classification est la plus petite complexité possible pour la classification d'une instance dans un système d'arbres de classes avec des passerelles.

Conclusion

Dans ce chapitre nous avons montré que l'algorithme de classification multi-points de vue dans TROPES est correcte et efficace.

Dans la première partie du chapitre nous avons prouvé la correction et la convergence de l'algorithme de classification. En effet, la classification multi-points de vue de TROPES classe correctement l'instance ; elle descend correctement l'instance vers les classes les plus spécialisées pour lesquelles elle satisfait les contraintes. Cette descente donne un résultat unique (à partir d'une même configuration de points de vue), indépendamment de l'ordre dans lequel sont explorés les points de vue, de l'ordre dans lequel sont données les valeurs des attributs manquantes, ou encore de l'ordre d'activation des passerelles.

Dans la deuxième partie, nous avons fait le calcul de la complexité maximale de l'algorithme. La complexité de la classification est déterminée par celle de la procédure d'appariement et celle de la procédure de propagation de marques. Le coût de l'appariement dépend du coût de la validation du type d'un attribut d'une classe. Pour réduire la complexité de cette validation, le système calcule le type normalisé de chaque attribut de chaque classe lors du chargement d'une base. Le parcours du graphe dépend du nombre de points de vue dans le concept, du nombre de passerelles et de la hauteur maximale de ces points de vue. L'utilisation des passerelles lors de la descente de l'instance réduit la complexité de l'algorithme.

Bibliographie

[AHO&74] AHO A., HOPCROFT J., ULLMAN J., *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[CAR&92] CARDOSO R., MARIÑO O., QUINTERO A., *Corrección y completud de la classification multi-puntos de vista de TROPES*, Rapport Interne, Département d'Informatique, Université des Andes, Bogotá, 1992.

[COM89] COMON H., *Notes du cours "Systèmes de réécriture",* ENSIMAG, Grenoble, 1985.

[MAR89] MARIÑO O., *Classification d'objets dans un modèle multi-points de vue*, Rapport de DEA d'informatique, INPG, Grenoble, 1989.

[MAR&90] MARIÑO O., RECHENMANN F., UVIETTA P. *Multiple perspectives and classification mechanism in object-oriented representation* , 9th ECAI, pp.425-430, Stockholm 1990.

[QUI93] QUINTERO A., *Parallélisation de la classification d'objets dans un modèle de connaissances multi-points de vue,* Thèse d'informatique, Université Joseph Fourier, Grenoble, juin 1993.

Chapitre 8

Extensions de la classification

Chapitre 8

Extensions de la classification

Introduction

Dans le chapitre précédent nous avons décrit l'algorithme de classification d'instances du modèle multi-points de vue TROPES. Dans ce chapitre, nous présentons des extensions de cet algorithme afin de prendre en compte l'aspect évolutif d'une base de connaissances, l'incomplétude de la connaissance et enfin la complexité des objets du monde représenté. À partir de ces trois aspects nous allons décrire ici trois extensions de l'algorithme de classification : la relocalisation d'instances, la classification hypothétique d'instances et enfin, la classification d'objets composites.

La relocalisation d'instances concerne la reclassification d'une instance déjà classée qui voit ses données changer. Ces changements peuvent être dûs à la suppression de la valeur d'un attribut, à l'ajout de la valeur d'un attribut non valué ou à la modification de la valeur d'un attribut. Dans chaque point de vue d'un concept, l'algorithme de relocalisation doit alors faire remonter dans la hiérarchie de classes l'instance modifiée pour retrouver une classe sûre, puis la redescendre vers des classes sûres plus spécialisées. La remontée de l'instance entraîne des changements dans le marquage des classes qui peuvent se répercuter sur l'activation et/ou la désactivation des passerelles.

La classification hypothétique consiste à faire des hypothèses sur la connaissance manquante pour pouvoir continuer la descente de l'instance au-delà des classes sûres. En effet, l'algorithme de classification simple (§ 6) s'arrête lorsqu'à partir de l'information dont il dispose, le système ne peut plus continuer la descente de l'instance. Lorsque la connaissance de l'instance n'est pas complète, des sous-classes de la plus petite classe sûre de l'instance dans un point de vue restent non marquées ou marquées comme possibles, c'est-à-dire le système ne peut ni affirmer ni nier l'appartenance de l'instance à une de ces classe. Pour pouvoir continuer la descente vers l'une des classes possibles, la classification hypothétique permet à l'utilisateur de poser des hypothèses, soit sur les valeurs manquantes de l'instance, soit sur ses classes d'appartenance, et de continuer la classification avec l'instance hypothétique choisie ; cette extension de la classification a un statut hypothétique et peut être remise en cause à tout moment.

Enfin, la classification d'objets composites concerne la localisation des objets appartenant à un concept composite, c'est-à-dire ayant des attributs de nature composant qui prennent des valeurs dans des concepts définis de la base de connaissances. Cette classification peut être maximale ou minimale. La classification maximale consiste à classer toutes les instances intervenant dans la description de l'objet composite à classer. Dans la classification minimale, seul l'objet composite est classé ; ses composants sont descendus dans leurs concepts seulement lorsque cette descente est nécessaire pour pouvoir continuer la classification de l'objet composite. En TROPES, par défaut, la classification d'objets composites est une classification minimale. Cette classification est comme la classification simple sauf pour les étapes d'appariement et d'obtention d'information qui font appel à des procédures de validation et d'instanciation des valeurs des attributs composants.

8.1. Relocalisation d'une instance

Relocaliser une instance déjà classée dans la base consiste à recalculer ses classes d'appartenance les plus spécialisées. La relocalisation est nécessaire lorsque la connaissance de l'instance évolue. Cette évolution peut être le fruit d'un enrichissement (on ajoute des connaissances à l'instance), d'un appauvrissement (on enlève des connaissances), et/ou d'une modification (on change des connaissances). Ce genre de technique est utilisé aussi par certaines logiques terminologiques pour mettre à jour la position d'un individu [NEB90].

8.1.1. Ajout ou modification de la valeur d'un attribut

Une instance déjà classée de la base comporte deux types de connaissances : d'une part la liste de ses plus petites classes sûres (une dans chaque point de vue) et d'autre part la liste des couples attribut-valeur.

La liste des plus petites classes sûres de l'instance établit des contraintes de type pour les valeurs des attributs de l'instance. Lorsque l'on modifie la valeur d'un attribut ou lorsque l'on ajoute la valeur d'un attribut qui était inconnu, deux cas sont possibles :

- soit la nouvelle valeur pour l'attribut satisfait toutes les contraintes pour cet attribut, décrites dans les classes courantes d'appartenance de l'instance (elle satisfait chacun des types de cet attribut pour chacune de ces classes ; en d'autres termes, la valeur appartient à l'intersection des domaines décrits pas ces types).
- soit elle ne satisfait pas le type de l'attribut dans au moins une des classes courantes de l'instance. Supposons par exemple, que dans une base de connaissances ayant le concept *Personne* la classe *Enfant* contraînt l'attribut *age* a avoir une valeur de moins de 13. Si pour l'instance *Isabelle*, attachée à la classe *Enfant*, l'age passe de 12 à 13 (elle a son treizième anniversaire), alors l'instance ne satisfait plus les contraintes de sa classe et elle doit être relocalisée dans la classe *Adolescent* - ce qui se fait en la remontant à la sur-classe d'enfant, la classe personne puis en la descendant vers adolescent.

Dans le premier cas, l'instance satisfait les types de ses classes d'appartenance. Elle est donc dans un état stable et n'est pas obligée de remonter dans aucun des points de vue. Par contre, il se peut que la nouvelle valeur d'attribut permette de trouver une classe plus spécialisée pour l'instance dans l'un des points de vue. Ainsi, pour ce premier cas, le système relance la classification simple (§ 6) avec la nouvelle valeur de l'attribut et à partir des plus petites classes sûres de l'instance.

Dans le deuxième cas, il y a une inconsistence, car l'instance ne satisfait plus les contraintes de ses classes d'appartenance les plus spécialisées. Pour faire face à cette contradiction, le système fait appel à une procédure de relocalisation dont l'objectif est de retrouver les plus petites classes sûres de l'instance dans les différents points de vue. Ainsi, on peut définir la relocalisation comme la procédure qui, à partir d'un état instable (une instance d'un concept qui ne satisfait plus les types de ses classes sûres), rétablit la stabilité de la base en mettant à jour la liste des plus petites classes sûres de l'instance.

Par la suite, nous allons expliquer la relocalisation d'une instance dans un point de vue instable, puis les répercussions de cette relocalisation dans les autres points de vue, par l'activation ou la désactivation des passerelles.

Relocalisation dans un point de vue

Dans les points de vue où des plus petites classes sûres ont été remises en cause par la modification de l'instance, le système doit faire une relocalisation, en remontant l'instance jusqu'à la nouvelle plus petite classe sûre, puis en la descendant le plus bas possible à partir de cette classe (Fig. 8.1).

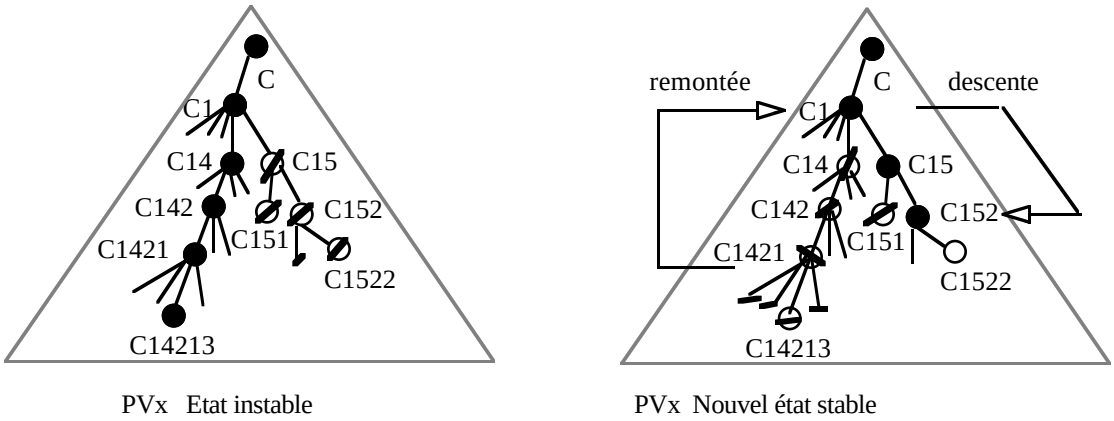


Fig. 8.1. Relocalisation d'une instance I à l'intérieur d'un point de vue. I ne satisfait plus les contraintes de sa plus petite classe sûre C14213, alors elle est remontée le long de ses classes d'appartenance jusqu'à la première classe pour laquelle elle satisfait les contraintes (C1) ; puis elle est redescendue, par classification vers sa nouvelle plus petite classe sûre C1522.

