Construction hypothétique d'objets complexes

Pierre Girard

N° d'ordre:

THÈSE

présentée à

L'UNIVERSITÉ JOSEPH FOURIER
LABORATOIRE LIFIA/IMAG

pour obtenir le titre de

DOCTEUR

Spécialité

INFORMATIQUE

par

Pierre GIRARD

Sujet de la thèse :

CONSTRUCTION HYPOTHÉTIQUE D'OBJETS COMPLEXES

Soutenue le 26 octobre 1995 devant le jury composé de :

Mme	**Marie-France**	**BRUANDET**	Président
MM.	**François**	**RECHENMANN**	Directeur
	Bernard	**CARRÉ**	Rapporteurs
	Roland	**DUCOURNAU**	
	Gia Toan	**NGUYEN**	Examinateur

Je tiens à remercier...

Monsieur Roland Ducournau, Professeur à l'Université de Montpellier, et Monsieur Bernard Carré, Maître de Conférences à Lille, d'avoir tous deux accepté de juger ce travail et de m'avoir aidé à améliorer ce mémoire par leurs remarques et conseils. Je les remercie également pour leur disponibilité.

Madame Marie-France Bruandet et Monsieur Gia Toan NGuyen d'avoir accepté de participer au jury de thèse. Je leur suis reconnaissant de l'intérêt qu'ils ont bien voulu porter à mon travail.

Monsieur François Rechenmann, Directeur de Recherche à l'INRIA Rhône-Alpes, de m'avoir accueilli dans son équipe. Je le remercie tout particulièrement pour sa confiance, ses encouragements et surtout sa patience.

Les membres de l'équipe SHERPA :

Patrice pour son aide et sa gentillesse (je vous le dis, Sacré Mac !), Pierre pour sa disponibilité, Jérôme et Gilles pour leur lecture de mon document et leurs conseils, Danielle ; Mathias, compagnon des premiers temps, ainsi que Bruno, Olivier, Jutta, les anciens ; Petko, Jean-Yves, Sueli, Ysabelle, les nouveaux.

Je remercie Jean-Marc, Nathalie, Sylvain, l'équipe de nuit bien sympathique, ainsi que la "petite syrienne" Nina pour ses desserts (hummmmm !) et sa compagnie réconfortante lors des nombreux soirs et week-end au labo.

Je remercie particulièrement Françoise, toujours si patiente, disponible et efficace. Et bien sûr Flo pour son café (que je n'ai jamais payé !) et surtout pour sa gentillesse et sa présence dans les moments difficiles.

Je remercie tout particulièrement Jerôme (le Gensel), mon complice de tous les instants, et Cécile, notre astrologue de l'après RU mais surtout une collègue bien sympathique et dévouée (c'est peu dire). Ce n'est pas un hasard si, avec ces deux autres marathoniens, nous avons franchi la ligne d'arrivée en même temps... Merci pour m'avoir soutenu, tout en courant vous-mêmes.

Tous mes amis :

Pascal, Patrice et Denis, mes amis de "10 ans déjà... vous vous rendez compte ?", pour tout... simplement.

Riton et Laurette, pour avoir été ma famille grenobloise, pour le réconfort que j'ai toujours trouvé auprès de vous,

Alejo, Arnie, Olga (la princesa), Pepe avec qui j'ai découvert le(s) nouveau(x) monde(s) dans une ambiance des plus chaleureuses et des plus rythmées. Merci d'avoir apporté tant de bons moments à l'époque "thèse".

Merci Jéronimo (t'as bien le droit au cumul), merci Marie-Laure, notamment, pour avoir (trop) souvent supporter nos discussions laborieuses.

Un grand merci international à mes amis colombianitos : Coquita, Brunito (par adoption), Claudia, Miguel (par annexion), Fernando, Harold (et son planning), Mary, Rodrigo ; vénézuélien : Carlos ; mexicains : Pepe2, Elizabeth et Nestor ; espagnols : Carmen et Vicente ; et bien sûr aux autochtones : Juan-Maria et Jean.

Et enfin un graciasototote à mi corazoncita et à ma famille, ma mère, Stéf, Marie-Anne, pour leur affection, leur soutien et leur confiance.

A Rubby et à ma mère

TABLE DES MATIERES

Introduction

INTRODUCTION

1. Contexte général

Dans un système à base de connaissances, la séparation entre les connaissances et les mécanismes qui permettent leur exploitation est un principe qui tend à favoriser l'utilisation multiple d'un même ensemble de connaissances. Toutefois, pour donner à ce principe toute son ampleur, il est primordial que les connaissances d'un tel système soient représentées dans un modèle capable d'en capturer les différents aspects et propriétés.

La représentation des connaissances est un domaine qui a été et qui est toujours très étudié. Différentes problématiques liées aussi bien à la conception qu'à l'exploitation des systèmes à base de connaissances — comme l'acquisition des connaissances, l'étude de mécanismes de raisonnement, l'explication du raisonnement, etc. — ont contribué à dégager des propriétés importantes que doit posséder un formalisme de représentation des connaissances.

A ce titre, la structuration des connaissances est devenu un thème central. La concision de la description des connaissances et la diversification des mécanismes d'inférence sont des problèmes fortement liés à celui de la structuration des connaissances.

La représentation par objets propose une organisation structurée des connaissances qui apporte des éléments de réponse à ces deux problèmes. En effet, l'organisation hiérarchique prônée par les modèles à objets s'avère être un support permettant à la fois une description factorisée et la mise en place de divers types de raisonnement, ou schémas d'inférence.

Le domaine de la représentation des connaissances par objets se ramifie en sous-domaines qui se distinguent par les différentes propriétés de la notion d'objet qu'ils mettent en avant et les types de raisonnement qu'ils cherchent à mettre en place. Parmi ces sous-domaines, nous nous intéressons plus particulièrement aux modèles à objets proposant une distinction entre la notion de classe et celle d'instance.

Dans ce type de modèle, un objet est perçu à travers la donnée de ses attributs. Si cet objet est une instance, il est un individu particulier de l'univers du discours, et ses attributs sont vus comme des valeurs; si c'est une classe, l'objet est alors une abstraction représentant un ensemble d'instances (individus), les attributs sont alors vus comme des descriptions, i.e. des domaines de valeurs. La hiérarchie de classes organisée par une relation de spécialisation décrit l'univers des instances.

2. Introduction à la problématique

La distinction entre les notions de classe et d'instance favorise deux types de raisonnement : l'*identification* et la *construction* d'objets.

D'une part, la hiérarchie de classes devient un support intéressant pour guider un raisonnement classificatoire. La classification de classe permet de trouver la place à laquelle insérer une classe dans la hiérarchie de spécialisation, tandis que la classification d'instance, aussi appelée *identification* ou *reconnaissance*, permet de trouver les classes d'une hiérarchie de spécialisation à laquelle une instance peut ou doit appartenir.

D'autre part, une classe constitue un contexte adéquat pour rassembler les connaissances décrivant une ou des méthodes de *construction* de ses instances. Dans les modèles à objets qui s'inspirent des langages de *frames* et des langages de programmation par objets, une classe peut contenir des connaissances indiquant comment gérer et évaluer les attributs de ses instances. Les connaissances d'une classe peuvent être exploitées dès qu'une instance lui est rattachée. La donnée d'un minimum de connaissances sur une instance peut alors permettre sa construction.

Différents domaines d'application, comme l'aide au diagnostic ou l'aide à la conception d'objets, doivent résoudre des problèmes qui requièrent la mise en place d'un raisonnement capable de gérer à la fois l'identification et la construction d'objets. Pour de tels domaines, un modèle à objets capable de représenter des connaissances exploitables aussi bien pour l'identification que pour la construction d'objets constituerait un support adéquat.

Concilier l'identification et la construction d'objets au sein d'un même modèle à objets est la problématique générale à laquelle ce mémoire s'intéresse. L'étude de cette problématique a pour objectif final la proposition d'un mécanisme général de construction d'objets par exploration d'une hiérarchie de classes. Ce mécanisme est baptisé, par la suite, *mécanisme d'identification/construction d'instances.*

La prise en compte de cette problématique au niveau de la représentation des objets requiert l'étude des deux aspects complémentaires d'un modèle à objets :

- *l'aspect représentation* : il s'agit de définir un modèle dans lequel la description des objets puisse rendre compte à la fois des aspects liés à leur identification et à leur construction.
- *l'aspect raisonnement* : il s'agit de définir le type général de raisonnement qui doit être mené pour résoudre un problème d'identification/construction d'instances à partir d'une description d'objets donnée.

3. Le cadre du travail

L'ensemble des propositions qui émergent de ces études prennent place dans le modèle à objets TROPES. Ce modèle propose un formalisme de description des objets qui obéit aux restrictions imposées par un raisonnement classificatoire.

Le langage de description de ce modèle est étendu afin de permettre l'intégration de différents mécanismes d'évaluation d'attributs. Parmi ceux-ci nous prenons en compte l'intégration de la notion de contrainte proposée dans le cadre d'un autre travail de thèse [Gens95].

Les problèmes de contrôle liés à la gestion simultanée de ces divers mécanismes sont soulevés. Ils font l'objet d'une proposition de modèle permettant d'exprimer et de développer la stratégie d'évaluation d'un attribut en fonction du raffinement d'une instance dans une hiérarchie de classes. Dans ce contexte, une solution au problème de la construction d'objet composite, c'est-à-dire le problème des échanges de connaissances entre un objet et ses composants, est aussi proposée.

Enfin, à partir de ce modèle TROPES étendu, un mécanisme d'identification/construction d'instances est mis en œuvre. L'originalité de ce mécanisme réside dans le caractère hypothétique du raisonnement mené. Il s'appuie sur la notion d'hypothèses pour poursuivre l'exploration d'une hiérarchie de classes au delà des limites atteintes par un mécanisme de classification d'instances. Ce mécanisme délivre pour résultat les différentes versions d'instance qui répondent au problème de construction qu'un utilisateur avait soumis.

4. Les contributions à la représentation par objets

Ce travail cherche à contribuer au domaine de la représentation des connaissances à objets par :

- la mise en évidence et la compréhension du rôle dual d'identification et de construction de la description des objets.
- la proposition d'extension d'un modèle à objets de type classificatoire vers un modèle permettant aussi la construction d'instances.
- la proposition d'une extension de l'architecture d'un modèle à objets afin de permettre le développement d'un raisonnement hypothétique sur les objets.

- la proposition de la mise en œuvre d'un mécanisme, dit d'*assistance hypothétique*, capable de dépasser les insuffisances du schéma déductif de la classification pour proposer différentes solutions possibles à un problème d'identification/construction d'instances.

5. L'organisation du document

Le document comprend huit chapitres organisés en trois parties.

La première partie, composée de trois chapitres, présente le contexte général du travail.

Le chapitre I décrit l'architecture et le fonctionnement des systèmes à base de connaissances et introduit la représentation des connaissances structurées. Le chapitre II présente les concepts de la représentation à objets et ses différents mécanismes d'exploitation. Enfin, le chapitre III introduit le raisonnement hypothétique, la gestion de la notion d'hypothèse au sein d'une base de connaissances et son application dans le contexte d'une base d'objets.

La seconde partie, composée de deux chapitres, est consacrée à l'étude de la dualité entre la construction et l'identification d'objets.

Le chapitre IV présente la classification d'instance incomplète comme un bon moyen d'exploiter l'identification pour guider la construction d'une instance. Cette approche est toutefois limitée par les possibilités de description d'un modèle classificatoire. L'étude de ces limites se concrétise par la mise en évidence de deux catégories de systèmes à objets : les uns présentent la classe comme une unité d'identification d'instances et les autres la présentent comme une unité de construction d'instances. Ce chapitre s'achève sur une présentation d'application et de systèmes qui adoptent une notion de classe mixte permettant de décrire des connaissances à la fois pour l'identification et pour la construction d'instances. Le chapitre V analyse et critique les différentes approches présentées dans le chapitre précédent pour proposer une nouvelle approche du problème d'identification/construction d'instances.

La troisième partie, composée de trois chapitres, décrit la mise en œuvre dans le modèle TROPES des propositions formulées dans la partie précédente.

Le chapitre VI s'intéresse à la mise en place de mécanismes d'évaluation d'attributs. L'intégration du module de contraintes au modèle TROPES est présentée. De plus, nous proposons l'intégration d'un mécanisme global d'évaluation des attributs, appelé *détachement procédural*. Le chapitre VII s'intéresse aux problèmes soulevés par l'intégration de divers mécanismes d'évaluation au sein du modèle TROPES. La proposition d'un modèle de *connaissances de production* permet de gérer le contrôle de l'évaluation des attributs d'une instance. Le chapitre VIII présente une extension de l'architecture du modèle TROPES par l'intégration d'un module dit d'*assistance hypothétique*. Ce module propose un mécanisme de configuration hypothétique d'instances permettant à un utilisateur d'exprimer explicitement et de résoudre ses problèmes d'identification/construction d'instances.

Partie 1

REPRESENTATION PAR OBJETS ET RAISONNEMENT

Chapitre I

SYSTEME A BASE DE CONNAISSANCES

1. Introduction

L'approche des premiers travaux sur la résolution de problèmes en Intelligence Artificielle (I.A.) consiste à exploiter en priorité les capacités de traitement des ordinateurs pour fournir des techniques générales de résolution.

Le principe adopté alors se résume en un processus de recherche de solutions dans un espace d'états. Le problème est fourni par la donnée d'un état initial et d'un ou plusieurs états but. L'espace d'états est, quant à lui, formalisé par un ensemble d'opérateurs qui permettent sa construction incrémentalement, c'est-à-dire la description du passage d'un état à un autre. Le raisonnement piloté par un algorithme général de recherche (A*, "générer et tester", AO*, "moyens-fins",... [Rich87]) exploite les opérateurs pour construire les états successifs menant à un état but. Le modèle STRIPS [Fike&71] et le "Resolveur Général de Problèmes" (GPS : General Problem Solver) [Newe&63b] fournissent de bons exemples de cette première approche. Le fonctionnement de ces systèmes basé sur un algorithme général et une description d'un espace de travail, laisse entrevoir le comportement général des systèmes d'I.A. réalisés par la suite.

Cependant, leur trop grande généralité devient vite un problème insurmontable quand il s'agit de s'attaquer à des domaines d'applications réelles nécessitant un grand nombre d'informations. Bien adaptées à des problèmes bien définis comme les jeux et la démonstration de théorèmes, les notions trop simples d'état et d'opération de transition d'état ne suffisent pas à décrire des problèmes complexes comme le diagnostic, le traitement des langues naturelles, etc.

C'est ainsi que l'un des premiers résultats rapidement mis en évidence dans les 20 premières années, est que l'intelligence requiert des "connaissances". Newell définit l'intelligence comme étant "l'application de la connaissance à la résolution de problèmes" [Newe81]. Cette notion de connaissances prend forme dans les systèmes qui apparaissent dans les années soixante-dix comme le système de diagnostic médical MYCIN [Shor76]. A cette époque, ils sont appelés systèmes "experts" parce qu'ils réunissent un ensemble de connaissances acquises auprès d'un "expert" sous formes de règles et de faits (cf. §I.2). Ces connaissances exploitées par le moteur d'inférence permettent de développer un raisonnement en produisant de nouvelles connaissances dans un but précis.

Ces systèmes, rebaptisés maintenant plus modestement systèmes "à base de connaissances de première génération", fournissent le principe général de fonctionnement d'un **système à base de connaissances**. L'étude et l'expérimentation de ces premiers systèmes ont ouvert de nouvelles et nombreuses directions de recherche comme la représentation des connaissances, l'explication du raisonnement, l'acquisition des connaissances, l'adaptation, la spécialisation et la diversification des mécanismes de raisonnement en fonction du type de problème à résoudre...

> Un raisonnement peut être défini comme un enchaînement d'énoncés ou de représentations symboliques conduit en fonction d'un but, ce but pouvant prendre des formes variées : démontrer, convaincre, élucider, interpréter, décider, justifier, expliquer, etc. [...] Ainsi le raisonnement dans un système à base de connaissances peut être schématisé comme un enchaînement de découvertes d'éléments de connaissances s'appuyant sur les informations connues, menant au but recherché. [Hato&91]

La représentation des connaissances joue un rôle central dans un tel système car elle détermine, par l'organisation des informations qu'elle propose, les capacités inférencielles, explicatives et acquisitionnelles. Deux approches de représentation duales sont proposées : l'une dite déclarative et l'autre procédurale.

La première s'intéresse plus particulièrement à la partie structurelle de la connaissance, c'est-à-dire à la description de la connaissance, et la sépare clairement du contrôle qui en indique l'utilisation. Cette approche facilite l'accessibilité, l'extension, et la modification des connaissances.

La deuxième, à l'instar d'un langage de programmation classique, encapsule contrôle et connaissances au sein d'un programme, ou procédure. L'utilisation de connaissances procédurales favorise l'efficacité et correspond à des situations dans lesquelles connaissances et contrôle peuvent être figés.

En général, dans un système à base de connaissances, l'approche est globalement déclarative et, pour des raisons d'efficacité ou pour remédier à un manque d'expressivité, des connaissances procédurales sont introduites localement (par exemple, par attachement procédural dans les *frames* [Wino75]). Toutefois, le haut degré de déclarativité est considéré comme une propriété importante et recherchée d'une représentation de connaissances car elle offre un éventail plus large de possibilités d'exploitation d'un même ensemble de connaissances.

1.1. Architecture d'un système à base de connaissances

Un système à base de connaissances se répartit en deux composantes principales (cf. Figure I.1) : la base de connaissances et le moteur d'inférence.

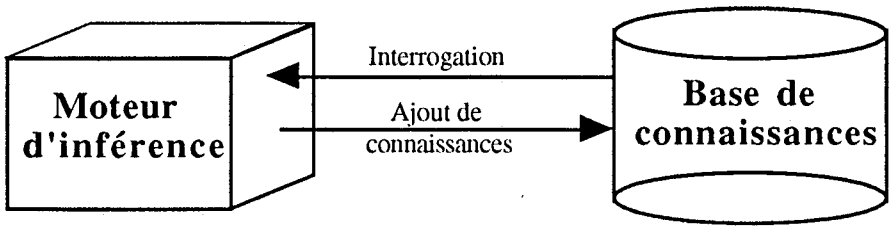

Figure I.1 : Architecture générale d'un système à base de connaissances. Les connaissances de l'application sont formalisées et réunies dans la base de connaissances. Le moteur d'inférence fournit un ensemble de mécanismes permettant d'exploiter les connaissances et d'en produire de nouvelles.

La base de connaissances réunit les connaissances nécessaires à l'application en charge du système. Ces connaissances sont codées dans le formalisme de représentation particulier fourni par le système.

Le moteur d'inférence propose, quant à lui, l'ensemble des primitives qui permettent d'exploiter les connaissances pour résoudre des problèmes. Raisonner consiste donc à appliquer un ensemble de mécanismes d'inférence sur la base de connaissances jusqu'à obtention du but recherché.

Les mécanismes d'inférence sont spécifiques à la représentation de connaissances utilisée ; ils rendent compte de la sémantique particulière de celle-ci et permettent de produire de nouvelles connaissances. Ils peuvent être simples comme l'activation-déclenchement de règles dans les systèmes à base de règles (cf. §I.2), ou plus complexes et précis comme l'héritage dans les représentations par objets et la classification dans les langages terminologiques (cf. §I.3).

1.2. Base de connaissances

La base de connaissances est donc chargée de rassembler les connaissances nécessaires à une application. La façon de représenter ces connaissances est déterminante puisqu'elle est porteuse d'une sémantique que les mécanismes d'inférence du système exploitent pour développer un raisonnement. La construction de la base de connaissances d'une application est donc l'opération cruciale par laquelle on définit les capacités de résolution du système.

Derrière le terme unitaire "connaissance" se cachent de nombreux concepts et nuances qu'il convient de caractériser pour souligner le rôle, l'importance et la pluralité des formalismes de représentation des connaissances utilisés.

1.2.1. Caractérisation des connaissances

Les connaissances nécessaires à une application réelle sont souvent nombreuses et hétérogènes. La modélisation d'un problème à résoudre peut demander l'acquisition de nombreuses connaissances décrivant à la fois les lois qui régissent le domaine d'application et les données manipulées. Cette description se répartit en général sur plusieurs niveaux (conceptuels) de connaissances :

- les connaissances factuelles, qui permettent de décrire une utilisation particulière de la connaissance. Elles définissent un état de la base de connaissances. Par exemple, la donnée d'un problème particulier et l'obtention d'une solution particulière sont des connaissances factuelles. Ce type de connaissances correspond aux faits dans les systèmes à base de règles et aux instances dans les systèmes à objets.
- les connaissances "générales, ou "théoriques", du domaine modélisé, dans lesquelles peuvent encore être distinguées des sous-catégories, comme par exemple :
  - des connaissances consacrées à la définition des entités du domaine modélisé ;
  - des connaissances dénotant les dépendances et relations possibles entre les différentes entités ;
  - des connaissances stratégiques qui permettent de décrire des raisonnements complexes et spécifiques qui définissent eux-mêmes les fonctionnalités attendues de l'application.
  - etc.

De plus, il peut être nécessaire d'attribuer aux connaissances des caractères particuliers permettant d'indiquer leur degré de croyance, leur aspect hypothétique ou non, leur aspect approximatif, leur aspect temporel... lorsque le domaine d'application manipule des connaissances incertaines, incomplètes, imprécises et/ou évolutives.

1.2.2. Représentation des connaissances

Un formalisme de représentation des connaissances se veut une théorie tentant de décrire la notion de connaissance elle-même. Un tel formalisme offre divers concepts (ou constructeurs) à partir desquels peuvent être décrites les connaissances. A chacun de ces concepts est associée une sémantique spécifique qui détermine son comportement dans le contexte d'un raisonnement.

La logique mathématique constitue l'un des premiers domaine étudié en I.A., notamment lors de la réalisation de programmes de démonstration de théorèmes comme *The Logic Theorist* [Newe&63a]. De ce fait, et parce qu'elle fournit un cadre formel bien connu décrivant à la fois des vérités du monde et des mécanismes de raisonnement sur ces vérités, la logique mathématique a rapidement été pressentie comme un formalisme de la représentation des connaissances. Plus particulièrement, la logique des prédicats a motivé la réalisation de nombreux langages de représentation ou de langage de programmation logique comme PROLOG [Colm&83]. Elle reste en général une référence à laquelle les représentations

d'aspiration moins formelle, ou moins classique, peuvent se confronter. Toutefois, la trop grande rigidité qu'impose ce formalisme limite ou rend difficile l'expression de connaissances qui ne se suffisent pas de la simple distinction vrai-faux (connaissances incertaines, incomplètes, imprécises...).

Les systèmes à base de règles (cf. §I.2), qui proposent simplement la notion de fait et de règle comme éléments de description des connaissances, permettent de décrire les connaissances par association de faits. Plus pragmatique que le formalisme logique, le formalisme des règles permet généralement de pondérer les connaissances en fonction de coefficients numériques introduits dans les règles pour exprimer le degré de croyance, ou confiance, à accorder aux informations. Les règles sont reconnues pour favoriser l'expression du savoir-faire d'un "expert", ou spécialiste du domaine.

L'un des reproches majeurs des représentations dont le formalisme est essentiellement axé sur la logique, ou les règles, réside dans leur pouvoir très faible de structuration des connaissances. Ces formalismes représentent les connaissances de façon uniforme et rendent plus difficile la modélisation d'applications nécessitant de nombreuses connaissances. En effet, la structure des connaissances constitue un support important qui permet de guider un processus de modélisation et d'acquisition des connaissances [Stee90] mais aussi, en retour, d'explication du raisonnement [Swar83]. Il est donc important que la représentation des connaissances rendent compte de cette structure.

Aussi, le degré de structuration permis par une représentation est un facteur important pour un concepteur d'une base de connaissances puisque le formalisme employé doit rendre compte des différents niveaux conceptuels mis en évidence par sa modélisation (cf. Figure I.2).