Par la suite nous allons décrire la remontée de l'instance dans un point de vue sans prendre en compte les passerelles ; dans la partie suivante nous reprenons cette remontée pour y ajouter les propagations, activations et désactivations des passerelles trouvées sur le chemin de la remontée.

Soit *a* l'attribut modifié de l'instance *I*, *v0* son ancienne valeur dans *I* et *v1* sa nouvelle valeur; enfin, soit *C_min0* la plus petite classe sûre de l'instance avant la modification de *a*, dans le point de vue en question. Alors la remontée de *I* est nécessaire lorsque *v1* ne satisfait pas le type de *a* dans *C_min0*. La remontée de *I* se fait, en suivant le chemin de ses sur-classes sûres à partir de *C_min0* jusqu'à la première classe *Cs* du chemin pour laquelle *v1* satisfait son type pour *a*. Toutes les classes entre *C_min0* et *Cs* (sans inclure cette dernière) sont marquées comme des classes impossibles pour *I*.

Etant donné que l'attribut qui a déclenché la remontée est l'attribut *a*, seul le type de cet attribut peut faire changer la marque sûre d'une classe pour la marque impossible ; ainsi, lors de la remontée, seul le type de *a* est concerné. Comme nous avons expliqué auparavant (§ 5.6.4), les types des attributs des classes sont structurés dans un graphe de types : chaque nœud du graphe a un type normalisé et une référence vers la première classe (en partant de la racine), qui a ce type. De plus, chaque classe a pour chacun de ses attributs une référence à son type normalisé (Fig. 8.2). L'algorithme de remontée parcourt ce graphe de types en partant du type de *C_min0* et en remontant le long des liens de sur-types jusqu'à trouver un type pour lequel *v1* soit une valeur valide. A ce type est associé la classe la plus spécialisée, D, qui a ce type pour l'attribut a. Comme D est la classe la plus spécialisée ayant ce type pour a, alors ses sous-classes ne l'ont pas. En effet, D est la sur-classe directe de la classe la moins spécialisée du dernier type révisé.

Supposons par exemple (Fig. 8.2) que la valeur de l'attribut entier *a* pour l'instance *I* change de 7 à 43 ; cette nouvelle valeur ne satisfait plus le type $T_0=[5 : 10]$ de *a* dans $C\_min0\ (C_{14213})$. L'algorithme commence par examiner le sur-type $T_1=[0 : 30]$ de T_0. Si pour T_1, *v1* n'est pas valide non plus, l'algorithme poursuit la remontée par le graphe de

types jusqu'à obtenir un type compatible avec *v1* (dans l'exemple le type des entiers positifs T_2*=[0 : INF[*). Une fois le type trouvé, l'algorithme obtient la classe la plus générale, sur le chemin de la remontée de la hiérarchie, dont l'attribut *a* n'a pas ce type (c'est la première classe à affiner le type *T2*, et donc à avoir le type T_1). Dans l'exemple, cette classe est *C14*, sa sur-classe *C1* est alors la plus petite classe sûre ayant un type valide pour *v1*. *C1* est donc la nouvelle plus petite classe sûre de *I, C_min.* Les classes *C14, C1421* et *C14213* sont marquées comme étant des classes impossibles pour *I* (Fig. 8.1)

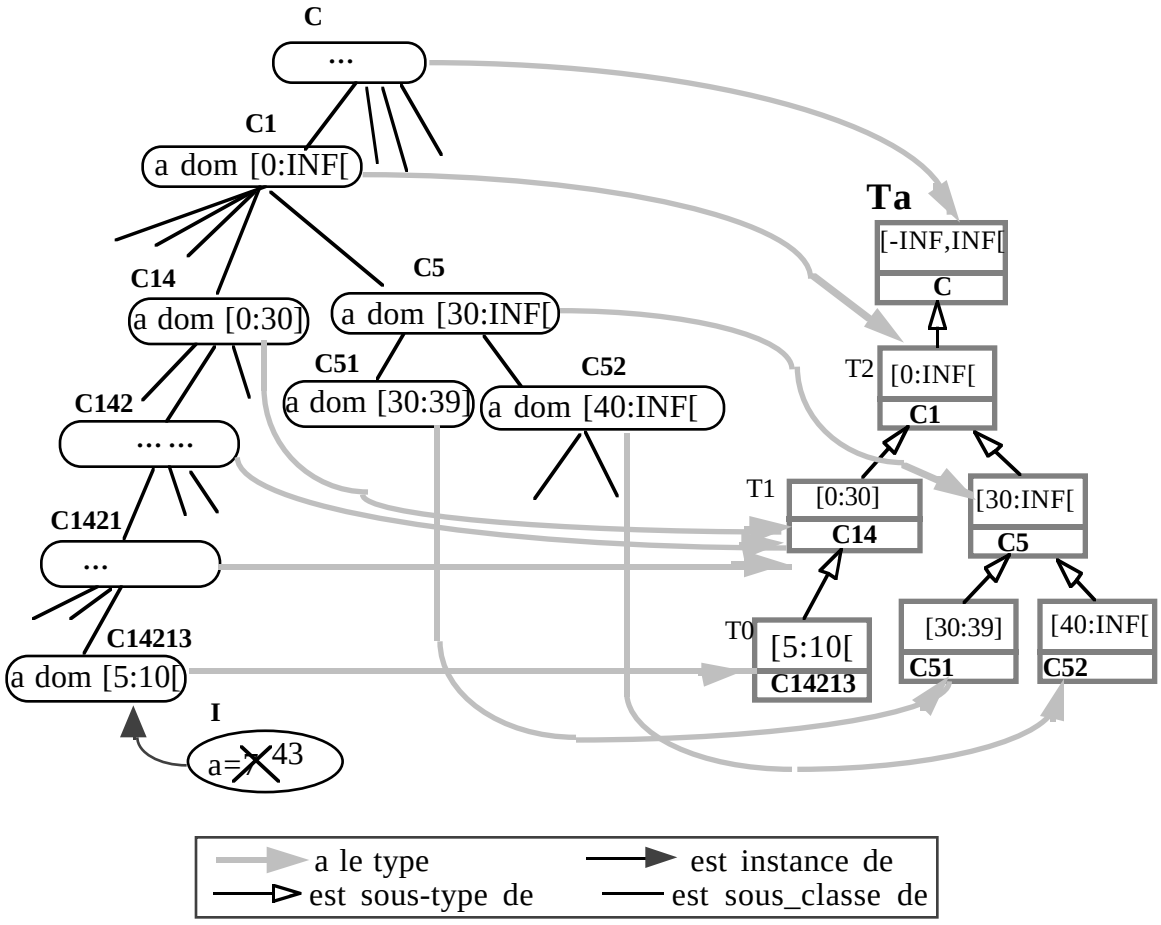
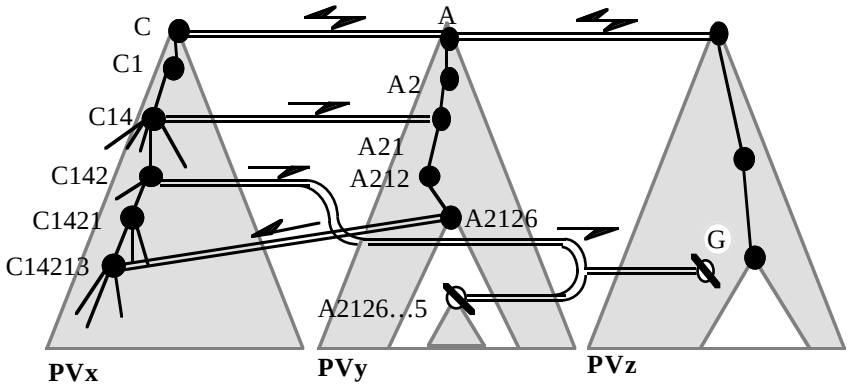


Fig. 8.2. Remontée de I lorsque sa valeur pour a passe de 7 à 43. L'algorithme parcourt le graphe de types de a à partir du type de C14123. 43 satisfait le type [0 : INF[, alors la plus petite classe ayant ce type, C1 (la sur-classe directe de C14) devient la nouvelle C_min.

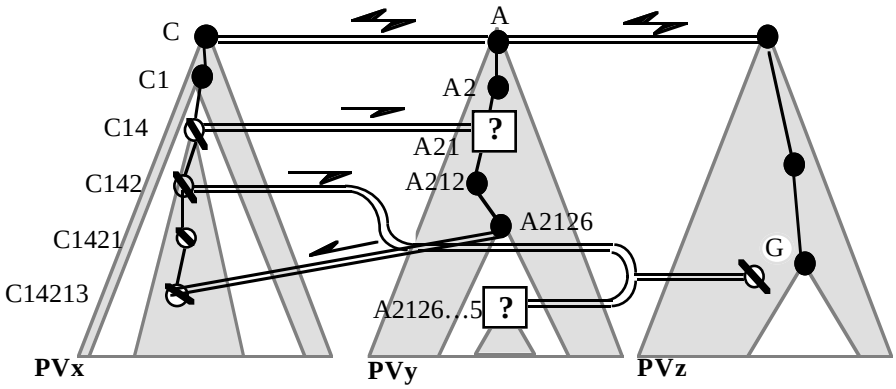
La remontée de l'instance est faite de façon indépendante dans tous les points de vue pour lesquels le type de la plus petite classe sûre n'est plus valide pour I. Le résultat de la remontée est une nouvelle plus petite classe sûre pour I, C_min, qui est forcément une sur-classe de l'ancienne plus petite classe sûre de I, C_min0. La marque de toutes les classes entre C_min0 et C_min (sans inclure cette dernière) a changé de sûre à impossible. La deuxième partie de l'algorithme de relocalisation de l'instance concerne la propagation de ces changements de marques vers les autres points de vue, par des passerelles, en défaisant des passerelles qui avaient utilisé les anciennes marques sûres de ces classes pour déduire d'autres marques.

Propagation de la remontée par des passerelles

La classification multi-points de vue dans TROPES tire parti des passerelles pour faire descendre l'instance dans un point de vue à partir des marques des classes d'autres points de vue (§ 6.3.5). Lorsqu'une instance est remontée dans un point de vue, certaines de ses anciennes classes sûres deviennent impossibles. Si, lors de la précédente classification de l'instance, les marques sûres de ces classes avaient servi pour déduire les marques d'autres classes par des passerelles, alors ces marques déduites ne sont plus valides et doivent être modifiées. (Fig. 8.3).



Avant de remonter l'instance dans PVx



Après avoir remonté l'instance et avoir marqué les nouvelles classes impossibles

Fig. 8.3. La remontée de l'instance dans PVx remet en cause la passerelle de C14 à A2 et celle de (C142, A2126…5) à G car la marque sûre de C14 aurait pu servir a marquer A2 comme sûre et celle de C142 (avec G impossible) à marquer A2126…5 comme impossible .

Comme l'algorithme de classification ne mémorise pas le chemin suivi par la classification ni les passerelles utilisées, le système ne peut pas savoir, une fois la classification achevée, si une passerelle particulière a été activée, c'est-à-dire si l'une de ses classes a été marquée en fonction des autres classes de la passerelle, et non pas par appariement direct avec l'instance.

Ainsi, lors de la remontée de l'instance, le système doit identifier les passerelles ayant pu être activées dans la précédente classification de l'instance et pour lesquelles cette activation ne serait plus valable. Pour chacune de ces passerelles, le système doit vérifier, par appariement direct avec l'instance, si la marque qui aurait pu être déduite par la passerelle est une marque valide ou non au vu de la nouvelle liste d'attributs-valeurs de l'instance. Si elle n'est pas une marque valide, alors le système doit entamer les actions nécessaires pour rétablir la cohérence dans le point de vue de la classe en question.

La procédure de propagation par des passerelles, après la remontée de l'instance modifiée, parcourt toutes les passerelles du concept en faisant une comparaison de leur état e_0 avant la remontée (stockée dans une structure temporaire) et leur nouvel état e_1 après la remontée. Huit sortes de changement de marques sont possibles (Fig. 8.4). À chacun de ces changements de marques correspond une action visant à retrouver la cohérence globale de la base. Cette action peut inclure, outre la procédure d'effacement de la passerelle (enlever la passerelle de la liste de passerelles actives), l'ajout d'une des classes de la passerelle à la liste *sûre-à-possible* de classes dont la marque a changé de sûre à possible. Après avoir parcouru toutes les passerelles du concept, le système appelle une procédure de *relocalisation_possible* pour mettre à jour les marques dans les points de vue des classes de cette liste (cette procédure est décrite plus loin). Nous allons détailler chacun des huit cas possibles de modification de marques d'une passerelle.

a) Dans l'état initial e_0 toutes les sources sont sûres et la destination est sûre. La classe qui change sa marque est la classe destination : dans l'état e_1 la destination est marquée impossible (Fig. 8.4.a). (exemple : Fig. 8.3 passerelle de *A2126* à *C14213*). Si la passerelle avait été utilisée pour déduire une marque, la seule marque déductible serait le marque sûre de la destination ; comme c'est précisément cette marque qui a été changée par la remontée, la seule action à effectuer est l'effacement de la passerelle car elle n'est plus active (§ 6.3.5).

b) Dans l'état initial e_0 la destination est sûre et parmi les sources il y a des classes sûres et des classes impossibles. Dans l'état e_1 la destination est devenue impossible (Fig.8.4.b). De même que pour le cas précédent, la passerelle ne change pas d'autre marque et elle devient inactive, il faut alors l'effacer.

c) Dans l'état initial e_0 toutes les sources ainsi que la destination sont sûres. Dans l'état e_1 une des sources devient impossible (Fig. 8.4.c) ; (exemple : Fig. 8.3 passerelle de *C14* à *A21*). Si la classe destination a été marquée comme sûre par propagation des marques des sources, alors, étant donné que pour l'une des sources la marque a changé de sûre à impossible, la propagation doit être révisée. Cependant, si la marque sûre de la destination n'a pas été obtenue par propagation de cette passerelle mais par appariement directe dans son point de vue, alors la classe conserve la marque sûre. Ainsi, le système valide l'instance I par rapport au type de la classe destination.

Si I $\in$ destination => effacer la passerelle
Si I $\in$? destination => ajouter la destination à la liste *sûre-à-possibles*[1]

d) Dans l'état e_0 la destination est sûre et les sources sûres ou impossibles, et dans l'état e_1 une des sources sûres devient impossible. La passerelle dans e_0 n'ayant pas été activée, il faut juste l'effacer.

e) Dans e_0 la destination et toutes les sources sauf une, sont sûres, cette dernière étant impossible (Fig. 8.3 passerelle de *[C142, A2126…5]* à *G*). Cette combinaison de marques a pu entraîner le marquage impossible de la source impossible (*A2126…5* dans l'exemple). Dans l'état e_1 une des sources sûres devient impossible (Fig. 8.4.e). Le système doit alors vérifier si la marque de la source qui était impossible avait été déduite par la passerelle (et en fait il n'y a pas de contradictions entre I et cette classe) ou par la descente de l'instance dans son point de vue. Dans le premier cas, la classe est enlevée de la liste de racines impossibles de son point de vue, L_imp, si elle y est. Dans le deuxième cas, le système ne fait aucune modification ; la passerelle reste inactive.

[1]La classe destination ne peut pas être impossible, car cela signifierait que la passerelle utilisée précédemment est inconsistante ou que la modification de la valeur de a pour I entraîne la non satisfaction du type de cette classe (auquel cas sa marque est déjà changée par la remontée de l'instance dans son point de vue).

f) L'état initial e_0 possède des sources sûres ou non marquées et la destination est non marquée. Dans l'état e_1 une des sources sûres devient impossible. La passerelle ne sert plus à déduire des marques et elle doit être effacée (Fig. 8.4.f).

g) L'état e_0 a la destination sûre et des sources non marquées. Alors indépendamment du changement, la passerelle doit être effacée (Fig. 8.4.g).

h) Dans tous les autres cas, la passerelle n'est pas changée et sa révision n'entraîne aucune action (Fig. 8.4 h).

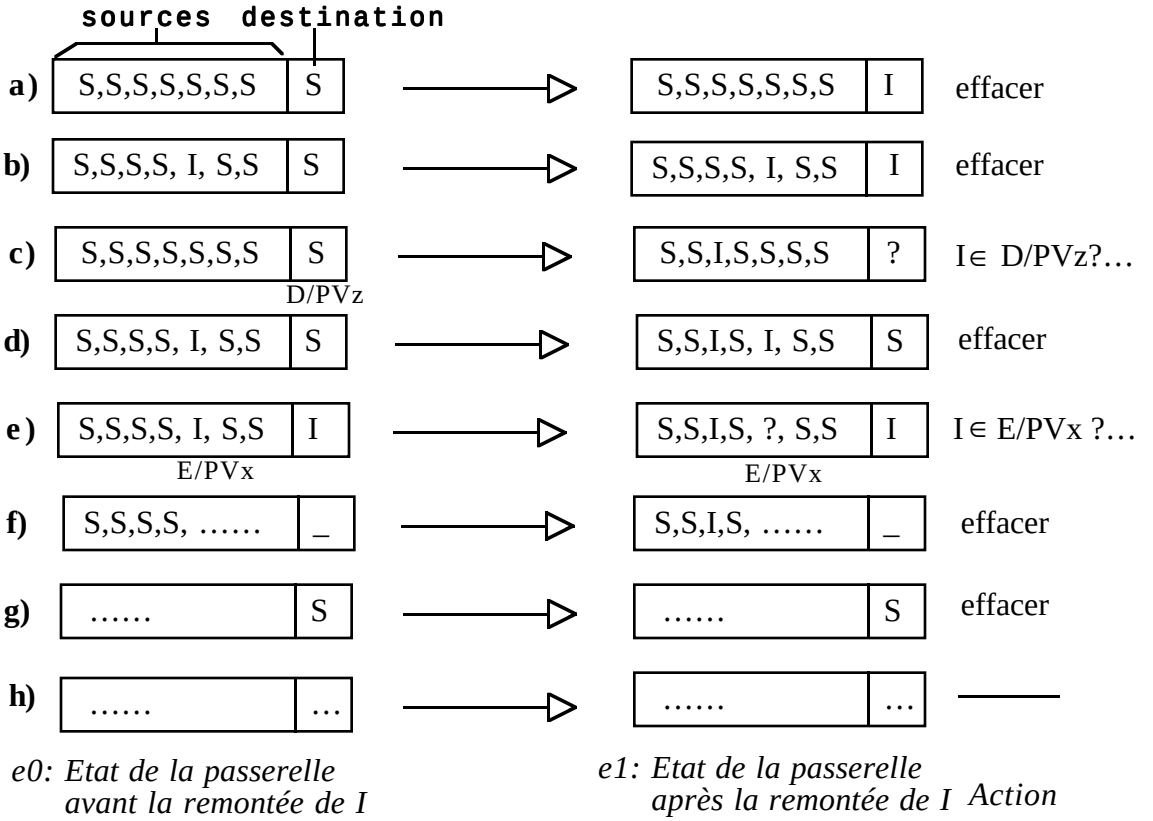


Fig. 8.4. Le changement de la marque sûre par impossible d'une classe faisant partie d'une passerelle peut remettre en cause la validité des inférences provoquées par la passerelle (ex. c), e)) ou bien rendre inactives des passerelles actives (ex. f)).

Relocalisation possible

Lorsque la marque d'une classe change de sûre à possible, l'instance doit aussi remonter le long de son chemin de classes sûres jusqu'à trouver une nouvelle classe sûre. Une classe est possible lorsque l'instance n'a pas de valeur pour au moins un des attributs de la classe et que les attributs valués de l'instance satisfont bien les contraintes de la classe. La procédure *remontée_possible* cherche à trouver la première classe pour laquelle l'instance a tous les attributs valués ; il est important de noter que cette procédure ne fait aucune validation de type, elle regarde seulement si pour une classe donnée l'instance a tous les attributs valués, auquel cas elle devient la plus petite classe sûre pour ce point de vue ; sinon elle continue avec la sur-classe de cette classe.

Après avoir remonté l'instance jusqu'à sa nouvelle plus petite classe sûre, le système propage le changement de la marque sûre à la marque possible, pour toutes les classes situées entre l'ancienne plus petite classe sûre et la nouvelle plus petite classe sûre (sans inclure celle-ci). Cette propagation ressemble à celle décrite précédemment : elle peut effacer une passerelle ou au contraire la rendre active, rendre possible une classe

impossible ou bien rendre possible une classe sûre. Cette dernière action se fait en ajoutant la classe concernée à la liste de *sûre-à-possible*. Ainsi, la procédure de *relocalisation_possible* continue tant qu'il y a des classes dans la liste *sûre-à-possible*. La terminaison de l'algorithme est garantie par le fait que chaque relocalisation fait monter l'instance au moins d'une classe dans l'un des points de vue ; donc au pire elle remonte dans tous les points de vue jusqu'à la racine.