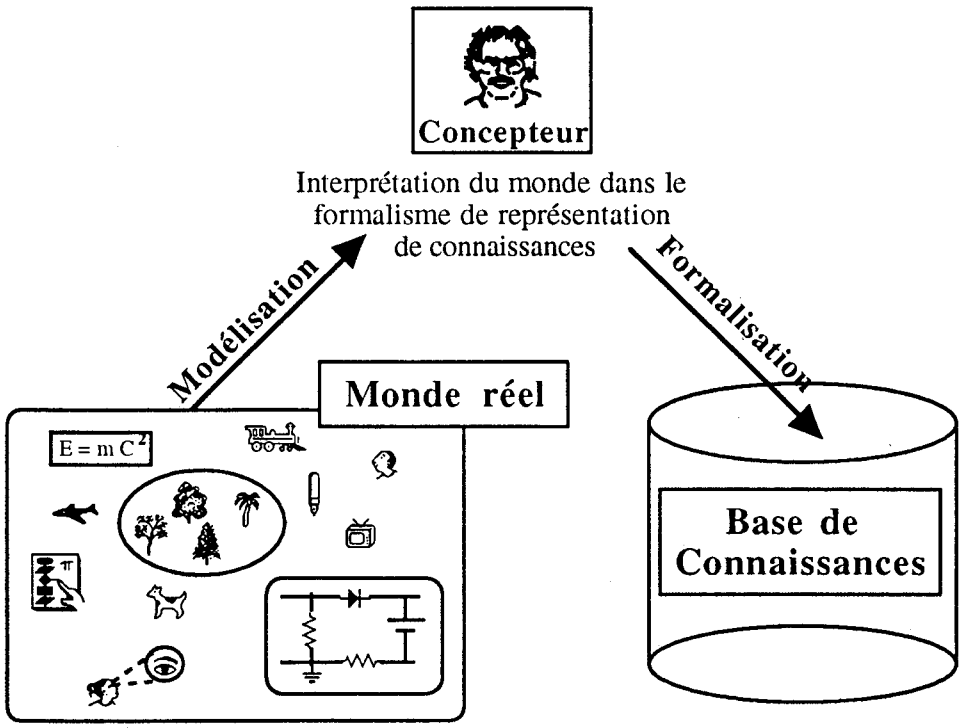

Figure I.2 : Le concepteur d'une application doit modéliser et interpréter les connaissances du monde réel dans le formalisme de représentation de la connaissances.

A partir d'une représentation de connaissances particulière, le rôle du concepteur est donc de choisir pour chaque connaissance du monde réel à modéliser, le ou les concepts qui le caractérisent au mieux dans le formalisme utilisé. Aussi, l'adéquation d'un formalisme de représentation des connaissances dépend de sa capacité à exprimer les différents niveaux et caractères de connaissances que le concepteur désire.

La structuration des connaissances est, de ce fait, devenue l'une des voies de recherche principales de l'Intelligence Artificielle. Dès la fin des années soixante, des travaux sur la compréhension des langues naturelles ont mis en avant la nécessité de structurer les connaissances et proposent comme modèle les *réseaux sémantiques* [Quil68] (cf. §I.3). Le principe est de représenter les connaissances en termes d'entités (*nœuds*) et d'associations entre ces entités (*arcs orientés*, ou *liens*, étiquetés). L'intérêt de cette structuration est double puisqu'elle permet de décrire les connaissances indépendamment de leur utilisation et, de plus, de les factoriser et donc de réduire la quantité et la recherche d'informations. Les entités représentent les objets sur lesquels le système doit raisonner et les associations représentent des relations binaires auxquelles est associée une sémantique particulière.

La relation **sorte-de** joue un rôle prépondérant puisqu'elle permet de "spécialiser" les objets et donc de décrire le **partage** d'informations entre objets. C'est-à-dire qu'un objet qui est relié à un autre par la relation sorte-de a accès aux informations de ce dernier comme si elles étaient siennes.

Au milieu des années soixante-dix, deux notions viennent enrichir les idées développées par les réseaux sémantiques, les *frames* [Mins75] et les *scripts* [Scha&77]. La notion de *frame* vient remplacer la notion de *nœud* des *réseaux sémantiques* en lui donnant un rôle de "centralisateur" d'informations relatives à un objet. Un *frame* est une unité de connaissances structurée représentant une situation stéréotypée. Plusieurs types d'informations sont attachées à un *frame* : des attributs auxquels sont associées des valeurs, des contraintes sur la valeur des attributs, des relations avec d'autres frames, des informations procédurales... Quant aux *scripts*, ils s'avèrent être un type de *frame* spécialisés dans la représentation de séquences d'actions.

Le foisonnement d'idées qu'introduisent les *réseaux sémantiques*, les *frames* et les *scripts* a donné lieu à de nombreux systèmes de représentation des connaissances dits alors "à ou par objets" [Masi&89] (cf. §I.3 et §II). Les principes fondamentaux communs à ces systèmes sont l'utilisation d'une unité de connaissances structurée, l'objet, et le partage d'informations, principe d'*héritage* ou de *prototypage*.

L'approche objet s'est répandue depuis pour être adoptée dans la plupart des systèmes. La notion d'objet peut venir compléter les systèmes à base de règles comme dans les systèmes ART [Clay86], KEE [Inte86], etc. Elle fournit dans ce cas l'unité de description des connaissances tandis que les règles en expriment l'exploitation.

1.3. Fonctionnement d'un système à base de connaissances

Un système à base de connaissances se caractérise par la dynamique des connaissances qu'il gère. Cette dynamique provient aussi bien de la production de nouvelles connaissances résultant du raisonnement mis en œuvre pour satisfaire la requête d'un utilisateur, que de l'intervention d'un concepteur qui désire étendre la base de connaissances. Le système constitue un environnement dans lequel viennent se confronter les connaissances de différents acteurs : utilisateurs et concepteurs. Chacun de ces acteurs exprime ses besoins d'exploitation des connaissances en interagissant avec le moteur d'inférence.

Le moteur d'inférence est chargé de contrôler le raisonnement. Lorsqu'un utilisateur (ou concepteur) a exprimé son but, le moteur se charge d'exploiter les connaissances pour y répondre. Le type de raisonnement (déductif, abductif, "par défaut", approximatif, monotone, etc.) qu'il met en œuvre est dépendant du but recherché, de la nature des connaissances manipulées et, plus généralement, du formalisme dans lequel elles sont exprimées.

Dans un premier temps (cf. §I.1.3.1), le fonctionnement d'un système à base de connaissances est décrit du point de vue de son utilisation afin d'en spécifier les fonctionnalités attendues. Dans un second temps (cf. §I.1.3.2), le fonctionnement interne est abordé en soulignant différents types de raisonnement.

1.3.1. Interaction avec un système à base de connaissances

Le système à base de connaissances est le centre d'intérêt de divers acteurs qui peuvent être divisés grossièrement en deux catégories : concepteurs ou utilisateurs (finaux) (cf. Figure I.3). Le but des acteurs de la première catégorie est d'intégrer de nouvelles connaissances et fonctionnalités spécifiques au domaine traité. Les utilisateurs, quant à eux, ont pour objectif l'application elle-même ; c'est-à-dire l'exploitation et l'accès aux connaissances.

Un concepteur interagit avec le système pour ajouter, modifier, valider des connaissances. Pour ce faire, il est important qu'il puisse accéder à tous les niveaux de connaissances et que le système lui fournisse des mécanismes de navigation "intelligents" ou encore "d'introspection" [Maes86] [Pitr90] ; c'est-à-dire des mécanismes qui raisonnent sur la façon dont sont représentées les connaissances. Outre l'accès aux connaissances de la base, le concepteur attend du système qu'il soit capable de rendre compte de l'état de cohérence de la base suite aux modifications ou ajouts effectués.

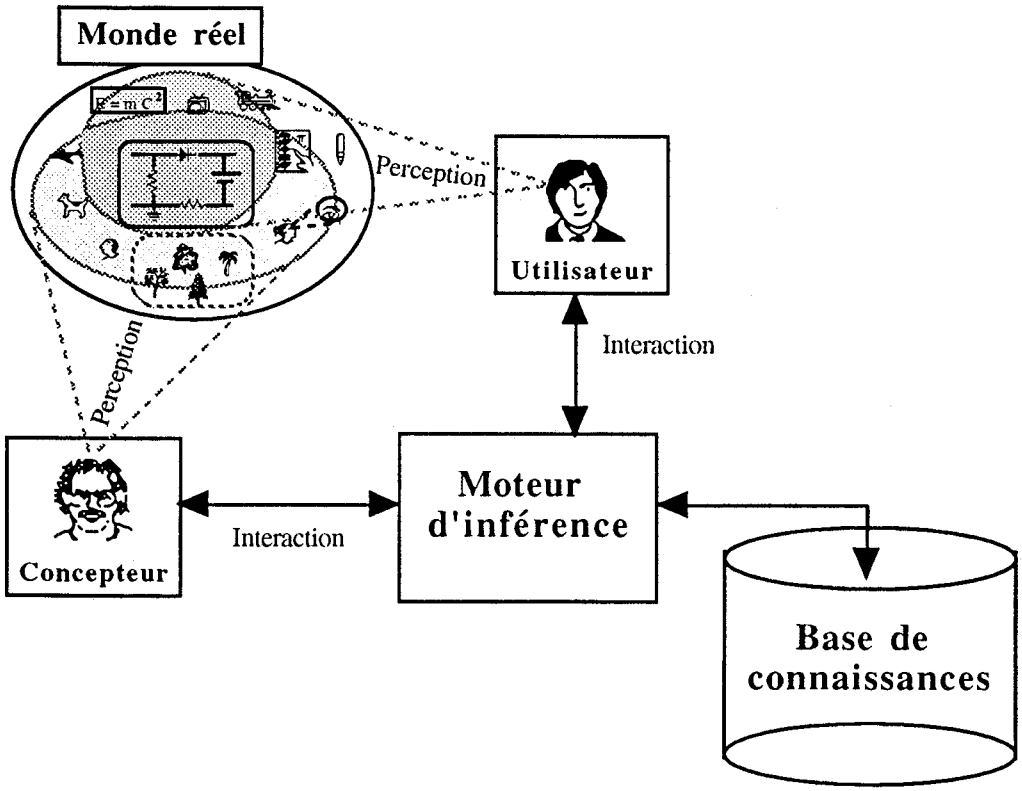

Figure I.3 : Fonctionnement global d'un système à base de connaissances montrant les différents types d'acteurs qui interagissent avec lui. Ces acteurs peuvent avoir des perceptions du monde et des besoins différents. Ils peuvent avoir un rôle de concepteur ou d'utilisateur selon qu'ils accèdent aux connaissances pour les modifier ou pour raisonner avec. Le système doit assurer son rôle de centralisateur de connaissances en offrant des facilités d'accès aux connaissances, des capacités de raisonnement et de vérification des connaissances.

Les besoins de l'utilisateur sont applicatifs, il doit pouvoir interagir avec le système pour entrer les données d'un problème, résoudre son problème, bénéficier d'aide si nécessaire, ou tout simplement naviguer à titre informatif dans la base de connaissances. Il fait donc appel à la capacité de raisonnement et aux possibilité navigationnelles [Griv&92] du système. De plus, l'utilisateur n'a pas nécessairement la même perception du monde que le concepteur qui a modélisé et introduit les connaissances dans le système ; le système doit être capable de tolérer des situations de spécification (données d'entrée d'un raisonnement) incomplète de problème, ou erronée. Dans tous les cas, le système doit rendre compte de la situation et, quand il le peut, y remédier ou proposer son aide.

Bien que ces deux catégories d'acteurs aient des centres d'intérêt différents, elles réclament du système à base de connaissances la réalisation des mêmes tâches sur la base de connaissances [Masi&89] :

- la recherche d'informations ;
- l'acquisition de nouvelles connaissances et, par conséquent, la vérification de la cohérence de la base ;
- le raisonnement qui repose sur l'exploitation des connaissances par le moteur d'inférence.

Ce qui distingue vraiment ces deux catégories ce sont les types de connaissances sur lesquels ils travaillent. Un concepteur est un utilisateur particulier dont l'objectif est de construire, étendre ou modifier la base de connaissances. Les connaissances qu'il manipule sont celles concernant la description des connaissances nécessaires à l'application. De plus, les rôles de concepteur ou d'utilisateur n'étant pas toujours si clairement définis, un utilisateur peut devenir un concepteur lorsqu'il est capable d'ajouter de nouvelles connaissances, par exemple, pour remédier à un manque de connaissances que le système a lui-même mis en évidence.

Enfin, le nombre d'utilisateurs-concepteurs n'étant pas limité, le système fait face à un ensemble d'acteurs possédant chacun sa propre perception du monde. Les différentes perceptions peuvent être complémentaires et correspondre à des approches différentes du domaine, il peut alors être intéressant de gérer la base de connaissances selon ces divers points de vue [Mari&90]. Cependant, les différentes perceptions peuvent correspondre à des prises de position sur la description du monde et peuvent être contradictoires entre elles. C'est souvent le cas dans une communauté de chercheurs où les connaissances ne sont pas consensuelles. Dans ce cas, un système à base de connaissances tient le rôle d'arbitre entre ces différents acteurs en leur permettant de centraliser les connaissances reconnues par tous (ou une majorité) et de confronter de nouvelles connaissances à la base et à l'appréciation de la communauté. Ce genre de système, appelé système à base de connaissances consensuelle [Rech93] [Taya93] [Lema93], constitue un domaine de recherche prometteur.

1.3.2. Raisonnement dans un système à base de connaissances

Le système à base de connaissances opère comme une machine abstraite dans laquelle l'unité de traitement correspond au moteur d'inférence et la mémoire est la base de connaissances. Le moteur d'inférence interprète les connaissances de la base pour développer un raisonnement. Chaque opération du moteur est appelée une inférence et a pour conséquence la production de nouvelles connaissances. Le raisonnement est donc guidé par les connaissances et caractérisé par une succession d'inférences.

On distingue différentes familles de raisonnement dans les systèmes à base de connaissances qui varient suivant la nature particulière des connaissances manipulées (incomplètes, incertaines, temporelles...), le type de contrôle, et bien sûr les possibilités d'expression du formalisme de représentation employé (typicalité, introduction de connaissances procédurales, etc.).

Deux écoles s'opposent sur la façon de *simuler* un raisonnement dans un système à base de connaissances :

> Le rôle de la logique dans le problème de la représentation des connaissances [...] est le thème d'un débat ouvert depuis la naissance de l'IA. Ce débat oppose deux de ses fondateurs, J. McCarthy et M. Minsky. Les partisans de J. McCarthy pensent que les langages de la logique mathématique peuvent constituer le support du raisonnement en vue de sa mécanisation même si la notion de démonstration ne reflète pas fidèlement la façon dont l'être humain pense. Ceux de M. Minsky pensent le contraire. Pour eux la simulation du raisonnement passe nécessairement par l'imitation de la façon dont l'esprit humain fonctionne, ce qui ne peut sûrement pas être obtenu en s'appuyant exclusivement sur les bases de la logique mathématique. [Hato&91]

En fait, ce débat traduit deux approches complémentaires : la première attachée à la logique mathématique est formelle alors que la seconde est pragmatique. La première fournit un

cadre strict bien connu qui propose des types de raisonnement généraux alors que la seconde propose des mécanismes de raisonnement spécialisés qui tirent parti des spécificités de modèles de représentation complexes.

L'approche pragmatique se caractérise par sa créativité et constitue pour l'approche formelle une source d'étude considérable. Notamment, la nécessité de manipuler des connaissances imprécises, incomplètes, incertaines, temporelles... constitue un résultat de l'approche pragmatique et fournit à l'approche formelle de nouvelles problématiques qui remettent en cause l'utilisation de la logique classique seule. Inversement, l'approche pragmatique peut bénéficier du cadre fournit par l'approche formelle pour analyser, interpréter et expliquer ses résultats. L'approche pragmatique propose, l'approche formelle dispose !

L'approche formelle met en évidence trois grandes familles de raisonnement [Hato&91] :

- **déductif**, qui permet de dériver les conséquences logiques d'un ensemble d'énoncés ou faits connus. Par exemple, "la terre tourne" peut être déduit à partir des connaissances "la terre est ronde" et "ce qui est rond tourne". Plusieurs techniques pour mettre en œuvre le raisonnement déductif sont proposées :
  - *dirigé par les buts* : le raisonnement décompose le but recherché en sous-buts qui permettraient de l'atteindre par déduction direct, et ainsi de suite jusqu'à atteindre des données connues. Le raisonnement est alors réduit à la production des connaissances "intéressantes" pour atteindre le but.
  - *dirigé par les données* : le raisonnement part des données initiales pour déduire de nouvelles connaissances, et réitère ce processus jusqu'à l'obtention du but.
  - *par résolution ou réfutation* : le raisonnement opère par l'absurde, il cherche à montrer qu'un énoncé est vrai en prouvant que sa négation produit une contradiction avec les énoncés et faits connus. Quand l'énoncé but comporte des variables, un mécanisme d'unification permet de préciser pour quelles valeurs de ces variables l'énoncé est vrai. Cette technique est employée par le langage de programmation logique PROLOG.
- **inductif**, permet de généraliser des connaissances à partir d'un ensemble de faits. Par exemple, l'énoncé "ce qui est rond tourne" peut être proposé par induction au vue des faits "la terre est ronde" et "la terre tourne".
- **abductif**, cherche à expliquer un fait en proposant des causes possibles qui le justifient. Par exemple, "la terre est ronde" est une cause expliquant que "la terre tourne", sachant que "ce qui est rond tourne".

Le raisonnement déductif est présent dans la plupart des systèmes d'I.A. mais son rôle est restreint : expliciter les connaissances contenues implicitement dans une base de connaissances. De plus, les connaissances manipulées en général ne peuvent se suffire des notions de validité ou non validité qui constituent les bases du raisonnement logique. Plusieurs travaux étendent le raisonnement logique à de nouvelles logiques permettant de prendre en compte différents aspects des connaissances : la logique multivaluée ou floue pour l'aspect incertain, logique des défauts pour l'aspect révisable, etc. On trouvera dans [Somb88] et [Hato&91] une revue des différentes logiques et techniques proposées pour répondre en partie à ces problèmes.

Un aspect important du raisonnement est sa monotonie ou non monotonie. Un raisonnement est **monotone** lorsqu'il ne remet jamais en cause les connaissances existantes (initiales ou produites) dans la base. En revanche, il est dit **non monotone** lorsqu'il a la capacité de retirer des connaissances. Le terme de monotonie est bien sûr en rapport avec la croissance de l'ensemble des déductions. La non monotonie traduit la gestion de la croyance, une connaissance est crue vraie puis s'avère fausse. Ce genre de changement de croyance remet en cause la partie du raisonnement qui s'appuyait dessus. Pour gérer cette remise en cause, un système à base de connaissances peut inclure un **système de maintien du raisonnement**. Le chapitre III donne une description des deux approches classiques de système de maintien de raisonnement. Ces deux types de système permettent l'exploitation de connaissances révisables mais se distinguent par le type de raisonnement qu'ils supportent : le premier (historiquement)

maintient un raisonnement non monotone, le second gère différents contextes dans chacun desquels le raisonnement est monotone.

L'approche pragmatique s'appuie sur l'étude d'applications [Clan85] [Chan86] ou de travaux dans d'autres domaines comme la psychologie [Mins75]. Les premiers s'attachent à faire ressortir la relation entre le type de problème et le type de raisonnement afin d'en dégager des structures ou enchaînements d'inférences précis. Les seconds se préoccupent en priorité de la dualité raisonnement/représentation.

Ces deux approches pragmatiques se traduisent en général par la spécification de langages de représentation des connaissances qui reposent sur des modèles conceptuels complexes (niveaux élevés de structuration) et la mise en place de mécanismes de raisonnement spécialisés capables d'exploiter efficacement et dans un but précis les connaissances structurées. Le moteur d'inférence s'enrichit en conséquence par l'introduction de ces mécanismes spécialisés. Par exemple, les langages de représentation par objets introduisent le mécanisme d'héritage et intègrent, pour la plupart, la notion de "réflexe" ou "attachement procédural" qui permet l'utilisation ponctuelle de connaissances procédurales. D'autres mettent en place des mécanismes complexes comme la classification qui permet de placer un objet dans une hiérarchie d'objets.

Les systèmes résultant de l'approche pragmatique combinent différents types de raisonnement décrits par l'approche formelle au sein même des mécanismes d'inférences complexes et spécifiques qu'ils proposent.

Dans la suite de ce chapitre, les principes généraux de différents types de systèmes sont présentés pour illustrer cette rapide présentation des systèmes à base de connaissances issus des travaux en intelligence artificielle. Les systèmes à base de règles sont abordés en premier car, en tant que descendants des premiers systèmes axés sur la logique, ils offrent des mécanismes d'inférence simples qui montrent clairement le cycle d'un moteur d'inférence. Dans un second temps, les systèmes qui préconisent une représentation des connaissances structurée et dans lesquelles apparaissent explicitement, ou implicitement, les notions d'objet et de hiérarchie d'objets donnent un aperçu de l'impact de la représentation sur le raisonnement.

2. Système à base de règles

A l'image des premiers systèmes basés sur la logique, les systèmes à base de règles s'articulent autour d'un mécanisme général simple. Celui-ci exploite une base de connaissances comportant deux types d'entités de description : les faits et les règles. Le développement et l'amélioration de ce type de système permettent, néanmoins, un pilotage du raisonnement plus élaboré, notamment par l'introduction de méta-règles qui peut jouer sur le contrôle du moteur ou par la pondération des données avec des coefficients de vraisemblance pour traiter des cas de raisonnement incertain.

2.1. Représentation de la base de connaissances

La base de connaissances se divise en base de faits et base de règles (cf. Figure I.4).

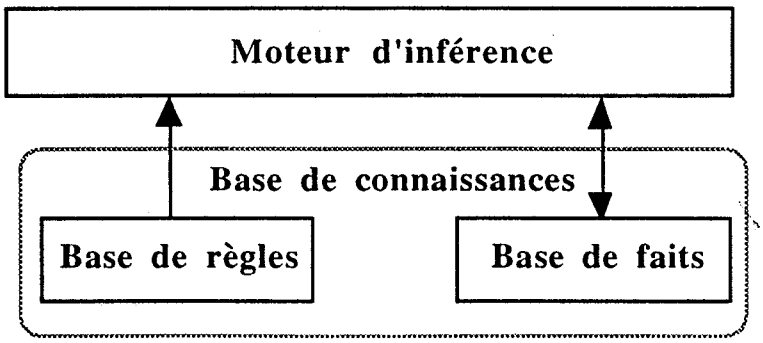

Figure I.4 : Architecture d'un système à base de règles dans laquelle la base de connaissances se divise en une base de règles et une base de faits.

La base de faits constitue la mémoire de travail dans laquelle sont inscrites les données initiales d'un problème particulier et celles qui sont produites au fur et à mesure du développement d'un raisonnement. La base de règles contient l'ensemble des règles qui définissent l'application du système.

Une règle, appelée généralement **règle de production**, formalise l'association entre un ensemble de conditions à vérifier et une action à entreprendre ; elle se présente sous la forme générale :

SI *conditions* **ALORS** *action*

Les conditions, appelées aussi prémisses, sont données par une formule logique. Si des faits de la base de faits valident cette formule, la règle est alors dite **activable**, ou **applicable**. L'action d'une règle décrit des modifications de la base de faits, et permet aussi d'autres opérations comme la demande d'informations auprès de l'utilisateur. Un exemple de règle du système de diagnostic en maladies infectieuses MYCIN [Shor76] est donné ci-dessous :

PREMISE		**SI**
(**$AND**	(**SAME** *organism coloration gram-positive*)	; la coloration de l'organisme ; est gram-positive, et
	(**SAME** *organism morphologie coque*)	; la morphologie de ; l'organisme est coque, et
	(**SAME** *organism conf-croiss agglutination*)	; la conformation de ; croissance de l'organisme ; est agglutination
		ALORS
ACTION		
	(**CONCLUDE** *organism ident staphylocoque*)	; l'identité de l'organisme est ; staphylocoque.

Cette règle permet d'identifier un organisme comme étant un staphylocoque au vu de sa coloration, de sa morphologie et de la conformation de sa croissance. Elle est activable si la base de faits contient les propositions suivantes :

- *(coloration org-1 gram-positive)*,
- *(morphologie org-1 coque)*,
- *(conf-croiss org-1 agglutination)*,

où *org-1* constitue l'exemplaire d'organisme que le raisonnement est en train d'étudier.

Les bases de règles et de faits sont souvent structurées en "paquets" permettant de décrire une décomposition de problème en sous-problèmes. Le système développe alors son raisonnement en considérant successivement différents paquets. Dans ce cas, une action de règle peut décrire le passage d'un paquet de règles à un autre.

2.2. Fonctionnement du moteur d'inférence

Partant de la base de règles et d'une configuration initiale de la base de faits représentant un problème à résoudre (par exemple, les faits décrivant les symptômes d'un malade dans MYCIN), le moteur d'inférence permet d'enchaîner les règles en fonction de leur applicabilité et d'en exécuter les actions (cf. Figure I.5).

Deux modes de fonctionnement peuvent être attribués à un moteur d'inférence d'un système à base de règles :

- en **chaînage avant** : le raisonnement est dirigé par les faits connus, c'est-à-dire qu'il déclenche les règles qui sont activables à partir de cet ensemble de faits dont la validité est reconnue, puis recommence à partir de la base de faits résultante, et poursuit ainsi de suite jusqu'à ce qu'il ne soit plus possible d'activer la moindre règle. La base des faits est alors dite *saturée* car tous les faits déductibles à partir de la base de règles et des faits initiaux ont été dérivés.