Les six cas considérés pour la propagation des marques lorsqu'une classe sûre d'une passerelle devient possible sont (Fig. 8.5) :

a) Si toutes les classes de la passerelle étaient sûres et si la destination D/PVx devient possible, alors elle est à nouveau marquée sûre (C_min := D/PVx) (Fig. 8.5.a).

b) Si la destination était sûre et qu'il y avait des sources non marquées, alors si la destination devient possible, la passerelle devient active (Fig. 8.5.b).

c) Si toutes les classes de la passerelle étaient sûres et si l'une des sources devient possible, le système compare I à la destination D/PVx ; si D est sûre pour I, la passerelle est effacée; sinon D/PVx est ajoutée à la liste *sûre-à-possible* (Fig. 8.5.c).

d) Si la destination et une source E/PVx étaient impossibles et toutes les autres sources étaient sûres, alors si une source sûre devient possible l'algorithme vérifie que la classe E/PVx est possible pour I, auquel cas cette classe E/PVx est marquée possible (elle est effacée de la liste des racines impossibles de son point de vue L_imp/PVx) (Fig. 8.5.d).

e) Si la destination de la passerelle était sûre, la passerelle est effacée indépendamment de la modification de ses sources (Fig. 8.5.e).

f) Dans les autres cas la passerelle n'est pas modifiée (Fig. 8.5.f).

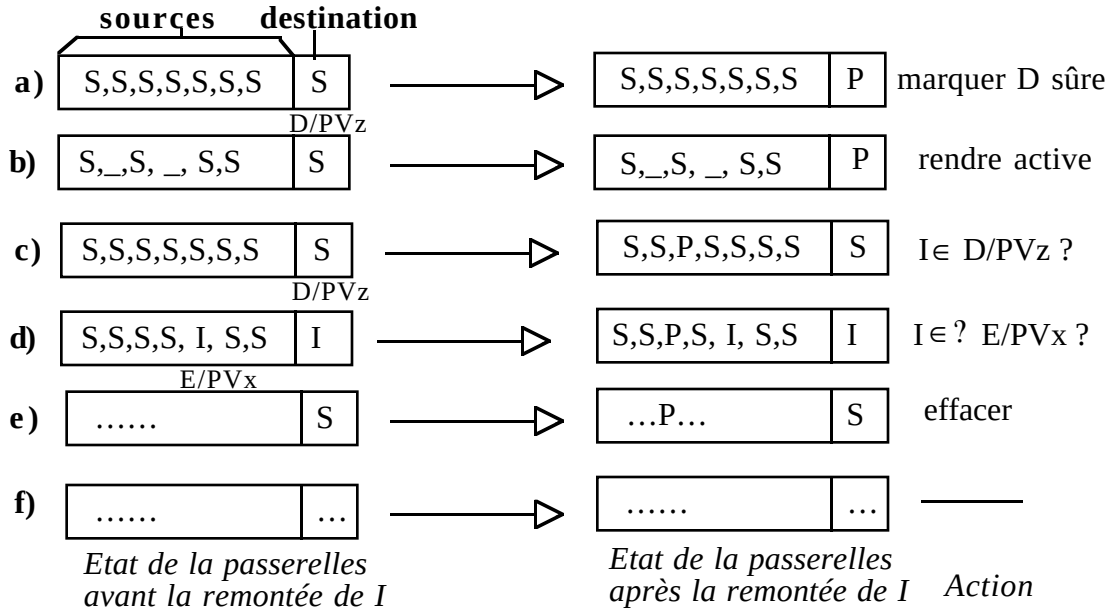


Fig. 8.5. Le changement de la marque sûre par possible d'une classe faisant partie d'une passerelle, peut remettre en cause la validité des inférences tirées de la passerelle (ex. c) d)) ou bien rendre actives des passerelles (ex.b)).

Descente de l'instance

Une fois la remontée et la propagation des marques terminées, la relocalisation essaie de faire descendre l'instance au maximum dans les différents points de vue du concept. Cette descente part d'un état stable et consiste à appeler, à partir de cet état, la procédure de classification (§ 6).

8.1.2. Suppression de la valeur d'un attribut

Lorsque la valeur d'un attribut *a* est supprimée dans l'instance *I* (elle devient inconnue pour I), les plus petites classes sûres de *I* qui établissent des restrictions pour cet attribut *a* doivent changer leur marque de sûre à possible. Ainsi, dans les points de vue de ces classes, l'instance remonte vers la première classe ne décrivant pas *a*. Cette remontée est faite par la procédure de *relocalisation_possible* présentée précédemment. À la différence du cas précédent, une fois que la remontée est finie (le système a trouvé la nouvelle plus petite classe sûre C_min pour I), l'algorithme ne cherche pas à descendre l'instance vers les sous-classes de C_min. En effet, la connaissance de *I* s'étant appauvrie et non pas enrichie, aucune nouvelle descente n'est possible.

La procédure de relocalisation que nous venons de présenter demande le parcours des passerelles du concept et l'éventuelle modification et propagation des marques des classes. On peut se demander si cette procédure ne devient pas plus coûteuse que relancer une classification normale à partir de la racine de tous les points de vue. On peut espérer que la relocalisation soit plus efficace que la classification, lors de la modification d'un attribut de l'instance puisque le coût de la procédure d'appariement de la classification est en général très élevé (§ 7.2.3) et que l'algorithme de relocalisation fait très peu de validation de types.

8.2. Classification hypothétique

La classification hypothétique consiste à classer une instance pour laquelle on a, outre les connaissances sûres, des hypothèses. Le but de la classification hypothétique est de continuer la descente d'une instance dans son concept lorsque celle-ci s'est arrêtée par manque d'information, en faisant des hypothèses. Une hypothèse est une affirmation sur une connaissance qui n'est pas sûre et qui, de ce fait, peut être remise en question à tout moment d'un raisonnement. Les déductions faites à partir d'une connaissance hypothétique ont une valeur de vérité relative : elles sont vraies si les hypothèses utilisées pour les déduire s'avèrent vraies.

La connaissance manipulée par la procédure de classification étant les valeurs des attributs de l'instance à classer et ses classes d'appartenance, c'est sur ces deux aspects que le système va permettre l'émission d'hypothèses.

8.2.1. Hypothèses sur les valeurs d'attributs

Lorsque l'algorithme de classification ne peut plus faire descendre l'instance dans aucun des points de vue du concept, le système offre à l'utilisateur la possibilité de continuer la descente avec des valeurs possibles pour les attributs manquants. Le système propose, comme valeurs possibles d'un attribut, les valeurs par défaut pour cet attribut dans les différentes sous-classes de la classe courante de l'instance du point de vue courant. L'utilisateur peut prendre une de ces valeurs ou bien proposer une autre valeur valide (satisfaisant l'intersection des types de l'attribut pour les classes d'appartenance de l'instance).

L'instance ainsi obtenue a un statut hypothétique. Elle comporte une partie sûre : la liste d'attributs et de valeurs sûres, et une partie hypothétique : la liste d'attributs et de valeurs hypothétiques. La classification essaie de descendre cette instance hypothétique le plus bas possible dans les différents points de vue.

La classification d'une instance hypothétique suit la même démarche que la classification d'une instance normale (§ 6). Elle fait descendre l'instance dans les différents points de vue de son concept. Les marques sûres, possibles et impossibles sont interprétées ici comme "sûre sous l'ensemble d'hypothèses H", "possible sous l'ensemble d'hypothèses H" et "impossible pour l'ensemble d'hypothèses H". La classification d'instances hypothétiques comporte les mêmes étapes que la classification normale. Lorsque, dans les étapes d'obtention d'information et d'appariement, l'utilisateur donne des valeurs pour des attributs manquants, ces valeurs peuvent être interprétées de trois façons différentes :

a) La valeur est sûre pour l'instance, indépendamment des hypothèses précédentes.
b) La valeur est hypothétique et cette hypothèse est indépendante des hypothèses précédentes.
c) La valeur est une valeur hypothétique choisie en fonction des hypothèses précédentes. C'est notamment le cas des valeurs par défaut, car celles-ci sont rattachées aux sous-classes des classes vers lesquelles l'instance est descendue grâce aux valeurs sures ou hypothétiques de ses attributs.

Dans le premier cas, la valeur sûre est ajoutée à la partie sûre de l'instance. Les deux autres cas sont traités par le système avecdeux modes de travail différents : le premier suppose que les hypothèses sont indépendantes les unes des autres tandis que le deuxième suppose qu'une hypothèse dépend de toutes les hypothèses précédentes. La distinction entre ces deux modes est importante lors de la modification d'une hypothèse.

Hypothèses indépendantes

Le premier mode correspond aux hypothèses indépendantes : la valeur hypothétique donnée à un attribut ne dépend pas des valeurs des autres attributs. La modification d'une hypothèse ne met pas en cause les valeurs hypothétiques ou sûres des autres attributs de l'instance, mais seulement les liens d'appartenance de celle-ci. Dans ce cas l'instance hypothétique ainsi modifiée est reclassée en faisant appel à l'algorithme de relocalisation d'instances présenté précédemment (§ 8.1).

Hypothèses dépendantes

Le deuxième mode de raisonnement prend en compte l'ordre d'émission des hypothèses : une hypothèse dépend de toutes les hypothèses précédentes. Ainsi, lorsque l'utilisateur modifie la valeur d'une hypothèse, h_i, toutes les hypothèses faites ensuite sont effacées, et l'instance est remontée au niveau où elle était avant l'émission de h_i.

Ce deuxième mode de raisonnement peut être vu comme un cas extrême du premier où plusieurs valeurs sont modifiées en même temps (h_i est modifiée et les hypothèses postérieures sont effacées). Le reclassement de l'instance peut être fait avec les algorithmes de relocalisation d'une instance lors de la suppression d'une valeur et lors de la modification d'une valeur (§ 8.1).

Par des raisons d'efficacité nous avons pris une approche un peu différente. À chaque itération de la boucle de classification, l'émission d'hypothèses sur les valeurs d'attributs cherche à compléter l'information nécessaire pour descendre l'instance d'un niveau dans un point de vue. Défaire une hypothèse h_i entraîne l'effacement des hypothèses postérieures et donc des liens d'appartenance de l'instance obtenus après l'émission de h_i (par toutes les itérations postérieures de la boucle de classification). Au lieu de recalculer l'état de l'instance avant l'émission de h_i, le système génère, pour chaque itération de la boucle de classification, un contexte hypothétique avec les hypothèses faites pour

descendre l'instance d'un niveau dans un point de vue ; les contextes hypothétiques sont liés entre eux de façon ordonnée.

À la fin de chaque itération de la classification un contexte hypothétique contient la description d'un état stable :

- La liste de classes les plus spécialisées de l'instance hypothétique pour chaque point de vue.
- La liste de racines d'arbres impossibles L_imp pour chaque point de vue.
- La liste ordonnée des hypothèses (tuples attribut, valeur) qui ont permis de descendre l'instance d'un niveau dans un point de vue dans cette itération.

Une hypothèse particulière h_i appartient à un contexte précis C_X. Défaire une hypothèse h_i consiste à récupérer le contexte précédant l'émission de h_i (C_{X-1}), et à partir de celui-ci recommencer la génération du contexte C_X avec les hypothèses de l'ancien C_X précédant h_i. La récupération d'un contexte se fait en effaçant de la liste de contexte ceux qui le suivent.