- en **chaînage arrière** : le raisonnement est dirigé par le but à atteindre : pour vérifier la validité d'un ensemble de faits (le but), la sélection des règles se fait sur leur partie conclusion, c'est-à-dire que sont sélectionnées les règles qui permettent d'engendrer ces faits. Les prémisses de ces règles deviennent les nouveaux buts à atteindre. Le raisonnement s'arrête lorsque les faits à vérifier appartiennent déjà à la base de faits.

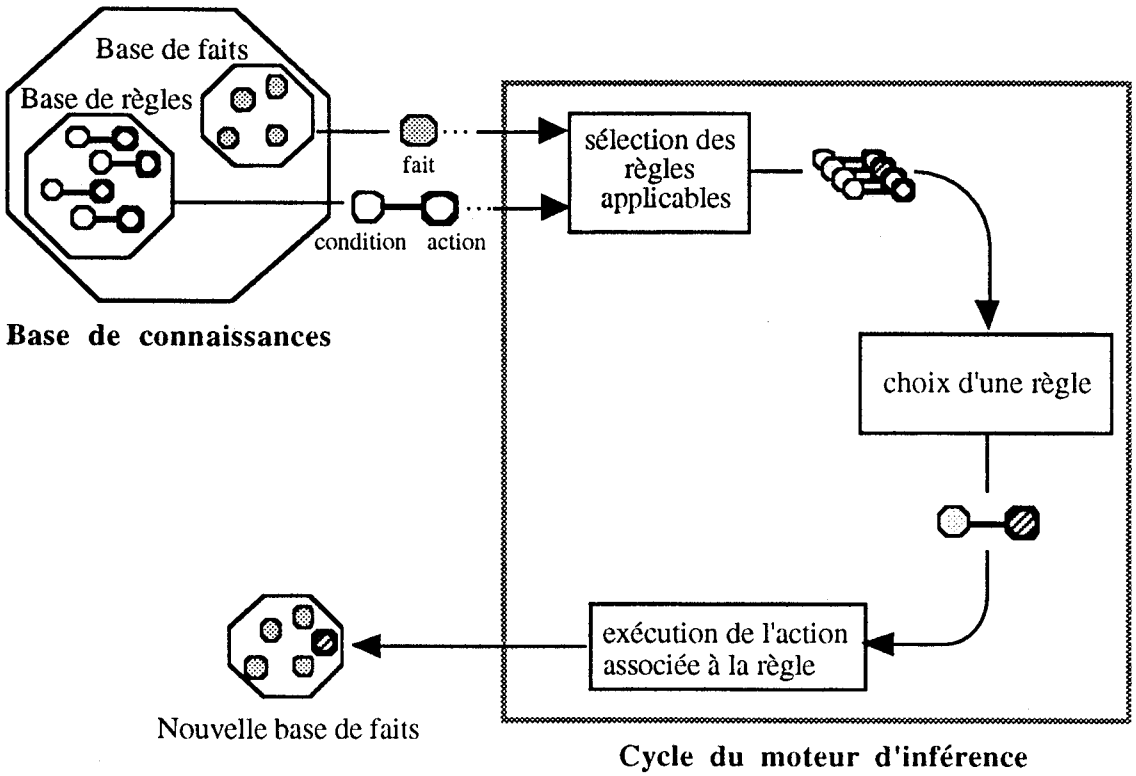

Figure I.5: Schéma décrivant le cycle du moteur d'inférence (en chaînage avant) [Mari93].

Un **chaînage mixte** peut être mis en place permettant d'alterner dans un raisonnement des phases en chaînage avant et en chaînage arrière. L'idée est de réunir les avantages provenant des deux types de chaînage. En effet, ils ont un rôle complémentaire : le chaînage arrière permet de décrire une décomposition de problème en sous-problèmes, alors que le chaînage avant permet la production de nouveaux faits. Pour mettre en place un chaînage mixte, le système doit distinguer dans la base de faits ceux qui sont connus de ceux qui représentent les buts à atteindre.

Enfin, les systèmes qui autorisent l'introduction de variables dans l'écriture des règles enrichissent le mécanisme de sélection de règles d'un mécanisme d'appariement. Dans l'exemple de règle donné dans la partie précédente, *organism* est une variable. Lors de l'application de la règle aux faits qui accompagnent cet exemple, les conditions de la règle sont appariées avec chaque fait en substituant à la variable *organism* la valeur *org-1*. L'introduction de variables rend la phase de filtrage particulièrement coûteuse car elle doit envisager les diverses substitutions que lui permet la base de faits.

2.3. Evolutions majeures du formalisme de règle

Outre l'étude des différentes stratégies de contrôle d'enchaînement de règles, les nombreux travaux sur ce type de système ont abouti à des résultats importants. Parmi ceux-ci, il convient de souligner l'utilisation de métaconnaissances pour décrire la stratégie de raisonnement, l'exploitation de coefficients de vraisemblance qui permet la mise en place d'un

raisonnement approximatif et enfin la compilation de règles qui cherche à augmenter l'efficacité des systèmes à base de règles.

2.3.1. La métaconnaissance pour structurer le raisonnement

On parle de métaconnaissance lorsqu'un système inclut des connaissances relatives à la façon de représenter ou d'utiliser ses propres connaissances. Introduite par R. Davis [Davi80] pour les besoins du système TEIRESIAS qui étend MYCIN, la notion de méta-règle correspond à l'expression de la métaconnaissance d'un système à base de règles.

Le rôle d'une méta-règle est d'exprimer une connaissance sur le contrôle des règles. Le raisonnement mené à partir des méta-règles considère les règles comme des faits et permet de définir une stratégie de contrôle adaptée à des situations précises de l'application. Pour ce faire, une méta-règle adopte la même syntaxe qu'une règle normale. Cependant, les conditions de la méta-règle permettent d'exprimer des sélections particulières de règles en fonction de leur contenu. La partie action d'une méta-règle permet d'inhiber, d'activer ou d'ordonnancer certaines règles. Enfin, le moteur d'inférence exploite les méta-règles en chaînage avant.

L'exemple ci-dessous présente deux interprétations de méta-règles de MYCIN. Les deux montrent comment des conditions peuvent exprimer des sélections de règles. La partie action de la première exprime un ordre de déclenchement de deux groupes de règles tandis qu'elle a pour but d'inhiber une sélection de règles dans la deuxième :

Méta-règle1 :

SI

le patient est un hôte à risque,
et il existe des **règles** qui mentionnent des pseudomonias dans une de leurs **prémisses**,
et il existe des **règles** qui mentionnent des klebsiellas dans une de leurs **prémisses**,

ALORS

il est probable (0.4) qu'il faille utiliser les premières **règles** avant les secondes.

Méta-règle2 :

SI

le site de la culture est non stérile,
et il existe des **règles** qui mentionnent dans une de leurs **prémisses** un organisme déjà rencontré auparavant chez le patient et qui peut être le même que celui dont on recherche l'identité,

ALORS

il est sûr (1.0) qu'aucune d'entre elles (les **règles**) ne peut servir.

Les méta-règles constituent un pas supplémentaire vers une représentation de la connaissances structurée. Elles permettent d'expliciter les connaissances stratégiques d'une application particulière, et par opposition aux solutions procédurales (codage dans le moteur d'inférence) d'introduire dynamiquement des heuristiques de contrôle.

2.3.2. L'introduction de coefficients pour raisonner avec l'incertain

Des coefficients de vraisemblance peuvent être attachés aux connaissances de la base et permettent d'indiquer la confiance qu'accorde un spécialiste du domaine à ces connaissances. Comme dans les exemples des méta-règles de la partie précédente, ces coefficients apparaissent dans la conclusion des règles et permettent ainsi d'affecter dynamiquement un coefficient de vraisemblance à un fait. Ainsi, les faits présents dans la base sont tous pondérés par un coefficient.

Ce coefficient est une valeur numérique comprise entre -1 (faux) et 1 (certain) qui traduit les niveaux de vraisemblance attribuables à un fait. Différentes formules dépendantes du système permettent le calcul du coefficient d'un fait en fonction du coefficient que lui accorde la règle qui le déduit et les coefficients des faits qui ont permis de la déclencher, les prémisses.

L'intérêt principal de l'introduction de tels coefficients dans les systèmes à base de règles est de pouvoir raisonner sur des connaissances incertaines et pouvoir ainsi explorer différentes solutions possibles pour un même problème. Chacune de ces solutions est alors fournie avec le coefficient de vraisemblance que le système a calculé. L'exemple des faits présentés ci-dessous montre que le système considère deux développements possibles de raisonnement : l'un suppose que l'organisme *org-1* est *E. Coli* et l'autre suppose que ce même organisme est *Klebsiellia.*

Exemple de coefficients de vraisemblance sur une règle et des faits de MYCIN [Laur87] :

Règle :

SI

le *site* de la *culture* est le *sang*,
et l'*organisme* est à *Gram négatif*,
et l'*organisme* est de *forme bâtonnet*,
et le *patient* est un *hôte à risque*,

ALORS

il est **probable (0.6)** que l'*organisme* est le *pseudomonia aeruginosa.*

Extrait de la base de faits :

(*Identité org-1 :*	*E. Coli*	**0.7**)
(*Identité org-1 :*	*Klebsiella*	**0.4**)
(*Sensibilité org-1 :*	*Pénicilline*	**- 0.9**)

Néanmoins, cette approche du raisonnement incertain, ou approximatif, est critiquée pour deux raisons principales. Premièrement, il est ardu de donner des coefficients de vraisemblance aux règles et aux faits quand ils sont nombreux. La mise au point d'un coefficient de vraisemblance est dépendante de ceux déjà établis, elle devient donc difficile et nécessite une vision globale de la base de connaissances. Deuxièmement, les combinaisons de coefficients que doit effectuer le moteur se basent sur des formules de calculs qui sont purement empiriques et difficiles à choisir.

2.3.3. La compilation de règles

L'étape de filtrage des règles est l'opération la plus coûteuse en temps, elle est proportionnelle au nombre de règles, de faits et de conditions par règles. Dans certains cas, cette opération peut occuper 90% du temps du moteur d'inférence.

La technique, dite de *compilation* d'une base de règles, se propose d'atténuer ce problème en transformant la base de règles en une structure permettant de mettre en évidence les parties de conditions communes entre règles. L'intérêt de cette opération est donc double : réduire la taille mémoire nécessaire au fonctionnement du système et augmenter l'efficacité de la phase de sélection de règles activables.

Cette technique s'appuie sur la construction d'un arbre d'unification des différentes conditions rencontrées dans les règles et d'un réseau de jointure décrivant l'association entre conditions et règles [Ghal88]. La première structure permet de réduire la phase d'unification entre faits et conditions des règles, tandis que la deuxième sur la base des conditions reconnues valides permet de réduire la sélection des règles activables. L'algorithme RETE [Forg82] est le premier algorithme de compilation de règles à avoir été proposé sur ce principe.

2.4. Inconvénients des systèmes à base de règles

Le fonctionnement de ce type de système, dans sa forme la plus simple, s'articule autour d'une seule unité de connaissances : la règle de production. Si on accorde généralement à ce genre de formalisme les avantages de la modularité et de la simplicité, on peut lui reprocher en contre-partie sa trop grande uniformité. En effet, le formalisme des règles, en tant que tel, souffre d'un manque de structuration des connaissances et s'avère inadéquat à rendre compte des objets manipulés et des diverses relations qui peuvent les lier.

Dans une application réelle, l'univers à modéliser est généralement constitué d'objets complexes qui possèdent un ensemble de propriétés et entretiennent entre eux certaines relations. La description de cet univers dans une base de règles ne pouvant être faite explicitement, les propriétés et relations des objets sont dispersées dans les règles.

La perte du concept d'objet est pénalisante pour le système dont la phase de sélection a trop souvent pour sous-but le regroupement dynamique d'informations relatives à un ou plusieurs objets. En effet, les différentes phases par lesquelles passent un raisonnement se caractérisent par les types d'objets qu'elles exploitent. Les objets constituent donc des centres d'intérêts temporaires pour le raisonnement. La notion d'objet s'avère être une connaissance structurante de haut niveau pour un raisonnement. Son intégration dans le formalisme de règles peut réduire considérablement l'étape de sélection des règles activables. C'est pourquoi le formalisme des règles est souvent complété par une représentation par objets permettant une description structurée de l'univers manipulé.

Plus généralement, la notion de règle formalise une connaissance de "savoir-faire" d'un spécialiste et rend les connaissances ainsi codées dépendantes de l'application. Cette dépendance rend difficile la réutilisation des connaissances à d'autres fins. Différents travaux [Clan83b] [Swar83] [Chan86] à travers l'analyse de systèmes à base de règles soulignent les aspects des connaissances que ce formalisme occulte par sa représentation et les problèmes que cela pose aussi bien lors de l'acquisition, de l'exploitation que de l'explication des connaissances. Les connaissances représentées sous forme de règle sont souvent qualifiées de "connaissances de surface" par opposition aux "connaissances profondes" [Stee90] [Horn91] du domaine d'application. Les "connaissances profondes" sont fortement structurées et indépendantes de toute utilisation. Les méta-règles constituent une première réponse à ce problème.

Parallèlement à ces travaux d'analyse, d'autres travaux ont commencé à apporter des éléments de réponse en proposant des systèmes de représentation de connaissances qui mettent l'accent sur la structuration. Ces systèmes, qui ont réduit considérablement le rôle accordé au formalisme de règle et introduit progressivement le concept actuel d'objet, sont présentés par la suite.

3. Vers un monde d'objets et de hiérarchies

Conséquence directe du constat qu'il est nécessaire et profitable de structurer les connaissances manipulées par un système d'I.A., la représentation des connaissances s'est imposée comme un domaine de recherche à part entière. L'apparition en I.A. des notions d'objet et de hiérarchie est le fruit de nombreux travaux parmi lesquels les réseaux sémantiques et les frames font figure de modèles de référence. Les langages de programmation par objets ont aussi constitué une source d'inspiration importante.

Cette partie se propose, dans un premier temps (cf. §I.3.1), de présenter les principes généraux des représentations par objets qui ont été dégagés par ces deux modèles de référence. Dans un second temps (cf. §I.3.2), les objets sont introduits en soulignant deux approches possibles, *prototype* et *ensemble*.

Cette distinction prend racine dans les origines des représentations par objets. Aussi, il peut être difficile de situer avec certitude la catégorie d'un langage de représentation actuel tant les deux approches partagent des notions similaires. Il n'est d'ailleurs pas impossible qu'un langage puisse se réclamer des deux approches à la fois en fonction des caractéristiques que l'on mettra en avant pour le présenter.

Etant donné la variété de travaux dans chacune des approches, seules leurs grandes lignes sont abordées dans cette partie. Des informations complémentaires peuvent toutefois être trouvées dans le chapitre IV qui reprend la distinction entre les deux approches pour montrer les différentes interprétations données à la notion de classe dans les représentations par objets.

3.1. Les origines des objets en Intelligence Artificielle

Avec l'introduction de la notion de graphe, les réseaux sémantiques marquent l'avènement de la représentation structurée des connaissances et constituent les bases de toutes les représentations par objets qui lui ont succédé.

Très rapidement l'héritage et l'appariement (ou unification de graphes) deviennent des mécanismes privilégiés pour exploiter les réseaux sémantiques. Puis, l'émergence de la notion de *frame* s'impose comme un nouveau tournant décisif de la représentation des connaissances en proposant de regrouper au sein d'une structure l'ensemble des connaissances associées à un objet du raisonnement.

3.1.1. Les réseaux sémantiques : naissance de l'héritage et de l'appariement

Conçus à l'origine pour représenter la signification des mots en anglais, les réseaux sémantiques s'inspirent de modèles psychologiques de la mémoire associative humaine [Quil68]. Leur grande originalité provient de l'utilisation de la structure de graphe pour représenter les connaissances. Cette nouvelle approche de représentation présente de nombreuses possibilités de manipulation des connaissances. En effet, elle favorise l'étude de différents types d'accès aux connaissances par la mise en place de mécanismes de parcours et d'unification de graphe.

Un réseau sémantique représente les informations sous forme d'un graphe orienté. Les nœuds du graphe permettent de fournir les entités, les abstractions, ou encore les événements relatifs à un domaine d'application, tandis que les arcs sont étiquetés par les noms des relations binaires entretenues par les concepts du domaine. Cette représentation facilite la compréhension des connaissances d'un domaine par la donnée d'une description graphique d'un réseau (cf. Figure I.6).

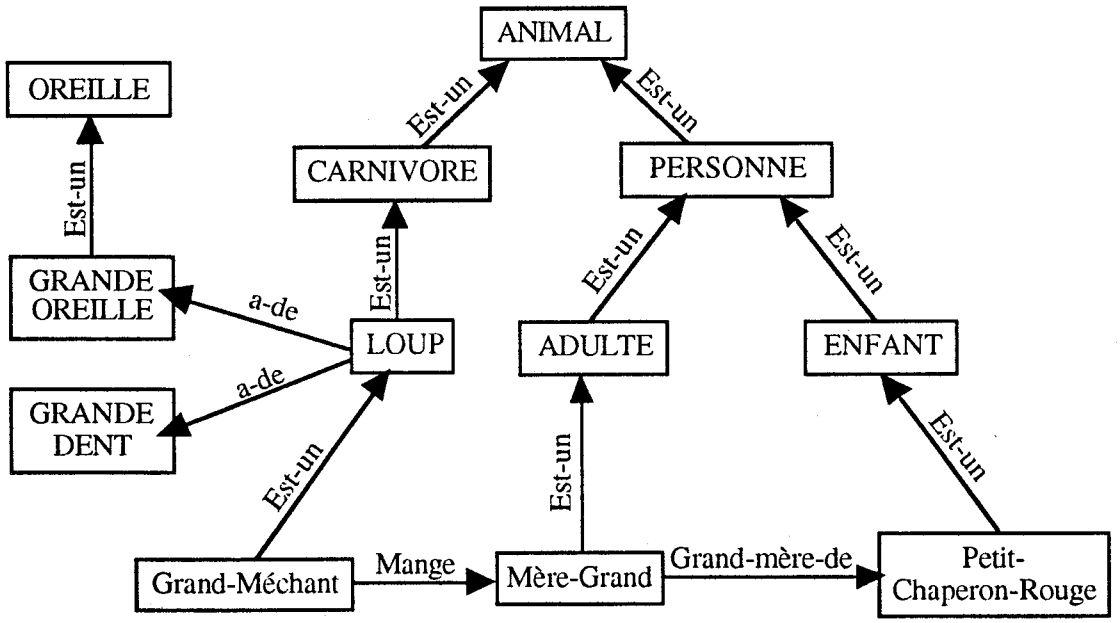

Figure I.6 : Exemple d'un réseau sémantique permettant d'exprimer, entre autre, que "le loup Grand-Méchant mange Mère-Grand qui est la grand-mère du Petit-Chaperon-Rouge".

L'originalité des réseaux sémantiques réside dans le type d'exploitation mis en place pour résoudre un problème. Le principe adopté est celui de la recherche d'information basée sur un mécanisme de parcours du réseau.

Ainsi, le premier mécanisme proposé dans [Quil68], appelé *recherche d'intersection*, consiste à trouver les relations existantes entre deux nœuds. Cette recherche permet de répondre

à des questions du genre "quelle est la relation entre le Grand-Méchant loup et le Petit-Chaperon-Rouge ?".

Les mécanismes proposés par la suite donnent une importance plus grande aux relations en leur accordant une sémantique particulière. Un mécanisme de recherche prend alors une signification spécifique en fonction des relations particulières qu'elle traite. C'est ainsi que la relation **est-un** prend une importance prépondérante dans les réseaux sémantiques et qu'elle donne naissance au mécanisme d'**héritage**. Ce dernier a pour objectif de récupérer dynamiquement à partir d'un nœud donné les informations des nœuds qui peuvent être atteints à partir des liens **est-un**. Le sous-réseau correspondant aux liens **est-un** constitue une hiérarchie, appelée **hiérarchie d'héritage**. Dans l'exemple, le parcours de ce lien permet de déduire que le Grand-Méchant loup possède de grandes oreilles et de grandes dents.

L'apparition du mécanisme de **filtrage** permet de confronter un réseau sémantique à un autre. L'idée est d'effectuer une demande d'information sur un réseau sémantique en la représentant elle-même par un réseau sémantique. On cherche alors les **appariements** possibles entre ce "réseau-question" et le réseau initial. Le "réseau-question" se caractérise par l'introduction de nœuds variables que le filtrage cherche à substituer, lors de la mise en correspondance, par des nœuds du réseau initial.

Le filtrage peut combiner **appariement** de graphe et **héritage**, ce qui lui permet de répondre à une question complexe comme pourrait en formuler le Petit-Chaperon-Rouge : "quel est l'animal avec des grandes oreilles et des grandes dents qui mange Mère-Grand ?" (cf. figure I.7).

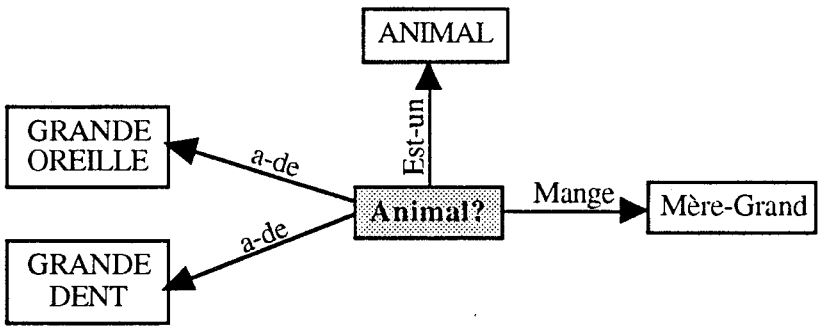

Figure I.7 : Réseau représentant la question "quel est l'animal avec des grandes oreilles et des grandes dents qui mange Mère-Grand ?". Lors de la confrontation avec le réseau de la figure I.6, le nœud variable **Animal ?** peut être substitué par le nœud "Grand-Méchant" qui est par héritage un loup et donc un animal avec des grandes oreilles et des grandes dents.

Les réseaux sémantiques s'avèrent être une étape cruciale de l'évolution des modèles de connaissances. La mise en avant de l'importance de certaines relations, ainsi que des principes de parcours et d'appariement de graphe, leur ont fait connaître un développement considérable par la suite.

Plusieurs points ont connu un intérêt particulier : la sémantique des liens, et notamment celle du lien **est-un** ("**is-a**") [Wood75] [Brac83] [Brac&85] ; le problème de la quantification des énoncés qui a donné naissance aux réseaux sémantiques partitionnés [Hend79] ; et plus généralement, la nécessité de structurer les connaissances à un niveau de granularité plus élevé ; c'est-à-dire à l'aide du concept d'objet.

3.1.2. Les *frames* de M. Minsky : naissance du concept d'objet

On ne saurait parler d'objet dans le domaine de l'I.A. sans citer le travail de M. Minsky [Mins75] qui en fut l'un des précurseurs en introduisant le concept de *frame*. Ses travaux sur la perception et vision par ordinateur l'ont amené à faire un ensemble de propositions d'ordre conceptuel sur la structuration possible des connaissances.

Inspiré à la fois par des théories de psychologie et les travaux d'I.A., M. Minsky se distingue d'emblée du courant basé sur le formalisme logique. Il part de l'idée selon laquelle l'humain possède en mémoire des structures d'informations lui permettant de représenter les situations vécues sous forme de stéréotypes. Selon ce modèle, l'efficacité d'un humain à réagir à une nouvelle situation s'explique par sa faculté à sélectionner parmi les stéréotypes qu'il possède celui qui semble correspondre au mieux à la situation courante. Si nécessaire, il peut adapter structure et données de ce stéréotype afin qu'il convienne au mieux à la nouvelle situation.

Pour ce faire, il propose la notion de *frame* comme unité conceptuelle de représentation d'une situation stéréotypée. Cette unité est une structure de données qui peut être vue comme un réseau de nœuds et de relations. Ce réseau est hiérarchisé en niveaux dans lesquels les nœuds des niveaux supérieurs représentent les "choses qui sont toujours vraies dans la situation considérée" tandis que les nœuds des niveaux inférieurs possèdent des attributs ("slots") auxquels peuvent être affectées des données spécifiques ou d'autres *frames* qualifiés dans l'article de Minsky d'un "plus petits". Chaque attribut peut spécifier les conditions que les données doivent vérifier avant de lui être affectées.

L'ensemble des *frames* est organisé en *systèmes de frames*. Un tel système représente un regroupement de *frames* en rapport avec un même sujet ; ces *frames* étant alors liés entre eux par les relations spécifiques à l'application. L'exemple classique de Minsky est celui du *système de frames* décrivant les différentes façons de percevoir un cube. Les relations comme "déplacer-à-droite" ou "déplacer-à-gauche" qui lient les différents *frames* permettent de rendre compte du passage d'une vue du cube à une autre.