Supposons, par exemple, qu'un département de police possède une base de connaissances de *Personnes*, permettant aux témoins d'un meurtre de reconnaître l'assassin. Les *personnes* de la base sont décrites selon différents points de vue, dont *odeur* (PV1) et *aspect* (PV2) (Fig. 8.6). Le point de vue *odeur* vise à classer l'assassin en fonction de son parfum et le point de vue *aspect* en termes de son aspect physique. De plus, la police a pu constater que les parfums doux sont utilisés par les femmes blondes aux yeux clairs et les parfums secs par les brunes alors que les hommes mettent de l'eau de Cologne.

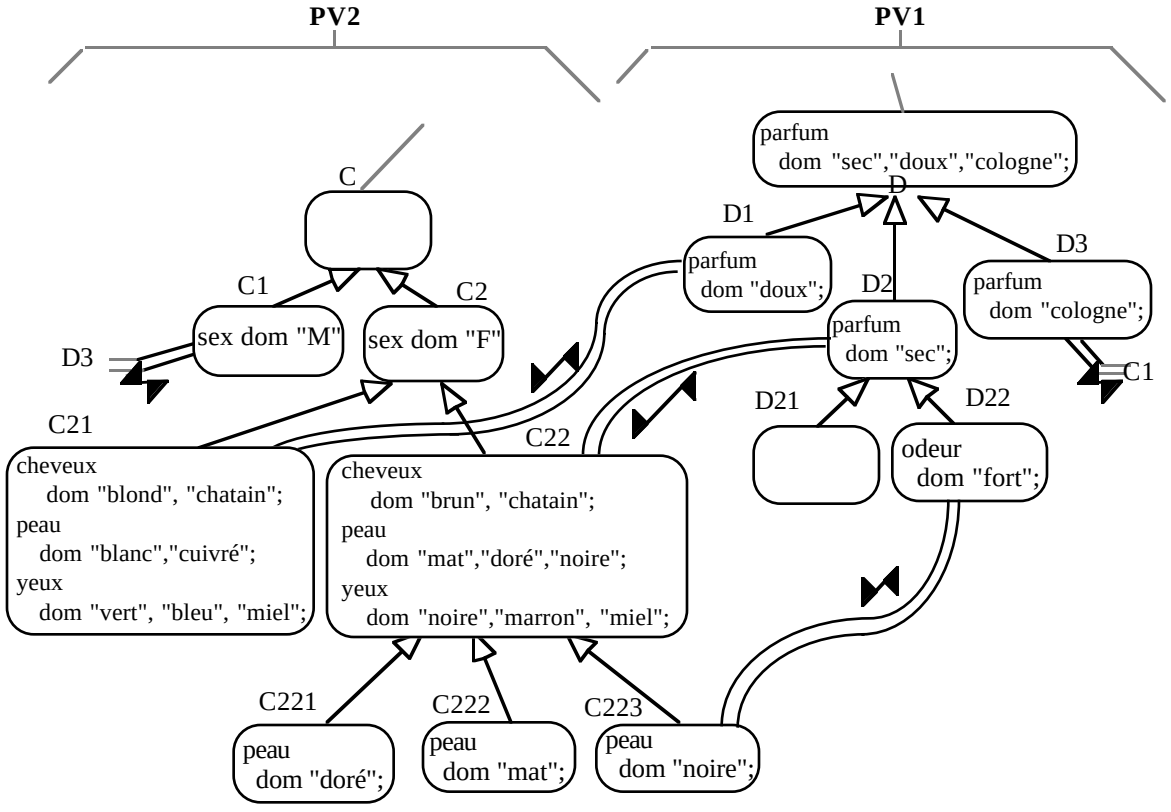


Fig. 8.6. Concept *Personne* vu des points de vue *Odeur* (PV1) et *Aspect* (PV2).

Si un témoin se rappelle d'avoir senti l'odeur d'un parfum lors de la fuite de l'assassin, alors l'instance est tout d'abbord attachée à la classe D/PV1 (et à la racine P

du point de vue PV2). Si, de plus, il croit que l'assassin avait un parfum sec, alors il peut faire l'hypothèse de la valeur "sec" pour l'attribut *parfum*. Cette hypothèse fait descendre l'instance vers la classe D2/PV1 et par une passerelle à la classe C22/PV2. Le premier contexte hypothétique est ainsi créé (Fig. 8.7a)).

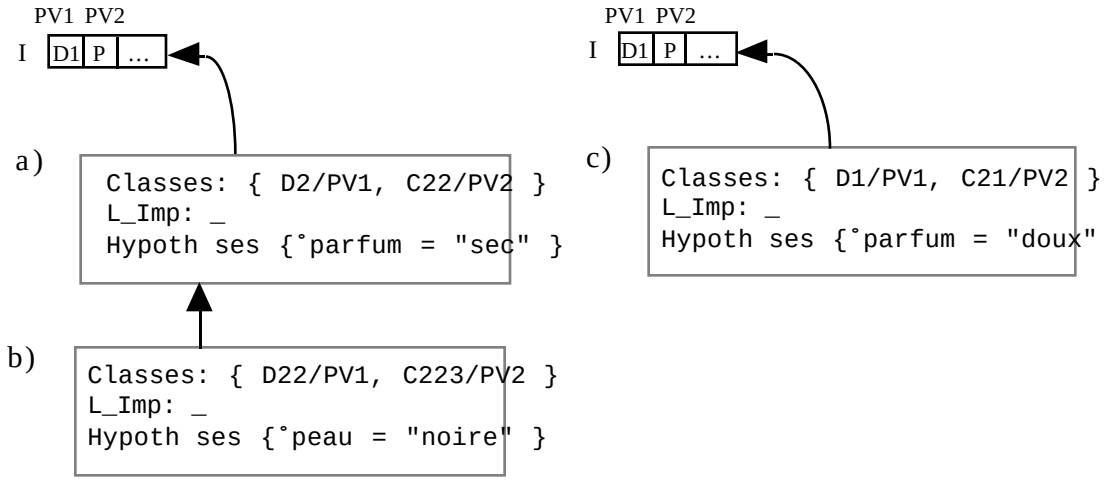


Fig. 8.7. I commence attachée à la classe D/PV1 et à la racine de PV2. L'hypothèse *parfum = "sec"* fait descendre l'instance vers D2/PV1 et C22/PV2. L'algorithme crée le contexte hypothétique a). L'hypothèse *peau="noire"* entraîne la création du contexte b) . Enfin, la modification de l'hypothèse sur la valeur de *parfum* entraîne l'effacement de a) et de b) et la création d'un nouveau contexte c).

Si de plus il fait l'hypothèse que la couleur de la peau de l'assassin est noire, alors l'instance hypothétique descend vers les classes C223/PV2 et D22/PV1 et le deuxième contexte hypothétique est créé (Fig. 8.7.b)). Enfin, si en réfléchissant mieux, le témoin décide de changer la première hypothèse d'un parfum sec à un parfum doux, alors le système efface les deux contextes hypothétiques et en crée un nouveau, où l'instance est attachée à D1/PV1 et à C21/PV2 (Fig. 8.7.c))

La classification hypothétique que nous venons de présenter a deux grandes restrictions :

- Elle laisse à la charge de l'utilisateur le choix et la mise en cause des hypothèses.
- Une hypothèse est soit indépendante de toutes les hypothèses précédentes, soit dépendante de toutes les précédentes.

Ces deux limitations sont été supprimées dans un prototype de TROPES qui inclut des tâches [ORS90] et des liens de dépendance entre les attributs d'une instance [GEN90].

- L'ajout des tâches au modèle permet le calcul automatique des valeurs des attributs. Une tâche peut être vue comme un ensemble de méthodes attachées à un attribut du concept. Ces méthodes calculent la valeur de l'attribut à partir des valeurs d'autres attributs du concept. Si une tâche retourne plusieurs valeurs, alors le système crée pour chaque valeur une instance hypothétique et continue la classification avec la première ; si celle-ci s'avère inconsistante, le système continue la classification avec la deuxième instance hypothétique générée (les valeurs obtenues pour d'autres attributs par le déclenchement d'autres tâches ne satisfont pas ses classes d'appartenance) [GEN90].
- Par ailleurs, la gestion de dépendances inter-attributs garde une trace de la liste des attributs $\{a_{i1}, a_{i2}, \ldots, a_{in}\}$ utilisés par une tâche pour le calcul d'un nouvel attribut a_i (liste des justifications de a_i). Lorsque la valeur d'un des attributs a_{ix} de la liste est

modifiée, la valeur de a_i est mise en question [GEN90]. Dans ce prototype, une hypothèse dépend des valeurs des attributs utilisés pour la calculer, et elle est mise en cause quand une de ces valeurs change.

8.2.2. Hypothèses sur les classes d'appartenance

Une deuxième façon de continuer la descente d'une instance lorsque la classification certaine s'arrête par manque d'information, est de faire des hypothèses sur les classes d'appartenance. Une hypothèse dans cette aproche consiste à choisir, à chaque niveau de la classification, parmi les sous-classes directes des plus petites classes sûres de l'instance, une classe possible d'appartenance de l'instance pour lui donner la marque "sûre hypothétique" et continuer la descente par cette classe.

La marque "sûre hypothétique" a dans la classification hypothétique le même effet que la marque "sûre" de la classification normale, à savoir : la propagation des marques à l'intérieur d'un point de vue et entre les différents points de vue pour retrouver un état stable et l'intersection du type de la nouvelle classe avec le type courant de l'instance.

La seule différence dans la démarche de la classification avec des hypothèses d'appartenance et la classification normale est que la première garde l'information de la classification sûre obtenue avant l'émission d'hypothèses, puis crée une copie de l'instance avec un statut hypothétique et classe cette instance hypothétique avec le même algorithme de classification normale.

Une hypothèse sur une classe d'appartenance d'une instance hypothétique est, par sa nature, dépendante des hypothèses précédentes. La gestion des modifications des hypothèses d'appartenance se fait par relocalisation de l'instance, en la remontant dans les différents points de vue. Cette relocalisation utilise l'algorithme de *relocalisation_possible* qui prend en charge la propagation des marques lorsqu'une marque sûre devient possible (§ 8.1.1).

8.3. Classification d'objets composites

Un objet composite en TROPES est une instance ayant des attributs de nature composante. La relation entre les composants (parties) et l'objet est une relation de dépendance exclusive et existentielle (§ 2.5.1). La décomposition d'un objet peut être multiple (dans différents points de vue) et imbriquée (les composants peuvent être des objets composites) ; elle peut alors être vue comme une forêt d'arbres de décomposition (§ 5.8).

Le but de la classification d'un objet composite est, comme pour toute autre instance de la base, de trouver ses classes d'appartenance les plus spécialisées dans les différents points de vue du concept et de compléter l'information associée à l'objet .

Deux interprétations sont possibles pour la classification d'objets composites [MAR91] : la classification minimale et la classification maximale. Les deux essaient de descendre l'objet composite le plus bas possible dans les hiérarchies de son concept, la différence réside dans le traitement des parties. Dans la **classification minimale**, on vérifie seulement que les parties satisfont les contraintes imposées par la classe de l'objet composite ; on essaie de descendre un composant dans les points de vue de son concept seulement si cela est nécessaire à la validation. Cette classification est la classification utilisée par défaut dans TROPES ; elle correspond à la classification normale

pour des instances composées. La **classification maximale**, par contre, classe tous les composants le plus bas possible dans leurs hiérarchies (Fig. 8.8).

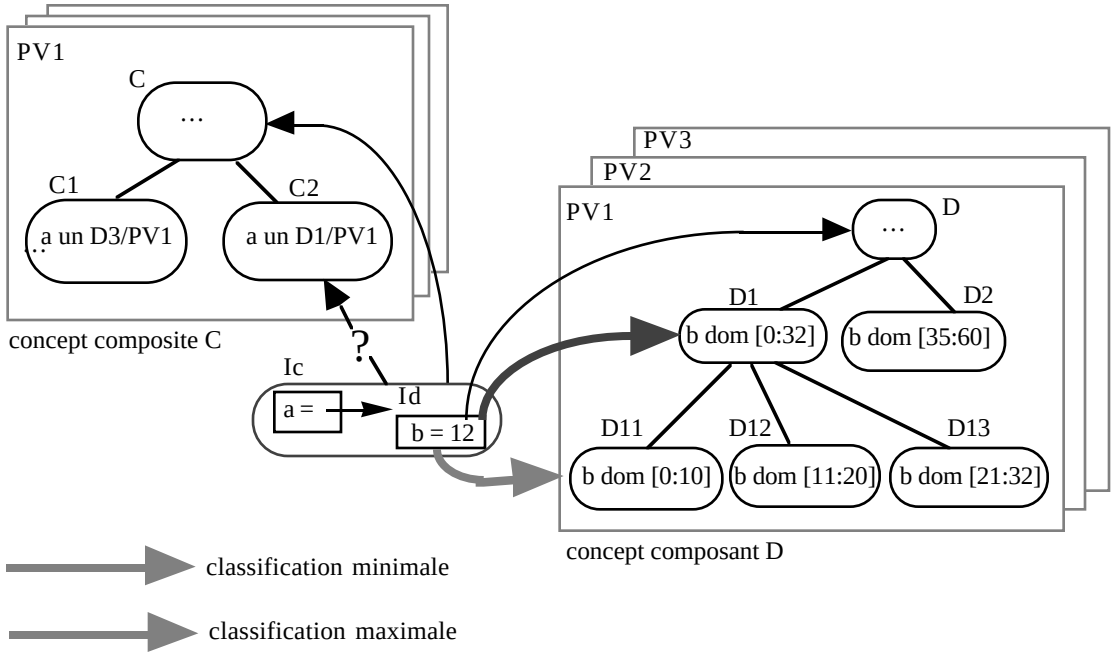


Fig. 8.8. Lors de la classification minimale de *Ic*, l'algorithme value si son composant *Id* est une instance de la classe D1/PV1 du concept composant D. La classification maximale ne s'arrête pas là, mais elle cherche à classer l'instance *Id* le plus bas possible ; elle descend *Id* jusqu'à la classe D11 .

La classification maximale peut être très coûteuse. En effet, un objet composite peut être composé de plusieurs instances qui peuvent être aussi des instances composites. Si la description d'un objet composite comporte en tout *x* instances de différents concepts, alors la classification de cet objet déclenche *x* classifications. Le nombre *x* de classifications étant au plus le nombre total X d'instances de la base, la complexité de la classification maximale d'objets composite est

O (classif_max) = X *O(classification).

Par la suite nous allons considérer uniquement la classification minimale d'objets composites. En termes généraux, cette classification est comme la classification simple. Elle est aussi décrite par une boucle qui compte cinq étapes : obtention d'information, appariement, propagation de marques, mise à jour de l'information et choix du prochain point de vue. Les procédures de propagation de marques, de mise à jour d'information et de choix du prochain point de vue sont les mêmes que pour la classification simple. L'appariement et l'obtention d'information changent pour prendre en compte la validation du type d'un attribut composant et l'instanciation de composants d'un objet composite.

8.3.1. Obtention d'information

La procédure d'obtention d'information vise à compléter l'information associée à une instance pour laquelle le système vient d'obtenir une nouvelle plus petite classe sûre, C, soit par l'activation d'une passerelle soit par une procédure d'instanciation rattachant explicitement l'instance à la classe. Dans ces deux cas, le système sait que l'instance appartient à C mais il ne connaît pas toutes les valeurs des attributs de C. Il doit donc essayer de compléter cette information. Compléter l'information des propriétés consiste à demander leurs valeurs à l'utilisateur et vérifier qu'ils satisfont le type de cet attribut pour la classe (§ 6.3.3). Compléter l'information d'un attribut composant, par contre, consiste à

créer automatiquement des instances des classes spécifiées par le type de cet attribut dans C, puis obtenir les valeurs des attributs de ces instances composantes de la même façon : pour les propriétés en posant des questions à l'utilisateur ; pour les composants en les instanciant. Comme nous l'avons présenté auparavant, l'existence des composants dépend de celle de l'objet composite ; donc la création d'un composant suit celle de l'objet composite. Si, dans la classe dappartenance de l'objet composite, l'attribut composant est mono-valué ou s'il est multi-valué de cardinalité connue, alors le système crée automatiquement la ou les instances qui les valident, par instanciation de la (les) classe(s) indiquée(s) dans C. Sinon, il repousse leur création (car il ne sait pas combien d'objets créer pour ce composant multi-valué).

8.3.2. Appariement

Dans TROPES, l'affinement d'un attribut composant multi-valué peut se faire par spécialisation en affinant son type ou sa cardinalité (comme pour les attributs définis[1]) ou par éclatement de l'attribut en plusieurs attributs (§ 5.8.2). Dans ce deuxième cas, chacun de ces nouveaux attributs ajoute des contraintes à l'attribut initial. Pour faire l'appariement dans ce cas, le système interprète un composant multi-valué comme une liste ordonnée de valeurs (§ 5.2.1). Lors de l'éclatement il fait correspondre les premiers éléments de la liste au premier attribut produit par l'éclatement, la deuxième partie de la liste au deuxième attribut et ainsi de suite, puis il vérifie si ces parties satisfont les contraintes correspondantes (§ 5.8.2).

8.3.3. Exemple de classification d'un objet composite

Pour finir cette partie, nous allons présenter un exemple de classification multi-points de vue d'un objet composite (instance du concept "automobile" de l'exemple présenté dans § 5.8, Fig. 5.26). Supposons qu'on veuille classer (avec une classification minimale) l'automobile d'immatriculation "8018 YV 38" pour laquelle on sait seulement qu'elle supporte une charge utile de 4 tonnes (Fig. 8.9). En sachant qu'il s'agit d'une instance d'automobile, le système peut l'attacher à toutes les classes racines du concept ; il crée automatiquement des instances de chacun des composants du point de vue *mécanique* car ils sont tous des composants mono-valués (le *moteur*, la *transmission*, la *suspension*, la *direction* et le *freinage*). Du point de vue *physique*, une automobile a deux composantes : *carrosserie* et *roues*. Ce dernier est un composant multi-valué dont la cardinalité n'est pas connue pour la classe racine *Automobile*. Le système diffère donc l'instanciation du composant *roues* du point de vue *physique* ; dans ce point de vue il instancie seulement le composant mono-valué *carrosserie*.

Supposons que la classification se fasse en partant du point de vue *utilisation* ; le système descend l'instance jusqu'à la classe *transport de marchandises et matériaux*, seule classe permettant une valeur aussi élevée de *charge utile*. Grâce à la passerelle qui part de cette classe, le système déduit l'appartenance de la voiture à la classe *économique* du point de vue *physique*. Dans cette classe les voitures ont 4 roues (le nombre de roues devient connu) et une carrosserie normale. Le système peut donc instancier le composant *roues* de la voiture en créant 4 instances du concept *roue*, attachées aux classes racines, et il peut descendre l'instance de *carrosserie* attachée à l'objet composite, à la classe *normale* du point de vue *physique*. Cette descente entraîne aussi automatiquement la création des composants de *carrosserie*, c'est-à-dire 4 portes à poignée et 4 sièges simples.

---

[1] À la différence des attributs définis des instances composées, les attributs composants n'utilisent pas les descripteurs de domaine car celui-ci n'a pas de sens ; en effet, les composants étant crée après et pour les objets composites, leurs clés ne sont pas connues en avance et on ne peut pas les énumérer pour indiquer un domaine.

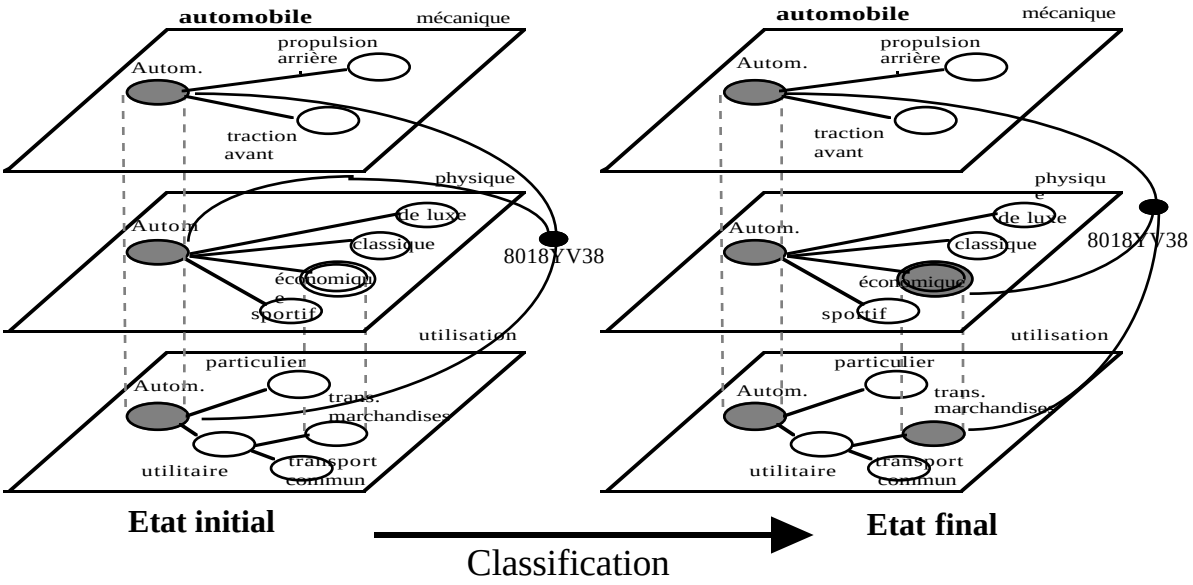


Fig. 8.9. Classification multi-points de vue d'un objet composite.