Afin de comparer une situation particulière à un *frame*, M. Minsky propose un mécanisme d'appariement permettant de confronter les informations fournies par la situation à celles contenues dans le *frame*. Parmi les informations contenues dans le frame, certaines, comme les valeurs par défaut, permettent de compléter ou modifier les informations décrivant la situation observée. Un frame peut aussi contenir les informations permettant d'établir quel autre frame peut le remplacer s'il ne correspond pas suffisamment à la situation courante.

Si la notion de *frame*, telle qu'elle est présentée par M. Minsky, n'a pas connu une réalisation effective, elle a par contre inspiré de nombreux systèmes de représentation des connaissances. La variété et le nombre de propositions provenant de son article constituent une source considérable d'idées et d'interprétations possibles dans laquelle il n'est pas rare que les travaux actuels puisent encore.

Toutefois, parmi l'ensemble de propositions faites, la notion de *frame* comme structure permettant de regrouper tous types de connaissances (procédurales, structurelles, défaut, relationnelles, etc.) pour donner corps à un concept complexe du raisonnement, reste le résultat principal de ce travail conceptuel. Déjà esquissé par les réseaux sémantiques, le concept d'**objet** trouve sa substance dans les *frames* et devient ainsi une nouvelle unité de représentation des connaissances.

3.2. Les représentations par objets

Le concept d'objet en I.A. s'inspire des principes généraux dégagés dans les réseaux sémantiques et la notion de *frame* de M. Minsky, à savoir : les connaissances sont organisées en hiérarchie d'objets, et chaque objet est une structure regroupant les différentes connaissances qui le décrivent. Les informations attachées à un objet peuvent être éventuellement complétées dynamiquement par héritage à travers la hiérarchie d'objets. Bien que ces principes soient communs à tous les modèles à objets, le concept d'objet peut cependant revêtir différentes "formes" selon le choix du modèle. Le terme "forme" fait référence aussi bien à l'aspect syntaxique qu'à l'aspect sémantique d'un formalisme à base d'objets.

Le premier aspect ne s'avère intéressant en tant que tel que par les indications qu'il apporte sur les sources d'inspiration d'un formalisme. Par exemple, les langages terminologiques (cf. 3.2.3) possèdent une syntaxe d'inspiration logique qui leur vaut aussi l'appellation *logique de description*. Pour ce qui est des langages de *frames* (cf. 3.2.2), il est

clair que leur syntaxe s'inspire en grande partie des indications fournies sur la structure d'un *frame* dans l'article de M. Minsky, mais aussi des travaux menés parallèlement dans le domaine des langages de programmation à objets. Les langages terminologiques sont le fruit d'une approche formelle des langages de *frames*, tandis que les langages de frames résultent d'une approche à caractère plutôt pragmatique.

L'aspect sémantique, quant à lui, mérite une attention toute particulière puisqu'il détermine les capacités inférencielles associées à un formalisme à base d'objets. A ce titre, la signification accordée à l'organisation hiérarchique des objets et, plus particulièrement, au lien entre deux objets, constitue un élément essentiel de la définition d'un modèle objet. Ce sujet est le thème d'une controverse opposant deux approches : l'approche par prototype et l'approche par ensemble.

La prochaine partie est consacrée à cette controverse, tandis que les deux suivantes décrivent deux familles de représentations par objets qui illustrent l'influence, plus ou moins directe, de chacune des deux approches précitées. Ces deux modèles sont les langages de *frames* et les langages terminologiques.

3.2.1. Prototype et ensemble

Il existe dans l'univers de l'objet deux grands modèles d'organisation de la connaissance : le modèle des **ensembles** et le modèle des **prototypes**. Ces deux approches cherchent à atteindre des objectifs différents et se traduisent par deux évolutions particulières de la représentation par objets. Afin d'appréhender au mieux leurs objectifs respectifs par la suite, les fondements de ces deux approches sont d'abord présentés en se focalisant sur les caractéristiques qui les distinguent.

La divergence fondamentale entre ces modèles est liée au problème de la représentation d'objets conceptuellement proches, et notamment du partage de connaissance entre ces entités.

L'approche par prototype conduit à considérer un modèle à un seul niveau d'entités, les objets prototypiques. L'existence dans la base d'un tel objet est indépendante de toute abstraction, mais l'objet est susceptible d'être le prototype d'autres objets partageant une partie de sa caractérisation. Le prototypage correspond à la création d'un lien "**est-presque-le même-que**", entre deux objets. Le mécanisme de partage de connaissances traduit un mécanisme de défaut, c'est-à-dire qu'un objet aura ses propres caractéristiques, plus les caractéristiques si nécessaire des prototypes auxquels il se réfère.

Par contre, le modèle des ensembles se base sur deux niveaux d'abstraction, la classe et l'instance. C'est pourquoi, dans le domaine des représentations par objets, cette approche est plus couramment appelée "approche **classe/instance**". La classe est le concept abstrait qui permet de regrouper des objets ayant les mêmes caractéristiques ; ces objets sont appelés les instances de la classe. Contrairement à la notion de prototype, l'apparition de la notion de classe permet de spécifier explicitement les caractéristiques communes d'un **ensemble** d'objets. Deux types de lien apparaissent dans ce modèle, le lien **est-un**, entre une instance et la classe à laquelle elle appartient, et le lien **sorte-de**, entre deux classes et définissant l'inclusion des propriétés de l'une dans l'autre. Ce modèle organise la base, au travers du lien **sorte-de**, en une hiérarchie de classes traduisant l'inclusion ensembliste et permettant l'héritage des caractéristiques d'un objet en fonction de sa classe d'appartenance.

Le prototypage favorise l'aspect approximatif des connaissances. En contre-partie, le flou qu'il introduit dans la description d'une hiérarchie d'objets est problématique [Brac85] lorsque l'on s'intéresse à des mécanismes, comme la classification d'objets, qui cherchent à établir les liens possibles entre plusieurs objets. Ce genre d'exercice est particulièrement difficile lorsqu'il prend place dans une base possédant un grand nombre de prototypes car, d'une part, l'accumulation d'approximations qui en résulte peut mener à des résultats erronés et, d'autre part, il nécessite la mise au point d'une mesure de similarité pour permettre la comparaison entre prototypes ; ce qui est déjà un problème complexe en soi.

Le modèle à base de classes s'avère plus rigoureux dans la définition des objets qu'il manipule. La relation **sorte-de** qui lie deux classes correspond à l'inclusion ensembliste tandis

que la relation **est-un** qui lie une instance à une classe dénote l'appartenance de l'instance à la classe. En assumant de telles sémantiques pour ces liens, le modèle des ensembles met l'accent sur la cohérence des connaissances. Il peut alors tirer avantage des hiérarchies de classes bien définies pour mettre en place un mécanisme puissant comme la classification d'objet (classe ou instance). A l'inverse des prototypes, on peut reprocher à cette approche sa trop grande rigueur qui ne permet pas l'approximatif. Les connaissances qui permettent d'établir une hiérarchie de classes dénotent déjà un niveau d'abstraction élevé qui n'est possible que si le domaine est bien défini et connu.

Telles qu'elles ont été présentées, ces deux approches forment deux solutions extrêmes dans lesquelles les solutions de l'une fournissent des réponses aux problèmes de l'autre et vice versa. Une réalisation réelle de langage de représentation par objets cherche en général un compromis entre ces deux approches et se réclamera plus de l'une ou de l'autre en fonction des objectifs qu'elle vise. Il est à noter, cependant, que la distinction entre un individu particulier et un concept abstrait, à l'image des notions d'instance et de classe, apparaît dans de nombreux langages de représentation par objets.

La plupart des langages de *frames* sont étiquetés de l'approche par prototypes [Napo92] parce qu'ils favorisent l'introduction d'informations de typicalité, d'exception et de défaut. En revanche, les langages terminologiques et quelques langages de *frames* qui incluent un mécanisme de classification [Dekk94] [Rech85] [Mari&90] suivent plutôt l'approche ensembliste. D'ailleurs, cette distinction entre les deux approches est reprise dans le chapitre IV par rapport aux contraintes qu'impose la mise en place d'un mécanisme de classification sur la description d'une classe dans un modèle à objets.

3.2.2. Les langages de frames

Directement inspirés de l'article de M. Minsky, les langages de *frames* permettent de décrire et gérer un ensemble de connaissances représentées par une hiérarchie d'objets appelés en l'occurrence *frames*. Cette notion commune à tous ces langages se présente comme une structure à trois niveaux, *frame-attribut-facette* [Masi&89]. Le *frame* représente une entité décrite par ses propriétés, l'ensemble des attributs, qui sont eux-mêmes décrits par un ensemble de facettes (cf. figure I.8). Une facette permet de préciser le type de l'information associée à l'attribut, voire sa valeur ou une valeur par défaut (facette déclarative), mais aussi son comportement lorsqu'on y accède en écriture ou en lecture (facette procédurale ou réflexe).

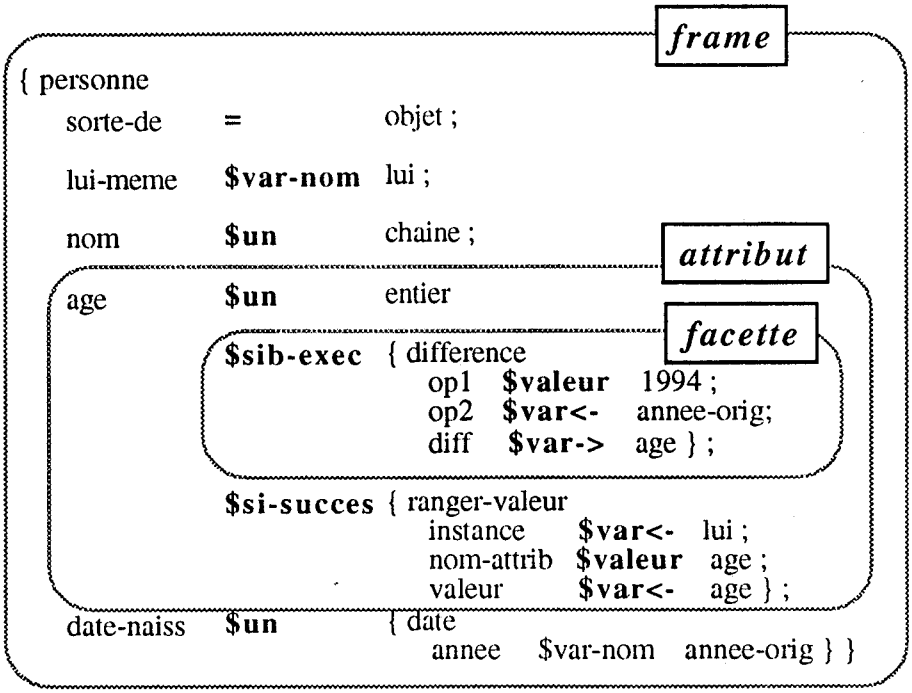

Figure I.8 : Exemple d'une structure de *frame* représentant le concept de personne dans le langage SHIRKA [Rech&89]. Les attributs **nom**, **age**, et **date-naiss** sont les propriétés décrivant le concept de personne. L'attribut **age** possède une facette déclarative (**$un**) indiquant qu'il prend ses valeurs dans le type entier, et deux facettes procédurales (**$sib-exec** et **$si-succes**) indiquant une méthode de calcul de la valeur de cet attribut et la procédure à suivre si une valeur est ainsi obtenue. L'attribut **date-naiss**, quant à lui, est dit **complexe** car il fait référence à un *frame* **date**.

Les *frames* sont organisés en hiérarchie, dite de **spécialisation**, par le lien **sorte-de** (ou parfois **est-un**). Cette hiérarchie de spécialisation est exploitée pour mettre en place un héritage dynamique permettant à un *frame* d'accéder aux structures des *frames* qu'il peut atteindre en suivant le lien de spécialisation **sorte-de**. Cet accès permet au *frame* héritant de récupérer dynamiquement les informations, les attributs et leurs facettes, de ses *sur-frames* ; c'est-à-dire les *frames* auxquels il a accès par héritage.

Un frame peut représenter un individu (instance) ou un concept générique (classe). La distinction classe/instance peut être explicite comme dans KRL [Bobr&77], FRL [Robe&77] et SHIRKA, ou implicite comme dans YAFOOL [Duco&86]. Dans ce dernier cas, une instance est un *frame* pour lequel chaque attribut possède un ensemble de facettes réduit à la seule facette **$valeur** qui désigne une valeur fixe de l'attribut.

La manipulation des *frames* va différer d'un système à un autre en fonction du type d'héritage mis en place, de la sémantique associée aux différentes facette, mais aussi des mécanismes généraux qu'il offre.

Pour ce qui est de l'héritage et des facettes, une technique de masquage est souvent adoptée. Elle permet de filtrer les propriétés héritées à travers la hiérarchie et, plus particulièrement, lorsqu'il s'agit de facettes exprimant l'obtention d'une valeur. Cette technique dénote l'influence de l'approche par prototype.

Certains fournissent un mécanisme d'interrogation par filtrage permettant de retrouver un ensemble d'informations dans l'ensemble des *frames* à partir d'un énoncé lui-même exprimé sous forme d'un *frame*. Ce mécanisme, déjà proposé dans les réseaux sémantiques, apparaît dans les premiers langages de *frames* comme KRL et FRL, mais aussi des systèmes plus récents comme SHIRKA.

Enfin, plus rare, la mise en place d'un mécanisme de classification permet de placer une instance dans une hiérarchie de classes. Ce mécanisme peut être considéré dans cette catégorie de langages à objets comme une originalité qui a été introduite par le système SHIRKA, puis reprise depuis dans le modèle TROPES [Mari93] qui lui succède, ainsi que dans le langage FROME [Dekk94]. La distinction classe/instance nécessaire pour ce genre d'opération donne à ces langages une forte connotation ensembliste qui les rapproche fortement des travaux sur la *subsomption* dans le cadre des langages terminologiques.

3.2.3. Les langages terminologiques

A la différence des langages de *frames* qui ont continué à explorer et exploiter de façon pragmatique les idées mises en avant dans les réseaux sémantiques et les *frames* de M. Minsky, les langages terminologiques sont le fruit d'un travail de formalisation de ces idées. De plus, leur approche est clairement celle des ensembles.

Ce courant trouve ses sources dans les travaux de R.J. Brachman sur le système KL-ONE. Ce dernier est conçu dans l'esprit de fixer les idées sous-jacentes aux réseaux sémantiques et aux *frames* qui souffrent, selon R.J. Brachman [Brac&85], de nombreuses inconsistances et ambiguïtés dues à leur définition informelle. Aussi, KL-ONE est-il présenté comme un système à réseau à structure d'héritage dont l'expression est basée sur un ensemble réduit de types d'objet et de relation indépendants du domaine d'application.

KL-ONE, ainsi que ces successeurs, utilisent un langage de description dans lequel la notion de **concept** représente celle d'objet (classe et instance). Un tel *concept* est exprimé dans un langage de description riche qui permet de combiner d'autres *concepts*, des **rôles** et des **contraintes structurelles**. Un *rôle* décrit une relation que doit entretenir une instance de ce *concept* avec une ou plusieurs instances d'autres *concepts*, tandis qu'une *contrainte structurelle* exprime une interrelation entre des *rôles* du *concept*.

```
PANIER-PIQUE-NIQUE
   ⇒ (AND    PANIER
             (AT-LEAST 2 boisson) (AT-MOST 2 boisson) (ALL boisson VIN)
             (AT-LEAST 3 nourriture) (ALL nourriture CHOSE-COMESTIBLE) )
```

Exemple de définition d'un concept : le *concept* PANIER-PIQUE-NIQUE est défini à partir du *concept* PANIER et donne la description des *rôles* boisson et nourriture en termes des *concepts* respectifs VIN et CHOSE-COMESTIBLE. (syntaxe du langage CLASSIC)

Deux types de *concepts* sont recensés : les **concepts génériques** décrivant un ensemble d'individus, et les **concepts individuels** qui ne décrivent qu'un seul individu au plus. Les *concepts génériques* sont eux-mêmes divisés en deux catégories : le **concept défini** dont la description constitue un ensemble de conditions nécessaires et suffisantes d'appartenance au concept, tandis que le **concept primitif** possède une description incomplète, c'est-à-dire un ensemble de conditions nécessaires mais non suffisantes.

De la même manière que les *frames*, les *concepts génériques*, *primitifs* et *définis*, sont organisés en une hiérarchie d'héritage traduisant les différents niveaux de généralité des *concepts* (cf. Figure I.9). Cette hiérarchie, appelée **taxinomie**, est le résultat d'une structuration de l'ensemble des *concepts* selon la relation d'ordre partiel de **subsomption** [Wood91].

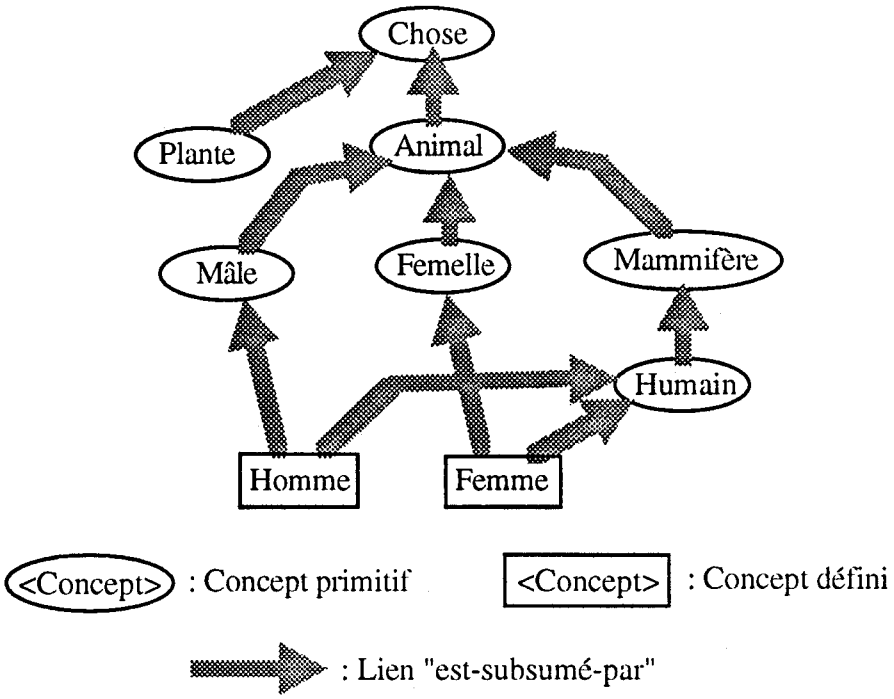

Figure I.9 : Exemple de *taxinomie* de *concepts*, *primitifs* et *définis*, de KL-ONE tiré de [Brac&85].

Cette relation de *subsomption*, qui est au cœur de tous les langages terminologiques, exprime qu'un *concept* est plus général qu'un autre (il le **subsume**). Elle peut être présentée de trois façons distinctes [Napo92], c'est-à-dire qu'un *concept* A subsume un *concept* B si et seulement si :

- en **extension**, tout individu du *concept subsumé* B est aussi individu du *concept subsumant* A ;
- en **intension**, l'ensemble des propriétés d'un individu dont la description est définie par le *concept subsumé* B contient l'ensemble des propriétés qui sont spécifiées par le *concept subsumant* A ;
- en **logique**, être individu décrit par le *concept subsumé* B implique (logiquement) être un individu décrit par le *concept subsumant* A.

En s'appuyant sur la sémantique des constructeurs de description des *concepts*, les langages terminologiques mettent en place un mécanisme, le "**classifieur**" (classifier), permettant d'insérer un nouveau *concept* dans la *taxinomie de concepts* en établissant quels sont ses *concepts subsumants* et *subsumés*. Le nouveau *concept* sera inséré entre les **concepts subsumants les plus spécifiques** (SPS) et les **concepts subsumés les plus généraux** (SPG). Pour ce faire, le *classifieur* s'appuie sur les descriptions des *concepts* et exploite, par conséquent, la définition en *intension* de la *subsomption*.

Mis en place dans KL-ONE, le *classifieur* se heurte au langage de description trop riche et n'assure pas l'établissement de toutes les relations de *subsomption* entre *concepts* ; il n'est pas complet. La réalisation des langages terminologiques suivants prend en compte ce résultat pour spécifier un langage de description épuré qui réduit ou supprime complètement ce problème : CLASSIC [Brac&91], NIKL [Kacm&86], BACK [Nebe88], etc.

Le langage LOOM [MacG&92] fait preuve d'originalité en s'intéressant plus particulièrement au mécanisme permettant de classer une instance dans la hiérarchie de *concepts* et dont l'objectif est de trouver les *concepts* d'appartenance les plus spécifiques. Ce mécanisme, appelé "**reconnaisseur**" (recognizer), est généralement mis en place dans les autres langages

en utilisant au maximum le *classifieur* ; c'est-à-dire que l'instance est remplacée par un *concept* lors d'une phase d'abstraction. En revanche, la version V1-3 de LOOM manipule directement la notion d'instance ; ce qui permet d'éliminer le problème des créations de *concepts* inutiles et d'effectuer des comparaisons *instance-concept* moins coûteuses que les comparaisons *concept-concept*. Etrangement, la version V1-4 de LOOM réintroduit dans le reconnaisseur les créations artificielles de *concepts*, et ne semble pas atteindre, bien que ces créations soit minimisées, l'efficacité obtenue par la version précédente.

Les langages terminologiques fournissent une approche intéressante de la représentation par objets qui tend par son souci de formalisation à standardiser le langage de description de tous les systèmes de cette famille. Bien que le degré d'expressivité de ces langages puisse être critiqué, leurs fondements logiques forment un cadre d'étude adéquat pour analyser l'impact de toute extension future. Par ailleurs, spécialistes de la classification, ils se sont jusqu'à présent surtout intéressés à l'aspect générique des connaissances, mais l'approche du système LOOM ouvre de nouveaux horizons qui devraient permettre d'enrichir substantiellement le langage des assertions consacré à la description des instances.

4. Conclusion

L'architecture générale d'un système à base de connaissances constitue l'un des premiers grands résultats de l'intelligence artificielle : les connaissances nécessaires à une application sont séparées des mécanismes (le contrôle) permettant de les exploiter.

Les problèmes de modélisation, d'acquisition, de validation, d'explication et d'exploitation des connaissances sont autant de contraintes qui participent aux développements de cette architecture. Le point commun à chacun de ces domaines reste l'adaptation du mode de représentation des connaissances au type de raisonnement souhaité. En ce sens, l'étude de l'interdépendance entre raisonnement et expression des connaissances a connu une évolution considérable, multipliant les formalismes de représentation et soulignant l'existence de connaissances de natures différentes.

En s'imposant comme un souci majeur des systèmes à base de connaissances, la structuration les connaissances s'est avérée un domaine d'investigation très fructueux. L'I.A. lui doit, entre autre, avec les travaux clefs comme les réseaux sémantiques et les *frames* de M. Minsky, l'introduction de la notion fondamentale d'**objet**.

Intégré, depuis, dans la plupart des systèmes à base de connaissances, le concept d'objet reste un mode de représentation des connaissances qui mobilise de nombreux travaux de recherche. Ce foisonnement de travaux dénote la difficulté, voire l'impossibilité, de donner une définition précise et arrêtée de ce concept. Toutefois, à l'image même de ce que tentent de réaliser les hiérarchies d'objets, des catégories de représentation par objets peuvent être établies en fonction de leurs caractéristiques particulières. A ce titre, les approches par prototypes et par ensembles fournissent deux stéréotypes de représentation qui se distinguent sur la façon de représenter les objets conceptuellement proches (qui partagent des informations en commun).

Les prototypes favorisent l'aspect approximatif des connaissances et se basent sur une conception informelle de l'objet. Cette perception de l'objet est à l'origine de la famille des langages de *frames*. En revanche, les ensembles fournissent un cadre de définition strict qui correspond généralement mieux aux langages cherchant à mettre en place des mécanismes de classification comme les langages terminologiques et certains langages de *frames*. Le choix d'une approche dépend essentiellement du degré de formalisation du domaine d'application : si l'aspect incertain du raisonnement est un facteur primordial de l'application et que l'établissement a priori d'une hiérarchie d'objets est un problème ardu, voire impossible, ce choix s'orientera vers une solution par prototypes. Par contre, si le domaine d'application est bien défini, que la cohérence des connaissances est assurée et doit être préservée quelqu'en soit l'utilisation, le choix se tournera vers une solution de type ensembliste.

Par la suite, la notion d'objet abordée se réclame plus de l'approche ensembliste car la distinction classe/instance est faite, la relation de spécialisation sur laquelle se fonde une

hiérarchie d'objets est stricte comme la subsomption l'est dans les langages terminologiques et enfin, elle doit permettre de supporter un mécanisme de classification d'instances.