La classification d'objets composites en TROPES est un sujet de recherche actuel du groupe. En effet, plusieurs questions restent à résoudre. Quelle sémantique accorder à la diffusion de valeurs et comment l'exprimer ? Comment exprimer et gérer les contraintes existantes entre le tout et les parties et entres deux parties ? Et enfin, comment présenter et expliquer la classification d'objets composites [CRU92]. Par ailleurs, un mécanisme pour la génération automatique de versions d'un objet composite est en cours de réalisation. Ce mécanisme est une extension de la classification qui voit son application pour un modèle de tâches [GEN&92].

Conclusion

Dans ce chapitre nous avons présenté trois extensions à l'algorithme de classification : la relocalisation d'une instance de la base, la classification hypothétique d'instances et la classification d'objets composites.

La relocalisation d'instances concerne la reclassification d'une instance déjà classée dont l'information change. L'algorithme de relocalisation recherche les nouvelles plus petites classes sûre de l'instance modifiée, en partant des plus petites classes sûres de l'instance avant la modification et en remontant les arbres de différents points de vue le long des liens de sur-classes et des passerelles. L'avantage principal de cette démarche par rapport à une nouvelle classification de l'instance modifiée, à partir des racines de tous les points de vue, est qu'elle tire partie des vérifications de type qui ont été faites lors de la première classification de l'instance.

La classification hypothétique descend d'une instance incomplète à des classes possibles à partir d'un ensemble d'hypothèses. Deux types d'hypothèses ont été présentées : des hypothèses sur les valeurs des attributs de l'instance à classer et les hypothèses sur les classes d'appartenance de l'instance. Pour le premier type d'hypothèses, deux approches sont possibles. Dans la première approche, la classification manipule un seul contexte dans lequel les hypothèses de l'instance sont gérées de façon indépendante. La mise en cause d'une hypothèse peut entraîner une relocalisation de l'instance mais ne modifie pas les autres hypothèses. La deuxième approche suppose que

l'émission d'une hypothèse dépend des hypothèses précédentes : la mise en cause d'une hypothèse entraîne celle des hypothèses postérieures. Cette approche gère une pile de contextes hypothétiques : un contexte hypothétique est créé à chaque fois que l'instance descend d'un niveau, et il contient, outre l'état des marques des classes après la descente, les hypothèses qui ont permis cette descente. La relocalisation de l'instance lors de la mise en cause d'une hypothèse se fait en enlevant de la pile tous les contextes à partir du contexte qui contient cette hypothèse. L'ajout des tâches au modèle et la gestion des liens de justification et de dépendance entre attributs permet une classification hypothétique automatique.

Enfin, la classification d'objets composites étend la classification simple aux objets qui sont formés d'autres objets dont l'existence dépend de celle de l'objet composite. Deux modes sont possibles : la classification maximale qui classe tous les composants et la classification minimale qui classe seulement l'objet composite. La structure de composition arborescente de TROPES et la restriction des diffusion de valeurs de l'objet composite à ses parties, garantit la terminaison et l'unicité de l'algorithme de classification d'objets composites.

Ces trois extensions de la classification permettent de prendre en compte l'évolution, l'incomplétude d'une base de connaissances ainsi que la complexité de ses éléments. Elles ont été conçues comme des extensions indépendantes de la classification. Pour avoir une dynamique complète des instances de la base, il faut encore intégrer ces trois mécanismes entre eux et avec les mécanismes de mise à jour des classes (ajout, relocalisation et modification). Cette intégration permettrait de faire la classification hypothétique d'objets composites, la relocalisation des instances (simples ou composites) d'une classe qui migre ou encore relocaliser les instances hypothétiques de la base.

Bibliographie

[CRU92] CRUYPENNINCK F., *Interface de visualisation et explication du raisonnement par classification d'objets complexes.*, Mémoire d'Ingénieur en Informatique, Conservatoire National d'Arts et Métiers, CNAM, 1992.

[GEN90] GENSEL J., *Gestion des dépendances et des hypothèses dans un modèle de connaissances à objets*, Rapport DEA d'Informatique, INPG, Grenoble, 1990.

[GEN&92] GENSEL J., GIRARD P., *Expression d'un modèle de tâches à l'aide d'une représentation par objets*, Actes Représentation Par Objets (RPO), EC2, La Grande-Motte, France, pp. 225-236, 1992.

[MAR91] MARIÑO O., *Classification d'objets composites dans un système de représentation de connaissances multi-points de vue*, RFIA'91, Lyon-Villeurbanne, pp. 233-242, 1991.

[NEB90] NEBEL B., *Reasoning and Revision in Hybrid Representation Systems*, Lecture Notes in Artificial Intelligence, LNCS, vol. 422, Springer-Verlag, Berlin, 1990.

[ORS90] ORSIER B., *Evolution de l'attachement procédural : intégration de tâches, méthodes et procédures dans une représentation de connaissances par objets*, Rapport DEA d'informatique, INPG, Grenoble, 1990.

Conclusion

Conclusion

Dans ce travail nous nous sommes intéressés au raisonnement classificatoire. Ce type de raisonnement consiste à classer un objet dans une structure de spécialisation de catégories d'objets. La représentation de connaissances à base d'objets étant une des techniques de représentation les plus appropriées pour la classification, nous avons aussi étudié les caractéristiques de ce type de modèle. Ainsi, notre travail a porté sur deux aspects différents, mais fortement liés: la représentation de connaissances à base d'objets et le mécanisme de classification d'instances.

L'étude des représentations à objets a montré deux problèmes qui affectent aussi bien la représentation que la classification :

- la structuration de toute la connaissance dans un seul graphe et
- le conflit d'héritage dû à la spécialisation multiple.

Nous avons développé TROPES, un modèle de représentation de connaissances à objets multi-points de vue. Outre les éléments classiques d'une RCO (classe, instance, attribut, facette, lien est_un, lien sorte_de), TROPES introduit les notions de concept, de point de vue et de passerelle. Le concept permet de résoudre le premier problème posé. Les points de vue offrent la possibilité de regarder une base selon un aspect particulier et d'en avoir une vision partielle et pertinente. Le problème du multi-héritage disparaît presque complètement avec l'introduction des points de vue. Les passerelles établissent les communications entre deux points de vue et représentent ainsi la collaboration inter-disciplinaire.

Une base de connaissances TROPES est divisée en familles d'objets disjointes, appelées concepts. Le concept est

- l'espace de définition des classes : les graphes de classes des différents concepts ne sont plus liés par la classe racine objet, ni par aucune autre classe. Chaque concept a sa propre structure de classes, ce qui empêche la création de classes partagées par deux concepts.
- l'espace de définition des attributs : un attribut a le même sens partout dans le concept et n'est pas connu ailleurs. Ce choix évite d'avoir deux attributs différents avec le même nom et élimine ainsi le conflit d'héritage de nom.
- l'espace de définition des instances : une instance n'est plus une entité dont l'existence et les caractéristiques sont subordonnées à sa classe d'appartenance, mais un objet ayant une structure minimale de départ et une identification à l'intérieur de son concept.
- l'espace de travail des différents mécanismes de raisonnement, en particulier de la classification. L'espace de recherche se réduit au concept.

Une des notions les plus originales de TROPES est celle de point de vue. Un point de vue d'un concept détermine aussi bien les attributs du concept qui vont être visibles de ce point de vue qu'une organisation particulière du graphe de classes. Ainsi un concept peut être manipulé à partir d'un seul point de vue sans se préoccuper des autres, ce qui réduit le nombre d'attributs et de classes à regarder et simplifie le graphe de

spécialisation de classes. Enfin, la communication entre les différents points de vue établie par des passerelles rend possible le partage d'informations entre les classes de différents points de vue.

Ce modèle de représentation de connaissances a permis le développement d'un algorithme de classification multi-points de vue qui fournit la classe la plus spécialisée pour une instance d'un concept dans chaque point de vue de ce concept. La classification dans TROPES tire parti des passerelles pour faire descendre l'instance plus rapidement dans un point de vue à l'aide des inférences faites dans d'autres points de vue. De plus, la structure multi-points de vue des concepts permet de faire une classification partielle d'une instance en ne considérant qu'un sous-ensemble des points de vue et donc d'attributs du concept.

L'algorithme de classification multi-points de vue s'appuie sur en une boucle composée de cinq procédures : obtention d'information, appariement, propagation de marques, mise à jour de l'information et choix du prochain point de vue. Cet algorithme est paramétrable, modulaire, correct et efficace.

Il est **paramétrable**, car l'utilisateur peut ordonner les points de vue du concept ; cet ordonnancement va déterminer le choix des points de vue à parcourir lors de la classification. La **modularité** a été confirmée par le développement des extensions : la classification hypothétique, la relocalisation et la classification d'objets composites. En effet, ces extensions s'obtiennent en ne modifiant que certaines procédures de la boucle.

L'algorithme est **correct**, il termine et converge, garantissant ainsi une réponse unique à la classification d'une instance. Par ailleurs, le calcul de la complexité a montré **l'efficacité** de l'algorithme. Les procédures les plus coûteuses de la boucle de classification sont l'appariement et la propagation de marques. La complexité de l'appariement est déterminée par celle de la procédure de validation du type d'un attribut. L'implémentation de l'algorithme fait une normalisation des types des attributs lors du chargement de la base pour réduire cette complexité. Le coût de la propagation de marques est réduit le plus possible, il dépend de la hauteur des arbres de classes des points de vue, du nombre de passerelles et du nombre de points de vue du concept.

Le prototype développé en CAML a permis de tester aussi bien le modèle que la classification multi-points de vue.

En plus des trois extensions déjà proposées dans cette thèse (relocalisation d'instance, classification hypothétique, et classification d'objets composites), le développement de ce travail a souligné aussi certaines voies de recherches tout à fait intéressantes:

- Le modèle TROPES travaille sur l'hypothèse de l'exclusivité entre les classes sœurs du graphe de classes associé à un point de vue. Cette hypothèse implique que les classes sont organisées en une structure d'arbre dans un point de vue et qu'il n'y a qu'une classe sûre unique par niveau lors de la classification. Cette hypothèse a été établie de façon empirique par une analyse des exemples présentés dans les articles traitant l'héritage multiple. En effet, la restructuration de ces exemples avec des points de vue éliminait complètement la spécialisation multiple. Même si, en théorie, on peut toujours ajouter un point de vue pour éviter une spécialisation multiple, cette solution peut ne pas être très naturelle. Nous proposons donc de supprimer l'hypothèse de l'exclusivité sauf dans le cas où elle peut être garantie statiquement par intersection des domaines des classes.
- Les attributs ont tous le même statut dans la description d'un concept et sont donc tous déterminants lors de la classification. Il paraît intéressant de pouvoir distinguer deux sortes d'attributs: ceux qui sont déterminants et ceux qui ne le sont pas pour la

classification. Cette dernière catégorie d'attributs correspond alors à un ensemble d'attributs à déterminer. Puisqu'ils ne sont plus considérés pour l'établissement de l'appartenance à une classe, on peut alors leur adjoindre dans la description d'une classe tout type d'informations sur la façon de les calculer (attachement procédural, satisfaction de contraintes…). La classification d'instance se basant sur les attributs déterminants pourra alors alterner la phase d'appariement avec une phase de production d'informations par calcul des attributs non déterminants.

- Une base de connaissances écrite en TROPES peut être très complexe, et la classification d'instance dans une telle base peut donc devenir difficile à suivre. L'étude et le développement d'une interface conviviale et d'un module d'explication sont donc nécessaires. À l'instar de SHIRKA, la visualisation des graphes de classes et de la classification d'instance est pertinente et utile. Les nouvelles notions (concept, point de vue, passerelle) introduites doivent être prises en compte dans la représentation graphique et l'explication de la classification.

Les perspectives à plus long terme, qui sont déjà les préoccupations actuelles de recherche du projet SHERPA, visent à étendre et à adapter le modèle TROPES à de nouvelles fonctionnalités :

- L'une des premières extensions consiste en l'intégration de la notion de contrainte au sein de la définition d'une classe. Si la logique de représentation reste la même, les capacités d'expression et les possibilités d'inférence sont quant à elles grandement augmentées. En effet, les contraintes permettent d'exprimer des dépendances entre attributs, tandis que le mécanisme de satisfaction de contraintes se charge, d'une part, de garantir l'intégrité des instances, et d'autre part, d'inférer dès que possible les valeurs d'attribut manquantes.
- Un modèle de tâches, dédié à la représentation de la connaissance stratégique, est en cours de spécification. Les originalités de ce modèle sont de représenter une tâche comme un objet composite, de représenter le contrôle par un ensemble de contraintes entre attributs, et de ramener l'exécution d'une tâche à une classification particulière d'objet composite. Ce modèle de tâches exploite donc au maximum la déclarativité offerte par TROPES ainsi que ses mécanismes d'inférence.
- Les problèmes de l'acquisition incrémentale de connaissances sont abordés en se focalisant sur la manipulation de classes. Ainsi, un mécanisme de classification de classes est étudié pour le modèle TROPES. Pour ce faire, il exploite la notion de type qui est sous-jacente à celle de classe. Un résultat attendu de ce travail est aussi la vérification de la cohérence structurelle d'une base.
- De par son orientation inter-disciplinaire, le modèle TROPES va devoir faire face à deux problèmes réels : la grande quantité de connaissances et la coopération entre les différents utilisateurs. Actuellement, un travail est mené sur la prise en compte du partage de la base de connaissances par plusieurs utilisateurs.

Enfin, une première application réelle de TROPES est en cours de développement dans le domaine de la biologie moléculaire. L'objectif est de fournir aux biologistes moléculaires un environnement coopératif d'analyse de séquences génomiques. Ce domaine scientifique réunit les caractéristiques nécessaires à la validation du modèle, et offre un large champ d'investigation pour les futures versions de TROPES. En effet, il considère de grandes bases de connaissances de natures diverses : descriptives, méthodologiques, textuelles et graphiques.

Annexe

Syntaxe du langage de TROPES

Annexe

Syntaxe du langage de TROPES

Nous donnons dans cette annexe la grammaire définissant la syntaxe du langage TROPES. Cette grammaire est utilisée pour générer une base de connaissance TROPES à l'aide du générateur d'analyseur syntaxique YACC. La base résultante se trouve sous forme de structures de données du langage CAML. La grammaire, directement extraite des fichiers YACC, est donnée sous forme BNF.

Grammaire générale

Cette grammaire donne la structure générale d'une base de connaissances. C'est un ensemble de bases qui se décomposent elles-mêmes en un ensemble de concepts. Il y a deux types de concepts: ordonnés ou non-ordonnés.
grammar for values gramtops =

```
Top     ::      Base b; Concepts lco; Attributs la

Base    ::      "base"; "{"; IDENT n;
                "concepts"; Listv co; ";" ;
                "points_de_vue"; Listv pvl; "}"

Concepts    ::  Unconcept c
            |   Concepts fsts; Unconcept c

Unconcept   ::  "concept"; Sconc c
            |   "concept_ordonne"; Sconord c

Sconc   ::      "{"; Infc (c,clef,pvl); "}"

Sconord     ::  "{"; Infc (c,clef,pvl); ";" ; "fct_ordre"; IDENT f; "}"

Infc    ::      IDENT c; "clef"; Listv clef; ";" ; "points_de_vue"; Listv pvl

Attributs   ::  "attribut"; Sattr a
            |   Attributs fsts; "attribut"; Sattr a

Sattr   ::      "{"; IDENT n ; Infa lat; "}"

Inf     ::      UnatAtt att
            |   Infa fsts; ";"; UnatAtt att
```

```
UnatAtt     ::    "concept"; Clenom s
              |   "valeur_dans"; Clenom s
              |   “tache"; Clenom f
              |   "table_tache"; Clenom t
              |   "commentaire"; STRING c
              |   "nature"; STRING s
              |   "fichier_image"; STRING s
              |   "multivaleur"; STRING b

Clenom      ::    "#";"["; STRING cle ; "]"
              |   IDENT cle

Listv       ::    "["; Listvirg l; "]"

Listvirg    ::    IDENT a
              |   Listvirg fsts; ","; IDENT a
```

Grammaire pour les instances

Grammaire donnant la syntaxe pour exprimer un ensemble d’instances avec leurs attributs et leurs valeurs.
grammar for values inst ::

```
Instances   ::    { typg "Term"} expt; "{"; Instance (id,attval_list); "}"

Instance    ::    IDENT id; Campos listatt
              |   Campos listatt

Campos      ::    Att1 l
              |   Campos ll; ";"; Att1 l

Att1        ::    IDENT atnom; "="; Atvalg a

Atvalg      ::    Atvaln a
              |   Atvalv a

Atvaln      ::    "?"
              |   "~"

Atvalv      ::    NUM n
              |   "-"; NUM n
              |   STRING s
              |   IDENT id
              |   "#"; Atlisc v
              |   Atlis v

Atlisc      ::    "["; Comp v; "]"

Atlis       ::    "("; Comp v; ")"
              |   "(";")"

Comp        ::    Atvalv a
              |   Comp fs; ","; Atvalv a
```

Grammaire pour les passerelles

grammar for values Pass=

```
Passerelles  ::   Unpass p
             |    Passerelles fsts; Unpass p

Unpass       ::   "pass";"{";"de";"=";{ typg "Term"} lldepart;
                  "vers";"=";Nomclas (co,cl,pv); "}"

Nomclas      ::   IDENT co; Literal "."; IDENT cl; Literal "/"; IDENT pv
             |    IDENT cl; Literal "/"; IDENT pv
             |    Clenom c

Clenom       ::   "#"; "["; STRING cle; "]"
             |    IDENT cle
```

Grammaire pour la description de classe

Cette grammaire est en charge d'analyser une description de classe, et donc tous les types de descripteur d'attributs. On y retrouve toutes les formes de facette possibles: "un", "ens-de, "domaine", "card", "sauf",…
grammar for values gramtydom ::

```
Classe       ::   "classe"; Sclass c

Sclass       ::   "{";Nomclas(c,cl,pv); Infcl(sc,lat);"}"

Infcl   ::        "sorte_de"; Clenom sc; ";"; Attclass lat
             |    Attclass lat

Attclass     ::   Unattcl atc
             |    Attclass fsts ; ";"; Unattcl atc

Unattcl      ::   IDENT atnom; Facettes f

Facettes     ::   Descripteur d
             |    Facettes fsts ; Descripteur d

Descripteur  ::   "un"; Typexp expt
             |    "ens_de"; Typexp expt
             |    "domaine"; Domaine (s,int)
             |    "sauf"; Domaine (s,int)
             |    "card"; Intcard s
             |    "cond"; Listvirg lp
             |    "calcule"; Mstring lc
             |    "defaut"; Atvalv a
             |    "valeur"; Atvalv a

Typexp       ::   Termes exp
```

```
Termes      ::    Term t
              |   Termes tr; "|"; Term t

Term        ::    Fact f
              |   Term fr; "&"; Fact f

Fact        ::    "("; Typexp exp; ")"
              |   Nomclas cle

Nomclas     ::    IDENT co; Literal "."; IDENT cl; Literal "/"; IDENT pv
              |   IDENT cl; Literal "/"; IDENT pv
              |   Clenom c

Intcard     ::    Unint x
              |   Atvalv x
              |   Intcard fst; ","; Unint x
              |   Intcard fst; ","; Atvalv x

Domaine     ::    Unint x
              |   Atvalv x
              |   Domaine (s,int); ","; Unint x
              |   Domaine (s,int); ","; Atvalv x

Unint       ::    Binf bi;":"; Bsup bs

Binf        ::    "["; "-"; "inf"
              |   "]"; "-"; "inf"
              |   "["; Atvalv vx
              |   "]"; Atvalv vx

Bsup        ::    "+"; "inf"; "["
              |   "+"; "inf"; "]"
              |   Atvalv vx; "]"
              |   Atvalv vx; "["

Atvalv      ::    NUM n
              |   "-"; NUM n
              |   STRING s
              |   "#"; Atlisc v
              |   Atlis v

Atlisc      ::    "["; Comp v; "]"

Atlis       ::    "("; Comp v; ")"
              |   "(";")"

Comp        ::    Atvalv a
              |   Comp fst; ","; Atvalv a

Clenom      ::    "#";"["; STRING cle ; "]"
              |   IDENT cle
```

```
Listvirg    ::  IDENT a
            |   Listvirg fst; ","; IDENT a

Mstring     ::  STRING s
```

Grammaire pour les types

Cette grammaire permet d'analyser les expressions logiques ("et" et "ou") contraignant un attribut abstrait à appartenir aux classes spécifiées dans l'expression.
grammar for values Type::

```
Typexp      ::  Termes exp

Termes      ::  Term t
            |   Termes tr; "|"; Term t

Term        ::  Fact f
            |   Term fr; "&"; Fact f

Fact        ::  "("; Typexp exp; ")"
            |   Nomclas cle

Nomclas     ::  IDENT co; Literal "."; IDENT cl; Literal "/"; IDENT pv
            |   IDENT cl; Literal "/"; IDENT pv
            |   Clenom c

Clenom      ::  "#"; "["; STRING cle; "]"
            |   IDENT cle
```