Chapitre II
REPRESENTATION DE CONNAISSANCES PAR OBJETS

1. Introduction

Le terme de "système de représentation de connaissances par objets" est souvent l'objet d'une confusion. En effet, la distinction entre les systèmes de représentation de connaissances basés sur des objets et les langages de programmation à objets utilisés pour représenter la connaissance n'est pas toujours faite, il convient de le rappeler ici. Dans un premier temps, le paradigme des langages de programmation à objets est présenté, puis comparé, dans un second temps, à l'approche représentation par objets.

1.1. Langages de programmation à objets

Les langages de programmation à objets (citons par exemple SMALLTALK-80 [Gold&83] [Robs&81], l'un des plus connus, ou C++ [Stro91], l'un des plus utilisés) proposent la notion de classe comme un moule d'instances.

Chaque classe décrit la structure et le comportement de ses instances. Ces deux composants d'une classe peuvent être hérités des sur-classes conformément au graphe d'héritage. La structure est donnée par un ensemble de champs, appelés **variables d'instance** ou **attributs** ; les comportements sont donnés par un ensemble de **méthodes** qui permettent de manipuler les variables d'instances (cf. Figure II.1). Ces méthodes sont décrites par leur **sélecteur** (le nom de la méthode), la liste des paramètres et le corps de la méthode.

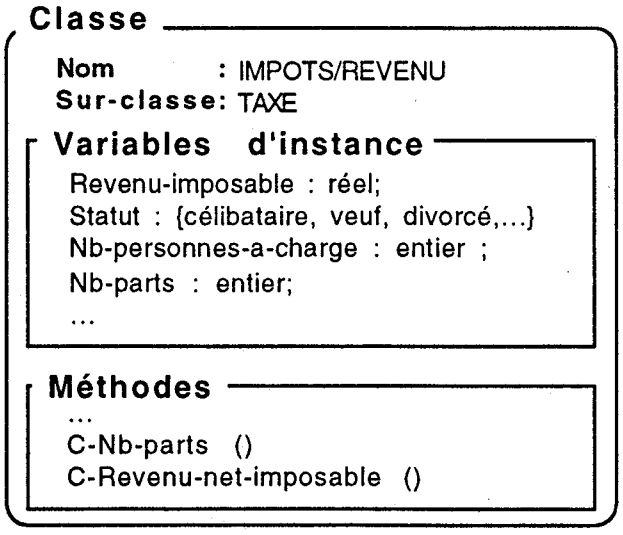

Figure II.1 : Exemple d'un schéma de classe d'un langage de programmation à objets. Suivant le principe d'encapsulation, les variables d'instances ne sont accessibles qu'à travers les méthodes qui définissent l'interface entre les objets de la classe et le monde extérieur (le programme qui les utilise). Ainsi, une instance d'IMPOTS/REVENU verra ses variables *statut* et *Nb-personnes-a-charge* évaluées lors du calcul du *C-Nb-parts*.

Contrairement au principe adopté par les représentations de la connaissance par objets, le principe d'**encapsulation** permet de percevoir un objet comme une boite noire qui cache sa

structure de donnée et communique avec l'extérieur via l'interface définie par l'ensemble des méthodes[1]. Ce principe interdit l'accès direct en lecture et écriture des variables d'instances. Toutes les manipulations possibles des variables d'instances sont donc définies par les méthodes dès la conception des classes. La notion de **message** est introduite pour mettre en place la communication entre objets ; c'est-à-dire pour utiliser leurs méthodes. Un message indique le **receveur**, l'objet concerné, et le sélecteur et les arguments de la méthode à appliquer [Masi&89].

Contrairement aux valeurs particulières des champs, les comportements (méthodes) ne sont pas portés par les instances. L'envoi d'un message à une instance déclenche la recherche de la méthode à exécuter dans la classe d'instanciation en fonction du sélecteur mentionné dans le message.

La procédure de création d'une instance est elle-même redéfinissable pour une classe particulière ; ceci est possible lorsqu'un niveau décrivant les classes est introduit dans le modèle. En effet, une idée originale des langages de programmation objet, comme SMALLTALK, est de considérer l'univers des objets comme uniforme : "tout est objet". Ainsi les classes sont des instances d'autres classes que l'on appelle **métaclasses**. Ces métaclasses permettent de donner la structure d'une classe et les méthodes d'utilisations de celle-ci. Lorsqu'une classe nécessite une méthode de création particulière pour ses instances, une métaclasse est créée afin de décrire cette méthode particulière. Toute classe qui est instance de cette métaclasse possédera cette méthode de création d'instance.

Pour conclure sur cette succincte présentation, il faut noter que ces langages ont avant tout pour vocation la programmation, ils s'intéressent donc à la notion d'objet essentiellement sous le point de vue structuration de données/programmes. Dans ce contexte, le mécanisme d'héritage reste un moyen de réutilisation de structure et de code.

1.2. Représentation et programmation par objets

Lorsqu'on utilise un langage de programmation à objets tel que SMALLTALK-80, ROME [Carr89], C++,... pour représenter des connaissances au sens où on l'entend en I.A., faits et contrôle ne peuvent être qu'intimement liés sous forme de programmes. Comme le souligne Patel-Schneider [Pate&90], les systèmes de programmation à objets, bien qu'ils soient souvent utilisés pour représenter la connaissance, ne sont pas, par nature, réellement appropriés à cette tâche. Ils ne peuvent que fournir une représentation procédurale de la connaissance. Ils sont trop souvent amenés à mélanger l'information destinée à la représentation des connaissances à une information concernant des détails d'implémentation qui ne concerne donc pas la représentation. De plus, leur puissance d'expression limitée ne peut être palliée que par l'écriture de procédures à la charge de l'utilisateur, notamment lorsqu'il s'agit d'extraire de la base des informations implicites.

Au contraire, dans les systèmes de représentation de connaissances basés sur des objets, la connaissance est décrite de manière déclarative et se trouve séparée des mécanismes d'exploitation. Les structures de données (objets) effectivement manipulées par ce type de système sont entièrement dédiées à la représentation. Les mécanismes d'exploitation sont définis par des programmes écrits en n'importe quel langage et sont chargés d'exploiter et de raisonner sur les bases de connaissances, dont ils sont effectivement séparés. Une logique de représentation est associée à ces systèmes ; elle définit la sémantique des diverses entités du système (classes, instances, attributs) et permet de déduire des faits implicites des connaissances déjà établies, notamment par le mécanisme d'héritage qui repose sur la structuration en hiérarchies des classes d'une base.

---

[1]Dans certains langages de programmation par objets, l'encapsulation peut être partielle. Par exemple, dans C++ ou Eiffel [Meye90], les droits d'accès (*public* ou *private* pour C++ et *export* pour Eiffel) aux variables d'instances et aux méthodes sont définis dans chaque classe. Ces droits d'accès déterminent ce qui est encapsulé et ce qui ne l'est pas.

Ce chapitre aborde la notion d'objet dans le contexte des représentations des connaissances par objets. La première partie (cf. §II.2) est consacrée à la présentation des éléments de description classique des objets, tandis que la seconde partie (cf. §II.3) se propose d'introduire un ensemble de mécanismes permettant d'exploiter les objets. Enfin, un aperçu du modèle TROPES (cf. §II.4) illustre l'approche d'une représentation par objets qui met en place un raisonnement classificatoire, et qui introduit de nouvelles notions augmentant les possibilités de structuration des connaissances : le *concept* et le *point de vue*.

2. Présentation de la notion d'objet

La notion d'objet s'articule autour des notions d'attribut, d'instance et de classe. Un objet est perçu à travers la donnée de ses attributs. Si cet objet est une instance, il est un individu particulier de l'univers du discours, et ses attributs sont vus comme des valeurs ; si c'est une classe, l'objet est alors une abstraction représentant un ensemble d'instances (individus), les attributs sont alors vus comme des descriptions, c'est-à-dire des domaines de valeurs. La hiérarchie de classes organisée par une relation de spécialisation décrit l'univers des instances.

Par la suite, les diverses notions qui composent une représentation par objets "classique" sont présentées en soulignant leurs relations. Le "pseudo-formalisme" employé cherche uniquement à donner les grandes lignes des représentations par objets nécessaires au travail présenté par la suite, il est parfois accompagné d'exemples ou de références à des systèmes existants. On trouvera dans [Duco95] la proposition d'un cadre formel valable aussi bien pour le noyau de tous systèmes à objets que pour les logiques terminologiques.

2.1. La notion d'instance

Une instance est un individu particulier qui est décrit par un ensemble de valeurs d'attributs. Classiquement, dans les systèmes de représentation, la notion d'instance n'a de réalité que par rapport à la notion de classe. Dans ce cas, l'ensemble des attributs intéressants d'une instance est égal à l'ensemble des attributs déclarés dans les classes d'appartenance de cette instance.

On parle d'**instanciation** quand une instance est créée sur le modèle d'une classe ; cette classe est alors dite **instanciée**. Cette terminologie est empruntée au langage de programmation à objets.

Néanmoins, la prépondérance de la notion de classe sur celle d'instance n'est pas toujours aussi nette. En effet, une instance est dite **rattachée** à une classe lorsque l'instance possède une existence antérieure à cette action de rattachement. La notion de rattachement est introduite par les systèmes qui permettent à une instance de migrer d'une classe à une autre, a fortiori ceux qui mettent en place un mécanisme de classification d'instances.

L'idée sous-jacente au rattachement est d'accorder à une instance le droit d'exister seule ; le rattachement à une classe est alors considérée comme une information supplémentaire sur l'instance. Dans le cas des langages de *frames*, comme SHIRKA, SHOOD, et FROME, qui permettent la migration d'instance, la création d'une instance reste cependant le fait d'une instanciation de classe, par défaut celle de la classe la plus générale (souvent appelée OBJET) qui décrit la structure minimale d'un objet.

Par contre, la capacité d'une instance à être créée sans passer par le modèle fourni par une classe est adoptée dans les langages terminologiques. En effet, ces langages peuvent déclarer l'existence d'une instance (un *individu*) sans plus d'indication, puis la décrire progressivement par la donnée d'assertions. Parmi ces assertions, les références à des classes (*concepts*) ne se démarquent pas des autres informations ; mais leurs descriptions sont appliquées à l'instance pour en enrichir la structure.

Exemple d'une description progressive d'un individu dans les langages terminologiques (tiré de [Napo92]) :

- create-ind [Rocky] ; **création** a priori de l'individu Rocky
- assert-ind [Rocky (FILLS vehicule-conduit volvo-17)] ; ajout d'information sur le
- ... ; rôle *vehicule-conduit*
- assert-ind [Rocky Personne] ; **rattachement** de Rocky au concept Personne

La notion de rattachement offre davantage de souplesse que celle d'instanciation car elle autorise la manipulation d'instance dont la structure n'est pas d'emblée complètement déterminée. Aussi, dans la suite, pour inclure cette notion de rattachement et parler de création d'instance, le terme de **configuration** d'instance sera préféré à celui d'instanciation dont il englobe les fonctionnalités : **configurer** une instance consiste à donner les informations connues concernant les attributs, leurs valeurs, et les rattachements aux classes. Ainsi, l'instanciation devient un cas particulier de configuration dans lequel seule une classe de rattachement est donnée comme information connue de l'instance.

Dans la suite, on note :

- A(I) l'accès à la valeur de l'attribut A pour l'instance I.

2.2. Description d'attribut

Un attribut peut représenter une propriété d'un objet, ou une relation qu'un objet entretient avec un (ou plusieurs) autre objet. Par exemple, la *couleur*, le *modèle* pourraient être des attributs décrivant une propriété d'un objet voiture, alors que le *moteur* pourrait être, quant à lui, un attribut qui met en relation (composition) l'objet voiture avec un objet moteur.

Outre le nom, plusieurs autres informations sont généralement attachées à un attribut :

1) les informations comportementales, indiquant la procédure à suivre lorsqu'on l'accède en écriture.

   Cette spécialité des langages de *frames* est réalisée en associant à l'attribut la procédure à exécuter lors d'une modification de l'attribut, c'est-à-dire lors de l'ajout ou retrait d'une valeur. Deux facettes permettent de prendre en compte ces deux cas de réflexe [Masi&89] : (**$si-ajout** <proc>), où la procédure <proc> est exécutée lors d'un ajout ; et (**$si-enleve** <proc>), où la procédure <proc> est exécutée lors d'une suppression.

2) les informations calculatoires, fournissant les moyens d'obtention de sa valeur. Ces informations peuvent avoir un caractère sûr ou hypothétique.

   Ainsi, la facette (**$defaut** <valeur>) des langages de *frames* permet d'introduire une valeur par défaut dans l'attribut.

   Il est aussi possible d'imposer une valeur à l'attribut par le biais d'une facette (**$valeur** <valeur>) dans les langages de *frames*, ou par la restriction de rôle (attribut) (**FILLS** <rôle> <valeur>) du langage terminologique CLASSIC [Brac&91] (par exemple : (**FILLS** couleur blanc)).

   Enfin, l'obtention d'une valeur d'attribut peut être dynamique soit par application d'une procédure associée à l'attribut, appelée attachement procédural, à travers la facette **$sib-exec**, soit par l'expression de contraintes inter-attributs (cf. §II.3.4.2).

3) les informations descriptives, concernant son domaine de valeurs. Pour ce faire, un ensemble de constructeur permet de préciser :

   - le **type** de l'attribut.

     C'est le cas des facettes (**$un** <type>), (**$liste** <type>), (**$ensemble** <type>), des langages de frames qui permettent de désigner le type de référence et son caractère structurel (mono-valué ou multi-valué).

Dans les langages terminologiques, les rôles sont a priori multi-valués, et le type de référence d'un rôle est exprimé par (**ALL** <rôle> <concept>), où <concept> désigne le concept prédéfini, primitif, ou défini, dans lequel le rôle prend ses valeurs.

- les **restrictions** que l'on désire appliquer à ce type. Ces restrictions peuvent être des réductions de domaine :
  - par **exclusion** de valeurs (facette (**$sauf** <ensemble-valeurs> dans les langages de frames).
  - par **énumération** explicite des valeurs d'un domaine.

    Dans les langages de frames, cela correspond aux facettes (**$intervalle** [<val1> .. <val2>]) pour des domaines ordonnés, (**$domaine** <ensemble-valeurs>) dans les autres cas.

    Dans les langages terminologiques, le constructeur **ONE-OF** permet d'introduire des énumérations dans les descriptions de concept.
  - par **réduction de la cardinalité** à un valeur entière précise ou à un intervalle d'entiers, quand l'attribut est multi-valué.

    Dans les langages de *frames*, cette réduction se fait par la facette (**$card** [<card-min> .. <card-max>]), et dans les langages terminologiques par les restrictions de rôles (**ATLEAST** <card-min> <rôle>) et (**ATMOST** <card-max> <rôle>).
  - par des **conditions dynamiques** à vérifier par les valeurs de l'attribut.

    Une méthode usuelle consiste à introduire une fonction (code procédurale) qui prend la nouvelle valeur proposée en entrée et rend comme résultat l'acceptation ou non de la valeur. Cette méthode est mise en place par la facette (**$si-possible** <fonction>) des langages de frames ou, dans un langage terminologique comme CLASSIC, par l'introduction de l'opérateur (**TEST-C** <fonction>) ; où <fonction> est le prédicat à appliquer.

    Une autre méthode, moins courante, consiste à exprimer les conditions dynamiques à l'aide d'un langage de contraintes qui permet alors l'expression d'interdépendances d'attributs (cf. §II.3.4.2).

Le type de référence d'un attribut peut être un type de base prédéfini comme **entier**, **réel**, **chaîne**... ou encore, quand l'attribut représente une relation entre objets, un ensemble d'objets dénoté par la (ou les) classe à laquelle ils doivent appartenir. Par exemple, l'attribut *propriétaire* d'un objet *voiture* peut avoir pour domaine l'ensemble des objets *Personne*. De plus, dans certains langages, comme les langages terminologiques, les types de bases sont exprimés comme des classes spéciales prédéfinies. Cette approche permet d'avoir une écriture et un traitement uniformes des descriptions d'attribut.

La description du domaine de définition d'un attribut est une information primordiale sur laquelle reposent, d'une part, la construction des hiérarchies d'objets et, d'autre part, les mécanismes d'identification d'objets. Elle est donc à la base d'une approche classe/instance.

Par la suite, afin d'éviter de surcharger la présentation par les différents formalismes d'objets, la description du domaine Dom(A, C) d'un attribut A dans une classe C sera désignée de manière générale par le prédicat $\delta_{A,C}$ qui, appliqué à une instance I, indique si la valeur A(I) appartient au domaine Dom(A, C) :

- $\forall I, \delta_{A,C}(I) \Leftrightarrow A(I) \in Dom(A, C)$

2.3. Description de classe

Dans cette partie, seules les descriptions d'attributs sont introduites pour définir la notion de description de classe. En revanche, toutes les informations relatives à la notion de hiérarchie de classes qui peuvent prendre place dans la description d'une classe (notamment pour exprimer

la factorisation de description que permet la relation de spécialisation) sont présentées dans la partie suivante de ce chapitre.

Une description de la classe C est ici donnée sous la forme d'un prédicat, noté Δ_C, qui permet de déterminer si une instance I vérifie les conditions nécessaires d'appartenance à l'ensemble Ext(C) d'instances représenté par la classe C. L'ensemble Ext(C) est appelé l'**extension** de la classe C.

Notations :

- $\mathcal{A}_C$: l'ensemble des attributs définis dans la classe C.

Le prédicat[2] Δ_C peut être défini en fonction des attributs d'attributs de $\mathcal{A}_C$ de la façon suivante :

- $\forall$ I, Δ_C (I) = $\bigwedge_{A \in \mathcal{A}_C} \delta_{A,\,C}(I)$

Cette présentation simplifiée de la description de classe omet de préciser, d'une part, la relation existante entre I $\in$ Ext(C) et Δ_C(I), et d'autre part, la valeur de Δ_C(I) lorsqu'un attribut de la classe C n'est pas connu pour I.

Pour ce qui est de la relation entre I $\in$ Ext(C) et Δ_C(I), les *concepts primitifs* et *concepts définis* des langages terminologiques fournissent deux solutions possibles. En effet, les descriptions des *concepts primitifs* fournissent un ensemble de conditions nécessaires d'appartenance à une classe, tandis que les *concepts définis* sont décrits en termes de conditions nécessaires et suffisantes :

- *concepts primitifs* : $\forall$ I, I $\in$ Ext(C) $\Rightarrow$ Δ_C(I)
- *concepts définis* : $\forall$ I, I $\in$ Ext(C) $\Leftrightarrow$ Δ_C(I)

L'intérêt d'imposer l'équivalence est de pouvoir se baser sur la description d'une classe pour établir l'appartenance d'une instance à la classe. Cette propriété est donc nécessaire pour assurer le succès d'un mécanisme de classification d'instance chargé de rattacher cette instance dans sa (ou ses) classe la plus spécifique [Mari93]. Dans la plupart des langages à objets qui ne considèrent qu'un type de classe, la description d'une classe est seulement vue comme un ensemble de conditions nécessaires, à l'image des *concepts primitifs* des langages terminologiques.

Les langages qui suivent l'approche des prototypes font figure d'exception puisque toute information fournie par la description d'une classe est considérée comme une connaissance par défaut [Masi&89]. La description d'une classe ne peut donc même pas être perçue comme un ensemble de conditions nécessaires. Cette approche n'est pas étudiée par la suite

La possibilité de manipuler une instance incomplète I (certains attributs ou leurs valeurs lui font défaut) pose, quant à elle, le problème de sa confrontation à une description de classe C. Ce problème souligne l'inadéquation d'un prédicat logique à deux valeurs ("vrai" ou "faux"), comme l'est sous-entendu a priori Δ_C.

La solution adoptée par un système comme SHIRKA est de considérer une logique tri-valuée permettant de statuer par "sûr", "impossible" ou "possible". Les deux premiers cas se ramènent à ceux traités par la logique binaire tandis que le troisième prend en compte la situation dans laquelle aucune description d'attribut n'est contredite (ce n'est pas "impossible"), et certains attributs nécessaires restent inconnus (ce n'est pas "sûr"). D'autres solutions consistent à distinguer des types d'attributs selon l'importance que l'on doit leur accorder dans la description d'une classe. Dans ces cas, ces types d'attributs sont interprétés et pris en compte

---

[2] Pour être générale, cette définition devrait aussi inclure les descriptions des contraintes inter-attributs pouvant apparaître dans la description de la classe. L'exposé, malgré cette omission, reste général et gagne en clarté. Par ailleurs, l'utilisation de dépendances entre attributs est abordée dans la suite (Cf. §3).

dans la définition du prédicat Δ_C qui peut parfois être transformé complètement en un calcul de compatibilité [Dekk94].

Les deux points soulignés, existence de différents types de classe et problème des instances incomplètes, sont repris et développés dans la partie II de ce document.

2.4. Hiérarchie de spécialisation de classes

Une hiérarchie de classes est un graphe orienté sans circuit de classes liées par la relation de spécialisation et qui possède un maximum appelé la **classe racine**. Cette dernière est donc par définition une classe dont toutes les autres sont des spécialisations et qui n'est spécialisation d'aucune autre.

L'univers "objet" introduit une riche terminologie autour des concepts de classe, instance et spécialisation dont les termes les plus connus méritent d'être rappelés ici. Lorsque la classe C est spécialisée par la classe C', la classe C est dite **sur-classe** de C', et inversement la classe C' est dite **sous-classe** de C. On dit aussi parfois que la description de la classe C' **affine** la description de la classe C. Le lien de spécialisation par lequel est déclarée une sur-classe d'une classe est habituellement nommé **sorte-de**, alors que le lien de rattachement d'une instance à une classe est généralement nommé **est-un**.

2.4.1. Spécialisation de classes

La spécialisation étant aux objets ce que la subsomption est aux termes (cf. §I.3.2.3), deux définitions de la spécialisation peuvent être données : intensionnelle ou extensionnelle. Avec la notation $C \leftarrow C'$ pour indiquer que C' est une spécialisation de C, les définitions sont :

- Extensionnelle :

  $C \leftarrow C' \Rightarrow (\forall I, I \in Ext(C') \Rightarrow I \in Ext(C))$

- Intensionnelle

  $C \leftarrow C' \Rightarrow (\forall I, \Delta_{C'}(I) \Rightarrow \Delta_C(I))$

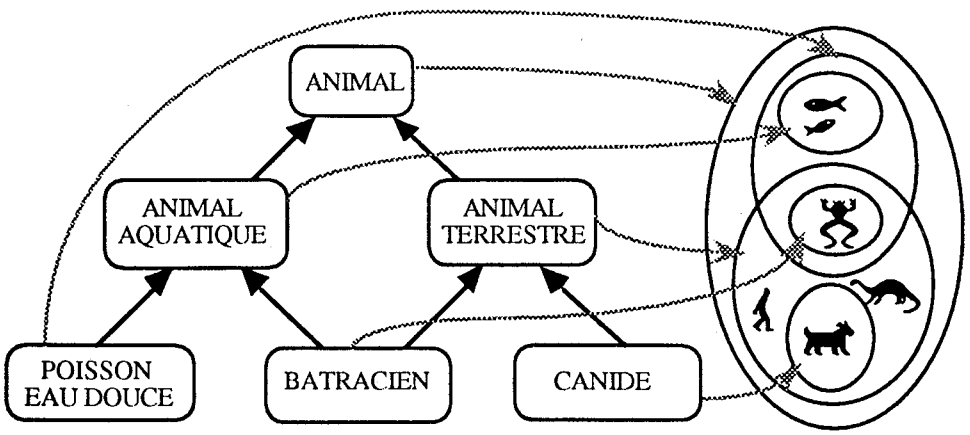

Figure II.2 : Exemple de hiérarchie de spécialisation de classes (à gauche) et son équivalent ensembliste (à droite). La classe BATRACIEN est un exemple de **multi-spécialisation**, c'est-à-dire qu'elle possède deux liens de spécialisation directe, l'un avec ANIMAL-TERRESTRE et l'autre avec ANIMAL-AQUATIQUE. Au niveau ensembliste, la multi-spécialisation se traduit par l'inclusion de l'ensemble BATRACIEN dans l'intersection des ensembles ANIMAL-AQUATIQUE et ANIMAL-TERRESTRE, et la transitivité de la spécialisation implique que l'ensemble CANIDE est inclus dans l'ensemble ANIMAL, etc.

La définition intensionnelle implique que :

- tout attribut A de C est un attribut de C' : $\mathcal{A}_C \subseteq \mathcal{A}_{C'}$; la description de C' peut donc introduire de nouveaux attributs.
- et, bien sûr, $(\forall A \in \mathcal{A}_C, \forall I, \delta_{A, C'}(I) \Rightarrow \delta_{A, C}(I))$.

La relation de spécialisation établit un ordre partiel entre les classes (définition intensionnelle) qui, dans une approche ensembliste (définition extensionnelle), se traduit par l'inclusion des ensembles d'instances représentés par ces mêmes classes (cf. Figure II.2).

Il est à noter que la définition intensionnelle de la spécialisation ne permet pas d'assurer la spécialisation extensionnelle lorsque les descriptions de classe n'imposent pas des conditions suffisantes d'appartenance. Sauf cas exceptionnel d'une approche par prototypes, la spécialisation extensionnelle entraîne par contre la spécialisation intensionnelle.

La relation de spécialisation pour être effective dans un modèle à objets nécessite la mise en place d'un mécanisme puissant : l'héritage entre classes.

2.4.2. L'héritage entre classes

L'héritage est le mécanisme qui met en œuvre la récupération des informations qu'une classe partage avec ses sur-classes. Ces informations peuvent être d'origine assertionnelle (propriétés d'attributs), structurelle (définition des structures des instances d'une classe), ou opérationnelle (héritage de procédure).

Du fait du mécanisme d'héritage, la hiérarchie de spécialisation peut connaître une construction incrémentale et "économique" : seule la partie de description qui distingue chaque classe de ses sur-classes est donnée (cf. figure II.3).

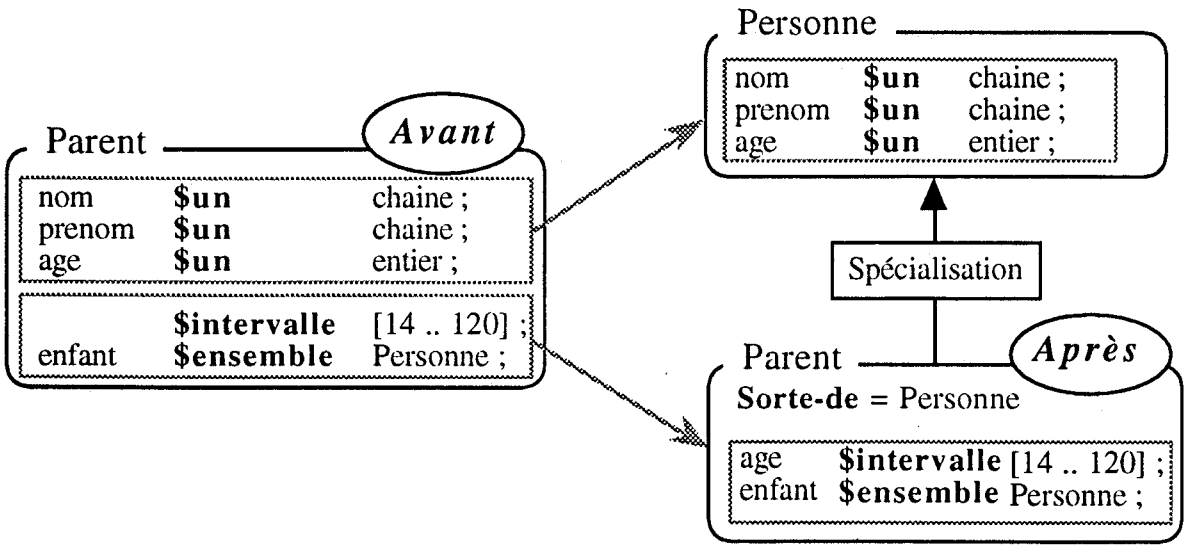

Figure II.3 : Description économique de la classe Parent assurée par l'héritage de la description de la classe Personne. Cette description avec héritage est d'autant plus pertinente que la classe Parent (*Avant*) sous-entend l'existence simultanée d'une classe Personne dans la description de son attribut *enfant*. De plus, l'héritage de la description de la classe Personne assure que la classe Parent (*Après*) est équivalente à celle de la classe Parent (*Avant*).

Le principal problème que l'on rencontre avec l'héritage apparaît lorsqu'une hiérarchie de spécialisation présente des cas de multi-spécialisation : une classe est la spécialisation d'au moins deux autres qui n'entretiennent entre elles aucun lien de spécialisation. En d'autres termes, C est une multi-spécialisation si et seulement si :

- $\exists$ C1, C2 ; (C1 $\leftarrow$ C) $\wedge$ (C2 $\leftarrow$ C) $\wedge$ $\neg$(C1 $\leftarrow$ C2) $\wedge$ $\neg$(C2 $\leftarrow$ C1))

Le traitement d'un tel cas peut poser des problèmes de conflits entre ce qui est hérité de part et d'autre. On trouvera dans [Duco&89] [Duco93] [Duco&95] une riche synthèse sur les problèmes de conflits (conflit **de valeur**, **de nom** ou d'**homonymie**) introduit par l'héritage multiple et sur les diverses solutions proposées. Deux catégories de conflits sont distingués:

Indépendamment de ces problèmes d'héritage, la prise en compte de l'aspect progressif de la construction de la hiérarchie par héritage permet de revoir la définition de la description d'une classe.

2.4.3. Prise en compte de l'héritage dans la description d'une classe

Grâce à l'héritage, la description d'une classe peut faire l'économie des descriptions d'attributs communes avec ses sur-classes.

Une description locale Δ_C^H de la classe C est donnée par trois types d'informations :

- $\mathcal{A}_C^H$ un ensemble d'attributs décrits localement dans C et inclus dans $\mathcal{A}_C$.
- Pour tout attribut A de $\mathcal{A}_C^H$, une description $\delta_{A,C}^H$ de A dans C qui vérifie :

  $$\forall I, \delta_{A,C}(I) \Rightarrow \delta_{A,C}^H(I)$$

- $\mathbf{S}(C) = \{C' / (C' \leftarrow^1 C)\}$; où $\leftarrow^1$ représente un lien de spécialisation directe, et donc $\mathbf{S}(C)$ représente l'ensemble des sur-classes directes de C.

Avec cette nouvelle définition, le prédicat Δ_C^H donné par :

- $$\forall I, \Delta_C^H(I) = \bigwedge_{A \in \mathcal{A}_C^H} \delta_{A,C}^H(I)$$

  permet de déterminer la validité d'une instance I par rapport aux attributs déclarés localement dans la classe C.

A partir de cette nouvelle définition, le prédicat global Δ_C peut alors être formulé de la façon suivante :

- $$\forall I, \Delta_C(I) = \left[\bigwedge_{C' \in \mathbf{S}(C)} \Delta_{C'}(I)\right] \wedge \Delta_C^H(I)$$

  Ce qui signifie qu'une instance vérifie la description de C si elle vérifie toutes ses sur-classes et vérifie les conditions locales à la classe

Pour les descriptions d'attributs, on a, de façon équivalente, la propriété :

- $$\forall I, \delta_{A,C}(I) = \left[\bigwedge_{C' \in \mathbf{S}(C)} (A \in \mathcal{A}_{C'}^H \Rightarrow \delta_{A,C'}^H(I))\right] \wedge (A \in \mathcal{A}_C^H \Rightarrow \delta_{A,C}^H(I))$$

  En d'autres termes, le domaine d'un attribut dans une classe est l'intersection de tous les domaines décrits par les descriptions de cet attribut dans toutes ses sur-classes, et de la description (si elle existe) propre à la classe.

La conséquence immédiate d'une construction de hiérarchie de classes suivant le principe d'héritage est que la relation de spécialisation (définition intensionnelle) est systématiquement assurée ; une classe héritant la description de ses sur-classes entretient nécessairement une relation de spécialisation intensionnelle avec celles-ci (cf. l'exemple précédent des classes Parent et Personne, Figure II.3).

3. Objet et Raisonnement

Outre le mécanisme d'héritage qui permet essentiellement de réaliser la factorisation des connaissances, une représentation des connaissances par objets peut se doter de mécanismes spécialisés dans des tâches précises et spécifiques à un niveau, ou type, de connaissances particulier : classe, instance et attribut.

3. Objet et Raisonnement

Outre le mécanisme d'héritage qui permet essentiellement de réaliser la factorisation des connaissances, une représentation des connaissances par objets peut se doter de mécanismes spécialisés dans des tâches précises et spécifiques à un niveau, ou type, de connaissances particulier : classe, instance et attribut.

Pour les classes (cf. §II.3.1) et les instances (cf. §II.3.2), la hiérarchie de classes devient un support intéressant pour guider un raisonnement de classification. Il s'agit, dans le premier cas, de trouver la place à laquelle insérer une classe dans la hiérarchie de spécialisation et, dans le deuxième cas, de trouver la ou les classes potentielles d'appartenance de l'instance. La classification de classe, aussi appelée catégorisation, nécessite un mécanisme d'appariement entre classes, tandis que la classification d'instance, aussi appelée identification ou reconnaissance, nécessite un mécanisme d'appariement entre instance et classe.

Les raisonnements au niveau des attributs (cf. §II.3.3) consistent généralement en des mécanismes d'inférence chargés d'évaluer un attribut dans le contexte particulier d'une instance. Ces mécanismes traduisent souvent des interdépendances entre attributs qui peuvent être exprimées de différentes façons (cf. §II.3.4). La plupart de ces expressions sont inscrites dans les descriptions de classes et peuvent être exploitées pour une évaluation d'un attribut lors d'un rattachement d'une instance à une classe. Néanmoins, cette solution gênante lorsqu'un système adopte un profil "classification" prononcé, peut être abandonnée au profit d'un principe de délocalisation des connaissances nécessaires aux inférences de valeurs d'attribut. Ces connaissances ne sont plus exprimées à travers les descriptions de classes.

3.1. Raisonnement sur les classes

Niveau de raisonnement de prédilection des langages terminologiques, le raisonnement sur les classes repose sur un mécanisme de comparaison de deux descriptions de classes. Son objectif est de répondre à la question : "est-ce que l'une des deux classes spécialise (est subsumée par) l'autre" [Brac&85] (cf. §I.3.2.3). Une fois qu'un système inclut un tel mécanisme d'appariement de classe, un mécanisme de classification de classe dans une hiérarchie peut être mis en place.

3.1.1. Appariement de classes

L'appariement de deux classes consiste donc à comparer leur description pour établir s'il existe une relation de spécialisation entre elles.

Dans le contexte des langages terminologiques, l'appariement de classes correspond au prédicat **subsume (C1, C2)** permettant de savoir si le *concept* C1 **subsume** le concept C2. Chaque *concept* étant décrit en termes d'autres *concepts* et de *rôles*, la première étape de ce prédicat consiste généralement à effectuer une normalisation visant à exprimer chacun des *concepts* C1 et C2 en termes des *concepts primitifs* sur lesquels ils sont construits et à réécrire leurs descriptions de *rôles* sous une forme canonique [Borg91] [Borg92]. L'établissement de la subsomption entre les deux *concepts* est réalisé en vérifiant que chaque *concept primitif* de C1 subsume un *concept primitif* de C2, et que les descriptions de chaque *rôle* de C1 subsument celles de ce même *rôle* dans C2.

Bien que moins systématique, l'appariement de classes est aussi traité dans le contexte des langages de *frames*. L'approche proposée dans [Capp94] consiste à typer les attributs dans chaque classe en appliquant un processus de normalisation sur leurs descriptions. Une relation de sous-typage est alors établie en accord avec la relation de spécialisation. Quand un attribut est complexe, c'est-à-dire que son domaine est un ensemble d'objets, la relation de sous-typage de cet attribut se confond avec la relation de spécialisation. Comme résultat de ce typage, un graphe de type est associé à chaque attribut d'une hiérarchie de classes (Figure II.4).

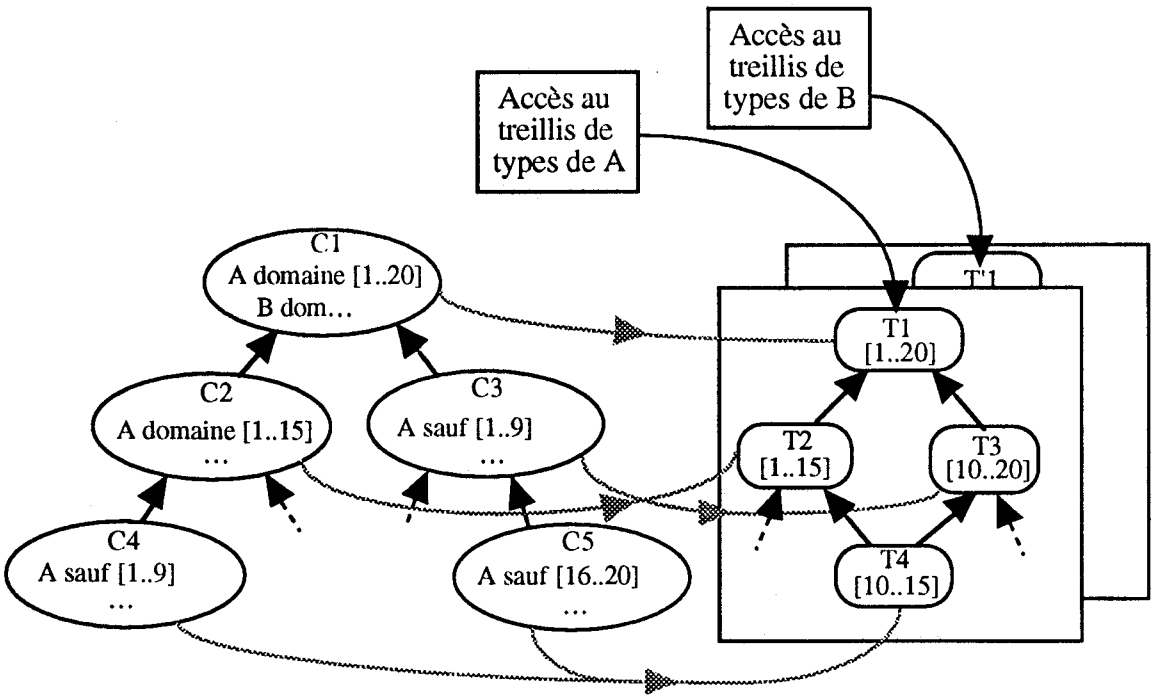

Figure II.4 : Exemple d'une hiérarchie de classes avec les treillis de types d'attributs associés.

A partir de ces hiérarchies de types, et de façon similaire à l'utilisation de relations de subsomption entre rôles dans les langages terminologiques, l'appariement entre deux descriptions de classes peut être effectué pour déterminer l'éventuelle relation de spécialisation qu'elles pourraient entretenir.

Une approche directement inspirée des travaux sur les langages terminologiques définit la relation de spécialisation des langages de *frames* en termes de subsomption, la **O-subsomption** [Napo&92]. Cette O-subsomption se fonde aussi sur une interprétation du langage de description des domaines d'attributs selon une relation de sous-typage.

Enfin, [Bisso94] propose de remplacer le calcul de subsomption ou de spécialisation entre classe par un calcul de similarité. Cette article constitue une étude préliminaire de ce que pourrait être la mesure de similarité dans une représentation par objets. La démarche consistant à inclure la notion d'approximation de connaissances au niveau même de la structuration des classes ouvre des perspectives prometteuses sur les possibilités d'appréhender des domaines en cours de formalisation ; ces derniers se heurtent généralement à la trop grande rigidité d'un modèle à objets fondé sur une relation de spécialisation stricte dès la phase de conception de la base de connaissances.

3.1.2. Classification de classes

A partir d'un mécanisme d'appariement de classes, il est alors possible d'envisager la mise en place d'un mécanisme de classification de classe dont l'objectif est de confronter une nouvelle définition de classe avec la hiérarchie de spécialisation déjà existante. Le principe général de la classification d'une classe C est d'effectuer la recherche des classes les plus spécifiques de la hiérarchie qui spécialisent C, ainsi que les plus générales de la hiérarchie qui sont spécialisées par C. L'étape finale consiste à insérer la classe entre les classes de ces deux ensembles.

L'exemple ci-dessous (cf. Figure II.5) illustre la classification de classe dans les langages terminologiques. Le parcours du graphe permet de construire l'ensemble SPS, des *concepts* Subsumants les Plus Spécifiques, et l'ensemble SPG, des *concepts* Subsumés les Plus Généraux.

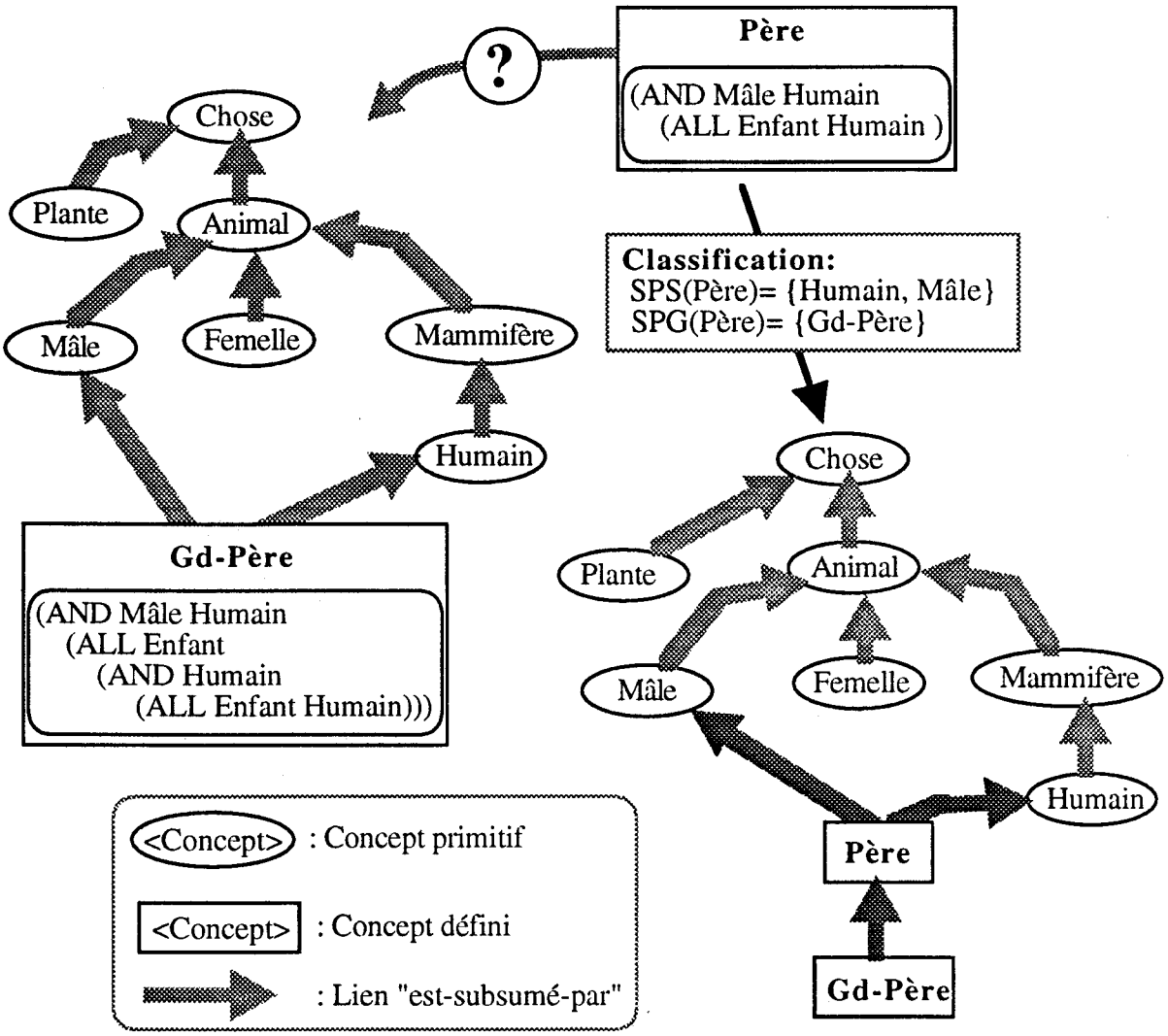

Figure II.5 : Exemple d'une classification de *concept* dans les langages terminologiques. Le *classifieur* calcule les deux ensembles SPS et SPG pour le *concept* Père. Celui-ci s'avère être subsumé par Humain et Mâle, d'après sa définition, et subsume Gd-Père dont le *rôle* Enfant est plus restreint que celui de Père. Père est inséré en fonction de ces deux subsumants et de ce subsumé.

Dans le cas de l'utilisation de graphes de types d'attributs présentée dans la partie précédente, la classification d'une classe [Capp93] peut s'opérer à un niveau de granularité de connaissances moindre : les attributs. En effet, à partir de la description d'une nouvelle classe C, deux fonctions, **sup(a, C)** et **inf(a, C)**, permettent de trouver respectivement, l'ensemble des classes de la hiérarchie dont la description de l'attribut **a** correspond à un sur-type minimal de **a** dans **C**, et l'ensemble des classes dont la description de l'attribut **a** correspond à un sous-type maximal de celui de **a** dans **C**. Ces deux calculs coïncident avec le placement du type de **a** de la classe **C** dans la hiérarchie des types de **a**. Cette méthode appliquée à tous les attributs de la classe C permet deux nouveaux calculs, **sup(C)** et **inf(C)**, qui décrivent respectivement toutes les sur-classes minimales possibles, et toutes les sous-classes maximales possibles.

Ainsi décomposé, le système à base de type, augmenté d'une interface de visualisation des graphes de types d'attribut, offre des outils interactifs appréciables pour assister l'utilisateur lors de la création de nouvelles classes.

3.2. Raisonnement entre instances et classes

La classification d'instance repose sur la confrontation des informations portées par l'instance avec celles contenues dans les descriptions de classes. Cette confrontation est appelée appariement d'une instance à une classe.

3.2.1. Appariement d'une instance à une classe

L'appariement d'une instance à une classe est déclenché lorsqu'il y a rattachement explicite à la classe, lors de la classification, ou lors d'une modification de l'instance (ajout, suppression ou modification d'une valeur d'attribut).

Pour déduire l'appartenance possible de l'instance I à une classe C, les valeurs d'attributs de l'instance sont comparées aux descriptions des attributs de la classe. Il correspond donc à la mise en œuvre du prédicat Δ_C présenté en §II.2.3.

De plus, ce mécanisme utilise l'héritage entre classes afin de déterminer pour chaque attribut son domaine dans la classe à apparier. Ce processus, comme le montre la définition de Δ_C prenant en compte l'héritage, établit par la même occasion l'appartenance possible aux sur-classes de C.

Il est à noter que le système de type présenté précédemment permet par le travail statique effectué de ramener le prédicat Δ_C à une vérification de type. Toutefois, quand le formalisme permet la mise en place de conditions dynamiques (code procédural, contraintes, etc.) dans une classe, la simple vérification de types ne suffit pas ; il faut aussi vérifier ces conditions.

Par ailleurs, il est particulièrement intéressant qu'un appariement fournisse des informations visant à expliquer son résultat. En effet, lorsque l'appariement rend "possible" ou "impossible", ce résultat peut être exprimé en termes de différence entre les informations contenues dans l'instance et les descriptions de la classe concernée.

3.2.2. Classification d'instance

La classification d'une instance dans une hiérarchie de classes est un mécanisme d'inférence qui cherche à trouver le degré d'appartenance d'une instance à chaque classe de la hiérarchie. Le résultat attendu de cette classification est d'identifier la (ou les) classe la plus spécifique d'une hiérarchie de spécialisation à laquelle l'instance peut, ou pourrait être rattachée étant données sa structure et ses valeurs d'attributs.

Pour ce faire, la classification s'appuie sur deux niveaux de relation : instance-classe, et classe-classe. Le traitement au premier niveau, relation instance-classe, correspond à l'appariement de l'instance à chaque classe visitée lors de la classification. L'appariement considéré dans cette description du mécanisme de classification est supposé rendre l'une des trois valeurs suivantes : "sûr", "possible" ou "impossible" (cf. §II.2.3). Par ailleurs, les relations du second niveau, classe-classe, sont exploitées pour contrôler l'exploration de la hiérarchie de spécialisation. Ce contrôle pourra donner lieu à deux types d'opérations : le raffinement consistant à sélectionner de nouvelles classes à apparier, ou la propagation du résultat d'un appariement à travers la hiérarchie de spécialisation de classes. L'exemple de la classification d'une instance de poisson (une baleine, pour être correct), nommée *Wanda*, dans la hiérarchie des animaux (cf. Figure II.6) illustre les différentes phases : appariement, raffinement et propagation. L'appariement ayant déjà été présenté dans la partie précédente, seules les deux autres phases sont décrites dans ce qui suit.

La phase de **raffinement** intervient quand une classe C vient de passer avec succès la phase d'appariement, statut "sûr". Elle a pour but de propager l'appariement de l'instance I considérée sur les sous-classes de C. Cette phase de raisonnement s'appuie sur l'ordre partiel fourni par la hiérarchie de spécialisation, et peut être résumée par la règle :

- **si** $\Delta_C(I)$="sûr" **et** ($\exists$ C', C $\leftarrow^1$ C') **alors** (apparier I a C')

qui s'exprime en termes ensemblistes par :

- **si** (I $\in$ Ext(C)) **et** ($\forall$ J, J $\in$ Ext(C') $\Rightarrow$ J $\in$ Ext(C)) **alors** (soumettre I $\in$ Ext(C'))

Cette dernière règle souligne l'aspect abductif du contrôle d'exploration de la hiérarchie par la phase de raffinement.

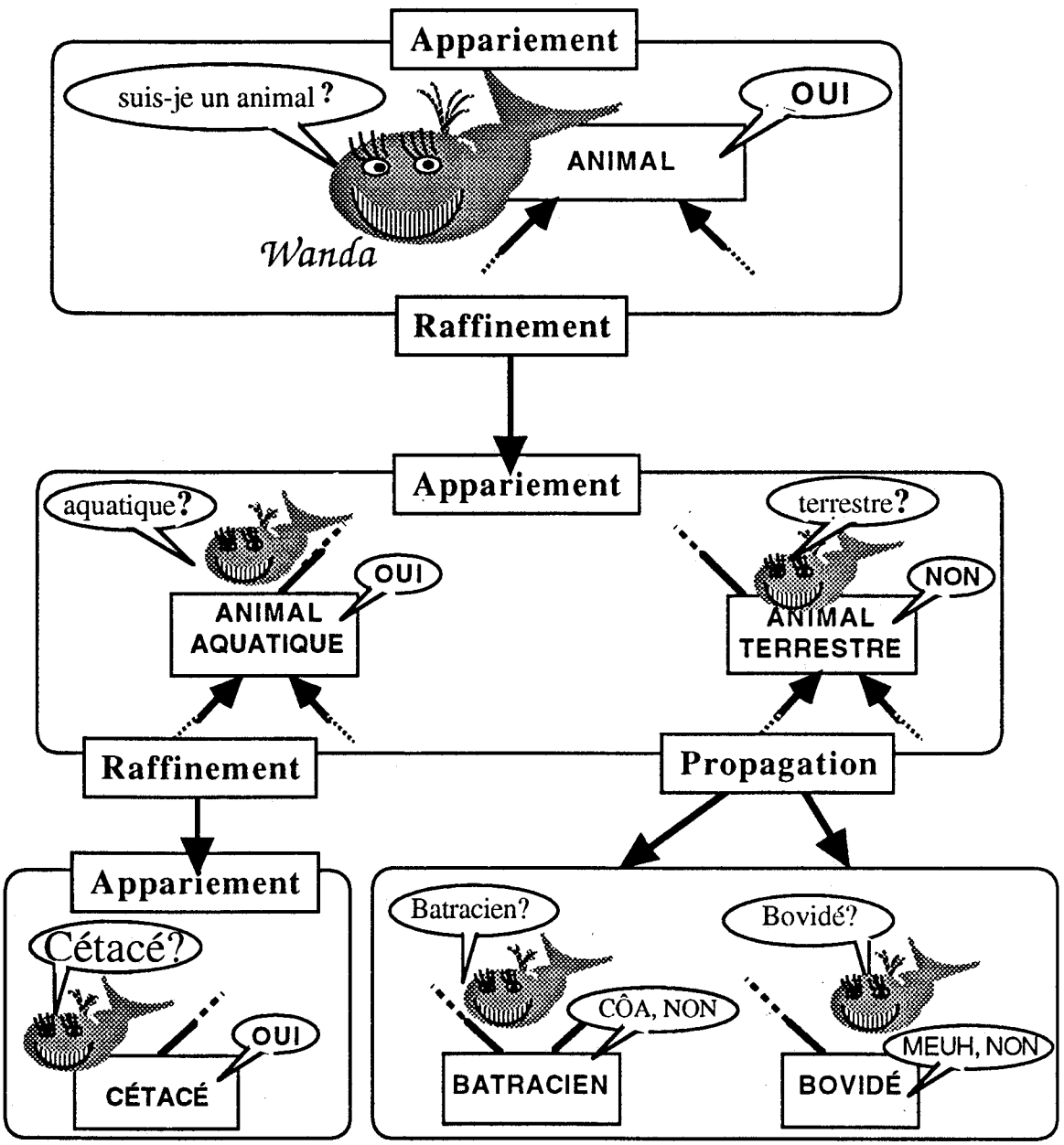

Figure II.6 : Exemple de la classification de l'instance de baleine *Wanda* dans la hiérarchie décrivant les animaux. Au départ, l'instance est appariée à la classe ANIMAL. Le résultat "sûr" (symbolisé par la réponse "OUI"), permet à la phase de raffinement de propager l'appariement aux sous-classes ANIMAL-AQUATIQUE et ANIMAL-TERRESTRE. Pour cette dernière classe, l'appariement est négatif (résultat "impossible"), la phase de propagation prend place pour propager ce résultat aux sous-classes BATRACIEN et BOVIDE. Enfin, le raffinement de la classe ANIMAL-AQUATIQUE déclenche l'appariement de l'instance sur la classe CETACE, "cétacé ?" dit la baleine, le réponse étant "oui" la classification s'achève sur cette classe la plus spécialisée.

La phase de **propagation** permet d'obtenir le statut de certaines classes par les déductions qui peuvent être menées à partir d'un résultat d'appariement et les liens existant entre classes. Son propos est de réduire le graphe d'exploration de la classification et d'éviter ainsi des appariements de classe inutiles. Le principe est que les sous-classes d'une classe déclarée "impossible" sont elles-mêmes déclarées impossibles. La prise en compte de la règle duale

propageant la marque "sûre" aux sur-classes d'une classe déclarée "sûre" est aussi intéressante quand la classification de l'instance ne commence pas à la racine ou en cas de spécialisation multiple, sinon elle n'a pas lieu d'être car les sur-classes ont déjà été marquées "sûres" lors de la descente. Les règles appliquées sont donc :

- **si** $\Delta_C(I)$="impossible" **alors** ($\forall$ C', C $\leftarrow$ C') (déclarer C' "impossible")
- **si** $\Delta_C(I)$="sûr" **alors** ($\forall$ C', C $\rightarrow$ C') (déclarer C' "sûr")

En ce qui concerne le cas d'une classe déclarée "possible", il est impossible de statuer de façon certaine sur ses sous-classes. La seule indication donnée est que ces sous-classes ne peuvent pas être "sûres". La propagation de la marque "possible" sur les sous-classes peut être adoptée pour éviter une descente qui ne permettra plus de trouver de classes "sûres". Dans ce cas, "possible" signifie que la classe n'a pas été explorée par la classification.

Ce schéma général de classification d'instance est adopté par le système SHIRKA [Rech&89] ; l'instance ne sera rattachée à une des classes déclarées "sûres" que sous la directive de l'utilisateur. Par contre, le modèle TROPES, tel qu'il a été proposé par Olga Mariño [Mari93], assure de plus que l'appartenance d'une instance à une classe est établie quand l'appariement rend "sûr" (les descriptions de classe forment des conditions nécessaires et suffisantes), c'est pourquoi le rattachement de l'instance suit l'évolution de la classification : l'instance est rattachée aux classes "sûres" les plus spécialisées à chaque cycle de la classification. De plus, la représentation des objets est enrichie des notions de point de vue et de passerelle entre classes qui permettent une phase de propagation plus riche.

La mise en place récente d'un mécanisme de classification dans le système FROME [Dekk94] (cf. §IV.4.4.2) suit le même principe général que SHIRKA et TROPES. La possibilité de distinguer des attributs qui participent à la description de la classe en tant que conditions nécessaires et suffisantes d'appartenance, et ceux qui n'ont qu'un rôle de conditions nécessaires, offre une solution de compromis entre celles de SHIRKA et de TROPES pour traiter le rattachement d'une instance au cours de la classification. Le modèle SHOOD [Liot93] (cf. §IV.4.2.1) fait aussi une distinction entre divers types d'attributs, mais le résultat de l'appariement peut être uniquement "possible" ou "impossible".

En ce qui concerne les langages terminologiques, l'approche adoptée consiste généralement à ramener la classification d'instance à un cas de classification de *concept* (le système LOOM est l'une des rares exceptions à cette règle) (cf. §I.3.2.3). Une première phase d'abstraction consiste alors à transformer l'instance en un *concept*, le mécanisme d'appariement utilisé est celui entre *concepts*. Cette technique correspond à la structure d'inférence *abstraction-appariement-raffinement* de la **classification heuristique** proposée par Clancey [Clan85].

3.3. Inférences de valeurs d'attributs

En dehors des calculs de types d'attribut ou de vérification de l'appartenance d'une valeur d'attribut à un domaine particulier, un système à objets fournit généralement des mécanismes d'inférence permettant l'évaluation d'une valeur d'attribut dans le contexte particulier d'une instance. Schématiquement, ce genre de mécanisme est déclenché lorsque l'attribut, ou du moins sa valeur, est déclaré inconnu, et que l'instance vérifie une certaine condition. Le mécanisme exploite alors les connaissances de l'instance pour établir la valeur de cet attribut. La règle de déclenchement d'une inférence φ sous la condition α de l'attribut A de I peut s'exprimer de la façon suivante :

- **si** A(I)=? **et** α(I) **alors** A(I) = φ(I) ;

  où A(I)=? indique que l'attribut A est à valeur inconnue dans I.

Dans la suite, les différents types d'inférence sont étudiés en fonction des connaissances employées.

3.3.1. Les inférences triviales

Une telle inférence apparaît quand le domaine D de l'attribut A à inférer, donné par la description $\delta(A)$, est réduit à une seule valeur : $D=\{v\}$. On a donc

- **si** $A(I)=?$ **et** $(\delta(A)(I) \Leftrightarrow A(I) \in \{v\})$ **alors** $A(I)=v$

L'inférence est évidente et rend v comme valeur de A(I).

La réduction du domaine d'un attribut à la valeur v s'obtient généralement par l'ajout d'une facette (**$valeur** v) au niveau de la description de l'attribut dans une classe. Dans ce cas, la condition α exprime la condition d'appartenance à la classe : $(I \in C)$.

3.3.2. Inférences par dépendances d'attributs

Ce style d'inférences peut être effectué lorsque l'attribut à inférer A est dépendant d'autres attributs $A_1,\ldots A_n$, suivant une fonction $f :(D_1\times \ldots\times D_n) \rightarrow D$ telle que :

- D1, ..., Dn, D sont les domaines respectifs des attributs A1,... An, et A.
- **si** $A(I)=?$ **et** $\alpha(I)$ **alors** $A(I) = \varphi(I) = f(A_1(I), \ldots A_n(I))$

Dans ce cas, la connaissance des valeurs $A_1(I), \ldots A_n(I)$ permet de déterminer A(I).

Différents types de mécanismes peuvent mettre en place une telle dépendance entre attributs :

- $\alpha(I) \Rightarrow (I \in \mathbf{Ext(C)})$: la condition α sur I inclut l'appartenance à la classe C.

  C'est un **moyen local** (à la classe) **d'évaluation** de l'attribut.

  - **Attachement procédural** : une procédure dont le code permet d'effectuer le calcul f est attachée à la description de l'attribut A par une facette **$sib-exec** <proc-f>. Le rattachement de l'instance I déclenche son exécution lorsque A est inconnu.
  - **Contraintes inter-attributs** : la description de la classe C contient un ensemble de contraintes impliquant les attributs $A_1,\ldots A_n$, et A. Pour appartenir à la classe une instance doit nécessairement vérifier cette contrainte. Quand l'instance est rattachée à cette classe, les contraintes peuvent être appliquées pour déduire les valeurs d'attributs manquantes. Dans ce cas, $A(I) = f(A_1(I), \ldots A_n(I))$ est une conséquence de ces contraintes.

- $\alpha(I) \not\Rightarrow (I \in \mathbf{Ext(C)})$: la condition α n'exprime pas spécialement un rattachement de I à une classe.

  L'inférence n'est pas locale à la description d'une classe. Contrairement à l'attachement procédural, c'est un **moyen global d'évaluation** de l'attribut (cf. §II.3.4.3). La condition α est alors un prédicat exprimant un ensemble de propriétés à satisfaire par certains attributs de l'instance, indépendamment de sa classe de rattachement.

Il est à noter que dans les langages qui combinent un formalisme de règle et un formalisme d'objet, il est possible d'exprimer directement sous forme de règle les moyens globaux d'évaluations d'attribut. La forme générale des règles permet de fournir une condition, dans ce cas, qui n'implique pas l'instance. Ce formalisme trop général ne prend pas en compte la structuration objet et ne fait donc pas partie du cadre étudié ici.

Par la suite, les différents types de mécanismes d'inférence d'attribut par dépendance d'attributs sont rapidement présentés et resitués par rapport aux systèmes à objets.

3.4. Expression de dépendance d'attributs

L'attachement procédural est une fonctionnalité importante et courante des langages de *frames*. D'ailleurs, l'idée était déjà suggérée dans l'article de M. Minsky [Mins75]. La notion de contraintes entre attributs au sein d'une classe est moins courante, elle demande la mise en

place d'un mécanisme spécialisé dans la gestion de contraintes. Enfin, le moyen global d'évaluation est une solution proposée pour introduire de façon propre des calculs d'attribut dans un système qui fait de la classification d'instance. Ce mécanisme est aussi nommé ironiquement "détachement" procédural par opposition au principe de l'attachement de procédure à une classe.

3.4.1. Attachement procédural

L'attachement procédural permet d'exprimer le calcul d'un attribut dans une classe, en lui associant une procédure. Cette dernière est alors désignée par une facette **$sib-exec** (signifiant "si besoin exécuter"). L'exemple de schéma de classe SHIRKA suivant illustre l'utilisation d'un attachement procédural :

Exemple du calcul de l'attribut *age* en fonction de l'attribut *année-naissance* :

```
{ Personne
   sorte-de =  objet ;
   nom              $un          chaine ;
   annee-naissance  $un          entier ;
                    $var-nom     an-origine ;
   age              $un          entier ;
                    $sib-exec    {  Difference
                                       op1   $valeur  1994 ;
                                       op2   $var<-   an-origine ;
                                       res   $var->   age ;  } ;
...}
```

Dans cet exemple, *Difference* est le nom de la procédure à exécuter. Elle prend en paramètres d'entrée (exprimés à l'aide des facettes dédiées $var-nom (renommage), $var<- (entrée)) la valeur 1994 et la valeur de l'attribut *annee-naissance* et rend en sortie (facette $var->) la valeur de l'attribut *age*.

Dans SHIRKA, l'interfaçage entre objet et code (Le-Lisp) se fait par le biais d'une déclaration de classe d'un type spécial : **méthode**[3]. Ainsi *Difference* est un objet de type **méthode** dont les paramètres sont définis comme des attributs. Un attribut de cette classe désigne la fonction Le-Lisp à appeler. L'exécution en est simple : il suffit de demander la création d'une instance en fournissant les valeurs des attributs d'entrée ; la fonction Le-Lisp correspondante est alors appelée avec l'instance de méthode en paramètres ; enfin elle rend cette même instance augmentée du résultat. Cet interfaçage à l'aide d'objet **méthode** permet de garder une trace des exécutions de **méthode** effectuées.

D'autres systèmes ne proposent pas d'interfaçage entre objet et procédure : FROME utilise une lambda-expression Lisp, YAFOOL indique juste le nom de la procédure Lisp qui prendra alors en paramètre l'instance qui l'a déclenchée, etc.

3.4.2. Introduction de contraintes dans une classe

Un langage de contraintes offre un formalisme adéquat pour exprimer et supporter des dépendances complexes entre attributs.

L'exemple suivant, tiré de [Born81], montre comment les contraintes sont introduites dans une classe du système THINGLAB :

[3]Ce type n'a pas de rapport avec la notion de *méthode* des langages de programmation par objets; l'appellation *procédure* serait peut-être plus opportune.

Exemple de contraintes dans THINGLAB :

objectif : définir la contrainte liant les deux extrémités d'une droite avec son milieu

```
Class MidPointLine
   Superclasses Geometric object
   Part Descriptions   Line :          a line
                       midpoint :     a point
   Constraints         midpoint = (line point1 + line point2)/2     ; Prédicat
                       midpoint <- (line point1 + line point2)/2    ; Méthode1
                       line point1 <- midpoint * 2 - line point2    ; Méthode2
                       line point2 <- midpoint * 2 - line point1    ; Méthode3
```

Comme le montre cet exemple, l'expression des contraintes de THINGLAB nécessite la donnée du prédicat à vérifier ainsi que la donnée de l'ensemble de méthodes nécessaires pour satisfaire la contrainte quand la valeur d'un attribut est manquante.

THINGLAB offre une approche originale de la description d'une classe puisqu'elle s'enrichit d'un nouveau niveau de définition, celui des définitions de contraintes entre attributs. L'expression des contraintes est, par contre, rudimentaire puisqu'elle oblige à indiquer comment traiter tous les cas d'utilisation d'une contrainte (prédicat et les méthodes de calcul).

Une intégration plus profonde consiste à coupler le modèle à objets avec un module de gestion de contraintes de type CSP [Gens93]. L'expression d'une contrainte dans le modèle est alors interprétée en termes d'un réseau de contraintes exploitable directement au niveau du module de gestion par les techniques de résolution de contrainte. Une telle extension d'un système de représentation par objets augmente à la fois sa puissance d'expression et sa capacité d'inférence (cf. §VI).

3.4.3. "Détachement" procédural ou moyen global d'évaluation

L'évaluation d'un attribut par un moyen global est proposé dans [Rech92] pour remplacer l'attachement procédural qui pose trois types de problème dans un système offrant un mécanisme de classification d'instance :

- **évolution des objets** : les informations portées par une instance devraient être conservées lorsqu'elle migre d'une classe à une autre. Notamment, quand une instance est classée et quelle passe d'une classe à une sous-classe, les attachements procéduraux de chacune d'elles devraient rendre les mêmes résultats quand ils s'appliquent aux mêmes attributs. Les calculs devraient être indépendants des classes.
- **déclenchement d'inférence** : il serait paradoxal d'utiliser lors de la classification d'une instance les attachements procéduraux d'une classe alors que l'appartenance de l'instance à cette classe ne peut être affirmée du fait des attributs manquants. Seule la reconnaissance de l'appartenance de l'instance à une classe peut permettre l'utilisation d'un de ses attachements procéduraux.
- **influence des attachements procéduraux sur la hiérarchie** : des classes sont trop souvent introduites dans les hiérarchies pour définir les conditions sous lesquelles doivent être effectués certains calculs.

Suite à ces constats, l'idée proposée est de supprimer les attachements procéduraux de la description des classes (cf. Figure II.7) pour exprimer la hiérarchie de spécialisation indépendamment des éventuels calculs d'attributs. Les méthodes de calcul d'attribut sont déclarées globalement et indexées par l'attribut dont elles permettent l'évaluation.

Ainsi, à chaque attribut peut être associé un ensemble de méthodes de calcul organisé de façon à faciliter la sélection de l'une d'entre elles lorsqu'une instance nécessite le calcul de cet attribut. L'organisation s'appuie sur les descriptions des paramètres d'entrées qui déterminent le critère de choix d'une méthode, et peut être exprimée avantageusement à l'aide d'une hiérarchie de méthodes.

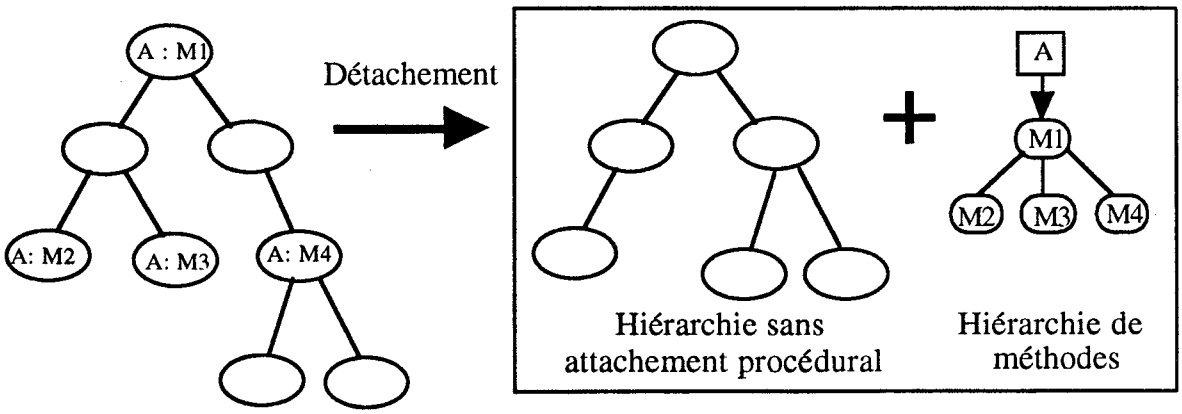

Figure II.7 : Principe de délocalisation de la connaissance procédurale attachée à un attribut dans les classes. Les méthodes de calcul d'un attribut sont déclarées à l'extérieur de la hiérarchie d'objets qui peut s'en trouver "allégée", et sont organisées en hiérarchies en fonction de leur critère d'application, leurs paramètres d'entrée.

Cette approche demande néanmoins une petite précision quant au choix du moment de déclenchement des méthodes de calcul d'un attribut. Ces moyens d'évaluation étant globaux, il est tout à fait envisageable de lancer tous les calculs d'attributs inconnus dès la création d'une instance, et cela jusqu'à ne plus pouvoir exécuter aucune méthode, soit parce que l'instance est complétée, soit parce que les attributs restant sans valeur ne sont pas calculables. Cette pratique de "calcul à l'aveuglette" ne tient pas compte des besoins limités en attributs d'un raisonnement : des calculs peuvent être inutiles car certains attributs ne sont pas significatifs dans le contexte d'un raisonnement particulier. Une évaluation paresseuse, proposée pour le déclenchement du calcul d'un attribut, semble alors être le bon compromis.

Enfin, une solution différente peut être envisagée pour résoudre le problème des calculs d'attributs lors d'un processus de classification. Comme le permet le système FROME [Dekk94], des rôles différents peuvent être associés aux attributs dans la description d'une classe, certains seront déclarés déterminants pour la classification et d'autres non. Dans ce cas, les attributs non déterminants d'une classe peuvent posséder un attachement procédural et être déterminé lors de la classification. Le moyen d'évaluation des attributs reste alors local.

4. Le modèle à objets TROPES

Le modèle TROPES[4] [Mari89][Mari91][Mari93] est né d'une étude sur le raisonnement classificatoire dans le cadre des représentations par objets qui distinguent la notion de classe de celle d'instance. C'est-à-dire que l'approche est ensembliste.

Ce premier noyau à orientation classificatoire prononcée introduit un certain nombre d'idées originales qui le distingue nettement d'un langage de *frames* classique (cf. §II.4.1). Parmi celles-ci, l'indépendance de la notion d'instance par rapport à celle de classes est primordiale ; elle prend son essence dans la notion de **concept** qui est introduite pour définir l'espace d'existence d'une instance. La notion de **point de vue** est, quant à elle, proposée pour offrir une description d'un même ensemble d'objets selon plusieurs hiérarchies de classes (§II.4.1.1).

Après avoir discuté de cette nouvelle optique de la notion d'objet, donné la description d'une base sous le format TROPES, le mécanisme de classification est présenté (§II.4.2). Enfin, cette description de la première version du modèle TROPES s'achève sur les différentes directions d'extension actuellement suivies (cf. §II.4.3).

---

[4]Le modèle TROPES, présenté dans cette partie et auquel il sera fait référence dans la suite du document, est celui proposé originellement par Olga Mariño. Cette présentation ne prend pas en compte les généralisations du modèle proposées par Jérôme Euzenat dans [Euze93a] et [Euze93b].

4.1. Répartition des connaissances

Un résultat important du travail mené sur le modèle TROPES est la remise en cause de la répartition de la connaissance telle qu'elle est habituellement admise dans une représentation par objets. En effet, l'importance accordée à la notion de classe et, plus généralement, à la notion de hiérarchie de classes est nettement réduite.

L'approche adoptée dans TROPES est motivée par l'évolution possible d'une instance dans une hiérarchie de classes. Un ensemble d'instances est circonscrit par la notion de **concept** qui permet de distinguer une famille particulière d'individus, de fournir l'espace d'attributs auxquels une instance de ce concept peut prétendre, et de décrire différentes décompositions ensemblistes sous la forme de hiérarchies de classes constituant l'espace de travail de la classification d'instance. Ainsi, la gestion et la définition des instances sont reléguées au niveau du concept.

Dans [Euze93b], une synthèse formelle reprend ce résultat en associant à un objet un aspect **ontologique** et un aspect **taxinomique**. Ces aspects traduisent la volonté de distinguer pour un objet ce qui lui donne droit à l'existence (création d'une instance dans le concept) de ce qui permet de le catégoriser (classification d'une instance dans les hiérarchies de classes d'un concept).

4.1.1. Description d'une base

Une base TROPES est constituée d'un ensemble de concepts dont les extensions sont disjointes deux à deux (cf. Figure II.8). Chacun de ces concepts représente une famille particulière d'objets, c'est-à-dire un ensemble potentiel d'instances, et est défini par un ensemble de descriptions d'attribut, et un ensemble de points de vue.

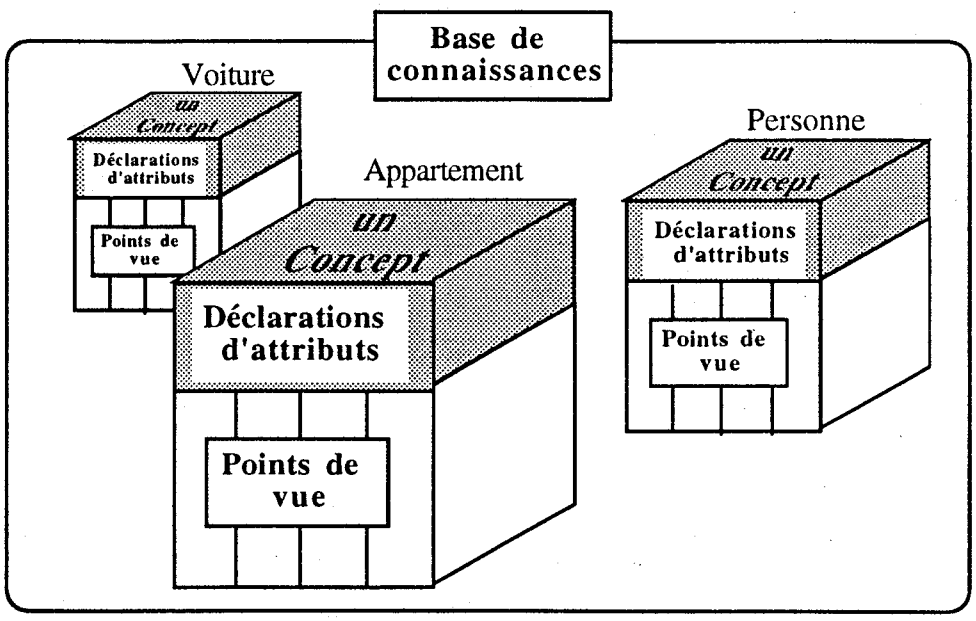

Figure II.8 : Partition de la base en concepts. Chacun d'eux représente une famille d'objets particuliers : *Voiture*, *Appartement*, *Personne*, etc. Un concept définit, d'une part, l'ensemble des attributs possibles d'un objet de la famille et leur description minimale et, d'autre part, les différents points de vue (hiérarchies de classes) à partir desquels une instance du concept peut être perçue.

L'ensemble de descriptions d'attributs fournit à une instance du concept l'espace des attributs qu'elle est susceptible d'inclure dans sa structure. Ces descriptions d'attributs incluent :

- son **type**, qui peut être un type de base (entier, réel, chaîne, booléen), ou un concept de la base.
- la **nature** de l'attribut :
  - **propriété** : un tel attribut traduit un caractère propre à l'objet. Par exemple, les attributs *couleur*, *age*, *date*, *taille*,... sont de nature **propriété**.
  - **composant** : l'attribut exprime un lien avec un composant de l'objet. Par exemple, les attributs *bras*, *carrosserie*, ... sont de nature **composant** et peuvent intervenir respectivement dans la composition des objets des concepts HUMAIN, et VOITURE. L'objet qui contient des attributs de nature **composant** est dit objet **composite**, la valeur des attributs composants sont des instances de la base appelées elles-mêmes **composants**. Le lien qui unit l'objet composite à ses composants est existentiel et exclusif : le composant est nécessaire pour que l'objet existe, sinon sa composition est incomplète et, en retour, les composants n'appartiennent qu'à un et un seul objet composite.
- le **constructeur** à appliquer à l'attribut :
  - **un** : l'attribut ne prend qu'une valeur du type <type>.
  - **liste-de :** l'attribut est une liste de valeurs du type <type>.
  - **ensemble-de** : l'attribut est un ensemble de valeurs du type <type>.

A chaque point de vue d'un concept correspond une hiérarchie de classes organisée par la relation de spécialisation (cf. Figure II.9). Un point de vue décrit une classification possible de l'ensemble des individus du concept. Tous les points de vue partagent au minimum la même classe racine qui contient l'ensemble de toutes les instances acceptables dans le concept. Une instance est donc liée à chaque point de vue par la classe à laquelle elle appartient dans ce point de vue (au minimum la classe racine).

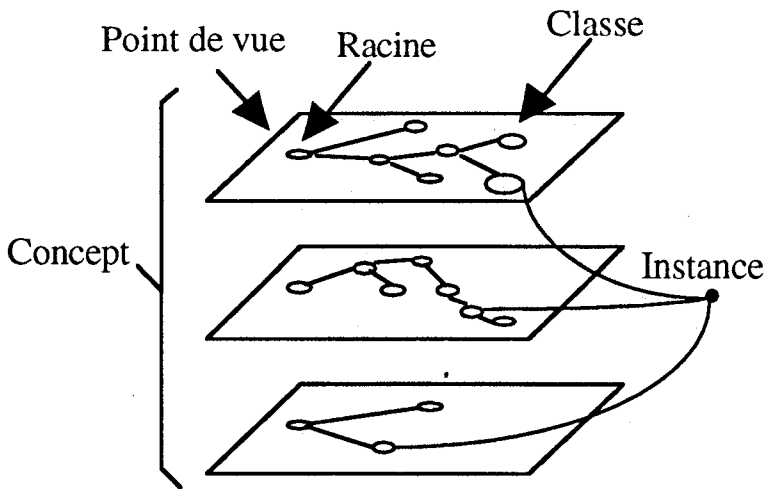

Figure II.9 : un concept de TROPES est visible selon plusieurs points de vue. Chaque point de vue est représenté par une hiérarchie de spécialisation de classes. La classe racine représente le même ensemble d'instances dans chaque point de vue : l'ensemble de toutes les instances du concept. Une instance du concept est liée à chaque point de vue par son lien d'appartenance à une, et une seule, classe de ce point de vue.

En outre, deux points de vue peuvent être mis en relation par la notion de passerelle qui traduit l'appartenance de toute instance d'une classe d'un point de vue dans une classe de l'autre point de vue. Cette notion de passerelle est généralisable à plusieurs points de vue.

Une classe représente un ensemble d'instances, l'extension de la classe. Cet ensemble est délimité par la donnée de la description de la classe, l'intension de la classe, qui forme un ensemble de conditions nécessaires et suffisantes d'appartenance à la classe. Cette intension correspond à la donnée des attributs du concept caractérisant n'importe quelle instance de cette classe, et d'un ensemble de conditions sur les domaines de valeurs de ces attributs. Ces conditions sont exprimées à l'aide de descripteurs de sous-typage, de restriction de domaine (par énumération ou exclusion) ou de restriction de cardinalité si l'attribut est de type liste ou ensemble.

4.1.2. Propriétés des hiérarchies de classes

La hiérarchie de classes que décrit un point de vue d'un concept possède les propriétés suivantes :

- La relation de spécialisation entre classe est stricte (lien **sorte-de**) ; c'est-à-dire que l'extension (l'ensemble des instances) d'une classe inclut toutes les extensions des classes qui en sont des spécialisations.
- Il y a exclusivité des sous-classes directes d'une classe, c'est-à-dire que les extensions des sous-classes sœurs sont disjointes deux à deux (deux sous-classes directes d'une classe sont dites sœurs). Une instance ne peut appartenir à deux classes qui ne sont pas liées, directement ou indirectement, par la relation de spécialisation.
- La description d'une classe constitue l'ensemble des conditions nécessaires et suffisantes d'appartenance à la classe. Cette contrainte est imposée pour la classification.

Les conséquences de ces propriétés sont que la hiérarchie de classes associée à un point de vue est un arbre, et que la multi-instanciation et le multi-héritage sont impossibles dans un même point de vue. Une instance est donc attachée à une et une seule classe dans chaque point de vue par le lien **est-un** et appartient à chaque classe située sur le chemin de cette classe à la racine.

4.2. Exploitation du modèle

La notion d'instance constitue l'interface entre l'utilisateur et la base de connaissances. A travers une instance, un utilisateur décrit la configuration de données initiales qu'il désire confronter à la description d'une base de connaissances.

Nécessairement créée dans un concept précis de la base, une instance y trouve à la fois ses limites et ses possibilités d'évolution. Pour ce faire, les données inscrites au sein de l'instance par l'utilisateur sont exploitées suivant leur nature : celles relatives à l'appartenance de l'instance à des classes (lien **est-un**) et celles concernant les caractéristiques propres de l'instance (attributs et valeurs d'attributs).

Par la suite, l'exploitation du modèle est présentée à travers les opérations de configuration d'une instance (création ou modification d'une instance) et le mécanisme de classification d'instance.

4.2.1. Configuration d'instance

> "La notion d'objet consiste avant tout en la création de voies de communication entre un utilisateur raisonnant en termes d'entités du monde réel, et les structures de données utilisées par un programme." [Rous88]

L'instance est un tout logique dans lequel est regroupée la connaissance concernant l'individu particulier auquel s'intéresse un utilisateur. Par le biais des informations initiales qu'il fournit sur une instance, l'utilisateur entame une coopération avec le système de représentation dans le but d'en compléter la connaissance.

La configuration d'une instance correspond donc à l'intervention de l'utilisateur pour créer une nouvelle instance ou modifier une instance existante. La création d'une instance est basée sur la donnée des spécifications suivantes :

- son concept,
- la classe à laquelle elle doit être rattachée dans chaque point de vue du concept (la classe racine par défaut),
- les paires (attribut, valeur) représentant la connaissance que l'utilisateur a des attributs de l'instance.

Lors de la configuration d'une instance, le système se charge de vérifier l'adéquation des données acquises en effectuant les appariements entre l'instance et les classes de rattachement fournies. L'ensemble des données doit être consistant sinon la configuration échoue : l'instance inconsistante ne peut être utilisée par les mécanismes de raisonnement.

La modification, ou reconfiguration, d'une instance consiste à changer les classes de rattachement, ou/et modifier des paires (attribut, valeur), le concept étant, quant à lui, irrévocable. Cette action dénote une utilisation non monotone de la base de connaissances nécessitant éventuellement un réajustement des connaissances déduites dans l'instance. Ceci est assuré par les mécanismes de maintien du raisonnement (relocalisation d'instance, et propagation des valeurs modifiées à travers un réseau de dépendances entre attributs (cf. III).

4.2.2. Classification d'instance

La classification d'instance est un mécanisme qui consiste à confronter la connaissance présente dans l'instance aux différentes hiérarchies (points de vue) de classes du concept. Basé sur un principe de parcours descendant de la hiérarchie de classes associée à un point de vue, ce mécanisme permet d'en diviser l'ensemble de classes en trois catégories reflétant le résultat de leur appariement avec l'instance :

- les classes **sûres** auxquelles l'instance appartient,
- les classes **impossibles** dont les conditions d'appartenance sont contredites par l'information portée par l'instance,
- les classes **possibles** dont l'appartenance n'a pu ni être prouvée, ni être réfutée en raison des informations insuffisantes de l'instance.

Dans TROPES, l'algorithme de classification proposé par Olga Mariño permet de descendre l'instance dans une hiérarchie de classes de façon incrémentale, en tirant parti des propriétés de la hiérarchie (cf. Figure II.10). Un point de vue est initialement choisi pour commencer le processus de classification.

```
construire_état_initial     ; configuration de l'instance
tant_que non (fini)         ; boucle tant que des points de vue
                            ; peuvent encore être explorés faire
  obtenir_information       ; demande d'information manquante
  faire_appariement         ; raffinement et appariement des
                            ; sous-classes
  propager_marques          ; propagation des marques à travers les
                            ; points de vue et les passerelles
  mettre-à-jour-l-information ; synthétiser l'information provenant
                              ; de chaque classe de rattachement
  choix_prochain_point_de_vue ; choix du prochain point
                              ; de vue à explorer
fin_tant_que
```

figure II.10 : Algorithme de classification multi-points de vue de TROPES proposé dans [Mari93].

A chaque étape, une phase de raffinement cherche à spécialiser l'instance dans les sous-classes directes de la classe de rattachement courante. Dès qu'une de ces sous-classes est jugée

sûre, l'instance y est rattachée ; les autres sous-classes sont alors marquées impossibles par application du principe d'exclusivité des classes sœurs (cf. §II.4.1.2).

Une phase de propagation est chargée de répercuter les nouvelles marques obtenues par les passerelles existantes entre les points de vue. Cette propagation tient compte de la sémantique des passerelles pour déduire de nouveaux marquages de classes dans les autres points de vue.

L'exploration d'un point de vue s'achève lorsque le raffinement ne permet pas de déterminer une nouvelle classe sûre ; les sous-classes sont alors soit impossibles, soit possibles. L'algorithme passe alors à un autre point de vue et recommence la phase d'exploration à partir de sa classe sûre la plus spécifique. La classification se termine quand tous les points de vue ont été entièrement explorés. L'instance est alors attachée à la plus petite classe sûre de chaque point de vue.

Durant la progression de l'instance dans la hiérarchie de classes, des demandes d'attributs peuvent être déclenchées. Cet enrichissement d'information sur une instance intervient lorsque la phase de raffinement n'obtient pas de nouvelle classe sûre mais au moins une classe possible. Dans ce cas, les descriptions des classes possibles sont utilisées pour centraliser les demandes d'attributs (un attribut n'est demandé qu'une seule fois), et les organiser de façon à optimiser la recherche d'une sous-classe sûre, et/ou des sous-classes impossibles. Si l'instance a été suffisamment complétée lors de la phase d'obtention d'attribut, la classification peut reprendre sa descente dans la hiérarchie, sinon elle abandonne ce point de vue.

4.3. Extensions du modèle

La première version de TROPES qui vient d'être présentée se focalise essentiellement sur les possibilités de classification multi-points de vue. Toutefois, elle a été aussi prévue pour accueillir un mécanisme d'inférence de valeurs d'attributs par un moyen global. En effet, la classification ne pouvant bénéficier des attachements procéduraux inscrits dans les classes, une première étude [Orsi90] sur la délocalisation des procédures propose d'introduire ce type de connaissances dans la définition des attributs au niveau du concept. La classification peut donc bénéficier de cette information (cf. §II.3.4.3), il suffit de le prévoir dans la phase de demande de valeur.

De plus, l'algorithme de classification proposé se base sur une normalisation des domaines de valeurs d'attribut pour mettre en place l'appariement d'une instance à une classe, mais aussi pour regrouper dynamiquement les informations que les rattachements aux classes de chaque point de vue apportent à une instance. Cette normalisation est à la base du travail sur le typage des attributs et l'étude d'un mécanisme de classification de classes [Capp94] qui sont en cours de réalisation dans la nouvelle version de TROPES. Outre les avantages déjà reconnus du typage d'attributs, d'autres propositions prometteuses émergent de ce nouveau système de typage : factorisation des graphes de typage en fusionnant les graphes des attributs qui possèdent le même type de base, la possibilité de définir de nouveaux types de base dans le modèle, etc.

L'étude sur la parallélisation du modèle TROPES, soit en le plaquant sur une architecture de "blackboard", soit en utilisant un modèle d'acteurs, a permis de mettre au point une version parallèle de l'algorithme de classification [Quin93].

Enfin, l'intégration de la notion de contrainte [Gens93] dans la description des classes permet d'augmenter considérablement le pouvoir d'expression du modèle. Cette notion cadre parfaitement avec l'esprit classificatoire originel du modèle (cf. §VI.2).

5. Conclusion

La représentation par objets est un domaine conceptuellement riche qui peut paraître assez hétérogène au vu des divers systèmes qu'elle recouvre. Toutefois, si la syntaxe et les noms des éléments de description changent d'un système à l'autre, la notion d'objet, l'organisation hiérarchique des classes, le mécanisme d'héritage, la majorité des opérateurs de description de domaines d'attributs... restent les invariants des représentations par objets.

Une hiérarchie de spécialisation offre à la fois un moyen "économique" de représenter les connaissances et un bon support pour mettre en œuvre un raisonnement classificatoire. Dans le premier cas, la récupération des connaissances factorisées dans la hiérarchie est assurée par le mécanisme d'héritage et, dans le deuxième cas, la classification s'appuie sur un mécanisme d'appariement pour confronter de nouveaux objets à ceux de la hiérarchie. Le chapitre IV souligne plus particulièrement le rapport entre les possibilités de description des objets et le mode d'utilisation des objets, classification ou construction.

En ce qui concerne le modèle TROPES, le premier noyau présenté ici pose les bases d'un système à objets clairement orienté vers le raisonnement classificatoire. Le cadre conceptuel strict qu'il impose constitue un point de départ intéressant pour envisager de nouvelles extensions. Tel quel, le modèle TROPES ne possède pas un langage de description adapté à un processus de construction d'objets. La possibilité de définir des calculs d'attributs globalement et l'intégration de la notion de contraintes préfigure de nouvelles possibilités en ce sens. La mise en place d'un processus capable d'allier phase d'identification et phase de construction d'objets est alors une nouvelle problématique envisageable dans le modèle TROPES. Ce rapport se propose de donner des éléments de réponse généraux à une telle problématique tout en bénéficiant du contexte favorable fourni par le modèle TROPES. L'aspect multi-points de vue de ce modèle ne jouant pas un rôle particulier dans ce processus, le modèle est ramené au cas d'un seul point de vue par concept.

Le problème de la connaissance incomplète s'avère être au cœur de la problématique soulevée, c'est pourquoi le chapitre suivant se propose d'introduire les techniques de traitement de connaissances incomplètes et leurs applications dans le contexte des représentations par objets.

Chapitre III
OBJET ET RAISONNEMENT HYPOTHETIQUE

1. Introduction

Les mécanismes de raisonnement mis en place dans les systèmes de représentation par objets, et plus spécialement pour ceux qui proposent un mécanisme de classification, se heurtent au problème d'incomplétude d'une base de connaissances. Principalement basés sur un mode de raisonnement déductif, ces systèmes se bornent à expliciter ce qui est implicitement décrit dans la base de connaissances. Le moindre manque de connaissance met en échec un tel raisonnement car il ne raisonne que sur des connaissances sûres.

Dans cette partie, le problème de la connaissance incomplète est abordé de façon générale dans un premier temps, l'objectif étant d'en faire ressortir les causes et les conséquences. Le raisonnement doit s'adapter au caractère incertain, et révisable, des faits qu'il manipule. Pour ce faire, il doit tolérer les mises-à-jour possibles des connaissances lorsqu'il est de type raisonnement par défaut, ou encore, il doit supporter simultanément plusieurs états possibles de la connaissance lorsqu'il gère explicitement la notion d'hypothèse. Dès lors, la principale difficulté réside dans la gestion de la base de connaissances dont la cohérence est garante de celle du raisonnement. Cette gestion peut être isolée du raisonnement lui-même et être déléguée à un module spécialisé, appelé système de maintien du raisonnement.

Le deuxième point présente à travers le TMS et l'ATMS la notion de système de maintien du raisonnement. Ces modèles indépendants de toute représentation particulière mettent en évidence les principes généraux soutenant un raisonnement non monotone dans le cas du TMS, et soutenant un raisonnement monotone et hypothétique dans le cas de l'ATMS.

Enfin, ces problèmes et solutions sont soulignés dans le contexte des représentations par objets. Plus particulièrement, un travail original sur une gestion d'instances hypothétiques est étudiée dans le cadre du modèle TROPES [Gens90]. A priori éloigné du fonctionnement de l'ATMS, l'étude de ce gestionnaire met en évidence l'utilisation implicite de notions similaires dans la construction d'un ensemble d'instances hypothétiques. Conçu spécialement pour mettre en place une classification hypothétique, il apparaît qu'il perd de sa généralité car la gestion qu'il propose présuppose l'utilisation qui va être faite des instances hypothétiques. Il reste néanmoins la source d'inspiration de la proposition d'un gestionnaire général de versions hypothétiques d'instances qui vient conclure ce chapitre.

2. Connaissances incomplètes

Il est clair qu'une base de connaissances ne peut prétendre décrire complètement le monde réel. Les causes d'incomplétude sont de trois types [Cord86] :

a) **Informations oubliées** soit par le concepteur, soit par l'utilisateur. Ces informations sont nécessaires.

b) **Informations implicites**. Par souci d'économie dans la représentation, certaines informations ne sont pas données. Elles sont résumées par un ensemble de conventions qui permettent de les retrouver si besoin est. L'hypothèse du monde clos [Reite78] est un exemple d'expression d'une convention assurant que ce qui n'est pas connu dans le monde est faux (Si A non déductible, alors affirmer $\neg$A).

c) **Informations inconnues**. Ce sont celles qui ne sont pas accessibles dans leur intégralité. Elles peuvent être remplacées par des hypothèses.

Alors que le premier type d'informations incomplètes doit être résolu par une coopération entre l'utilisateur et le système, les deux types suivants sont traités, respectivement, par le raisonnement par défaut et par le raisonnement hypothétique.

Le raisonnement par défaut s'appuie sur un ensemble de règles qui permet de compléter la connaissance du monde au fur et à mesure. Ces règles, ou conventions, peuvent traduire une hypothèse globale sur la représentation du monde (e.g. hypothèse du monde clos), dans ce cas le monde modélisé est décrit en connaissance de cause ; ou encore, elles peuvent traduire une hypothèse ponctuelle (e.g. expression de typicalité), dans ce cas les règles sont intégrées directement dans la description du monde modélisé sous forme de défauts.

L'utilisation de défaut donne lieu à un raisonnement à caractère non monotone, c'est-à-dire que la prise en compte d'une nouvelle inférence n'assure pas une augmentation croissante de la base de faits. En effet, les hypothèses introduites sont crues jusqu'à preuve du contraire, elles peuvent être invalidées par l'introduction d'une nouvelle connaissance (cf. Figure III.1). La remise en cause d'une hypothèse doit aussi entraîner la remise en cause des faits qu'elle a permis de déduire.

Soient la règle de défaut R-Déf et la règle normale R-Nor :

R-Déf : "Ceux qui sont à Grenoble au mois d'août sont bronzés"
R-Nor : "Ceux qui rédigent leur thèse ne sont pas bronzés"

Soit F1 le fait suivant :

F1 : "Pierre est à Grenoble au mois d'août"

La règle de défaut R-Déf est alors appliquée :

=> "Pierre est bronzé" (R-Déf)

Soit F2 le nouveau fait suivant :

F2 : "Pierre rédige sa thèse"

La règle normale R-Nor est alors appliquée

=> "Pierre n'est pas bronzé" (R-Nor)
=> **Invalidation** du résultat précédemment obtenu par R-Déf.

Figure III.1 : Exemple de non-monotonie introduite par les défauts.

Le raisonnement hypothétique consiste à remplacer les connaissances inconnues par un ensemble d'hypothèses possibles. Le raisonnement cherche sous quelles hypothèses le but recherché peut être atteint. Plusieurs groupes, ou conjonction, d'hypothèses peuvent mener à une solution, chacun de ces groupes représente une solution au problème initial. Par exemple, le diagnostic médical met en œuvre un raisonnement hypothétique qui consiste à supposer un certains nombre de maladies suggérées par les signes cliniques constatés, puis à chercher celles qui sont possibles.

Le raisonnement hypothétique peut être vu comme un raisonnement non monotone où les hypothèses sont incorporées progressivement une à une, certaines peuvent être abandonnées lorsqu'elles donnent lieu à une incohérence. La remise en cause des hypothèses est mise en place par un mécanisme de rétrogression.

Le raisonnement hypothétique peut aussi être vu comme plusieurs raisonnements monotones qui se développent séparément. Les connaissances produites par l'un de ces raisonnements forment son **contexte hypothétique**. Lorsqu'un contexte mène à une inconsistance, il est abandonné. Les différents contextes hypothétiques développés sont éventuellement contradictoires entre eux. Chaque contexte hypothétique correspond à une des combinaisons possibles et cohérentes d'hypothèses. Une telle combinaison d'hypothèses est appelée un **environnement**.

Que ce soit le raisonnement par défaut ou le raisonnement hypothétique, les deux nécessitent une gestion des remises en cause d'hypothèses pour assurer la cohérence de la base de connaissances. Par la suite, la notion de système de maintien du raisonnement est abordée à travers le traitement des mises à jour (TMS) et des gestions de contextes (ATMS).

3. Maintien du raisonnement

Il faut pouvoir traiter la propagation des changements dûs à la non monotonie, et/ou la gestion de plusieurs contextes hypothétiques. Ces rôles sont généralement délégués à un module particulier qui s'interpose entre le système de raisonnement et la base de faits. Ce module est appelé un système de maintien du raisonnement (SMR). Les différents modèles de SMR s'inspirent des **TMS à propagation** et des **TMS à contextes** dont le TMS [Doyl79] et l'ATMS [DeKl86] sont les représentants respectifs présentés par la suite.

3.1. TMS (Truth Maintenance System)

Jon Doyle a proposé la mise en place d'un système de maintenance de la vérité à **propagation** comme soutien à un système de raisonnement. Son rôle d'interface entre le raisonnement et l'ensemble des faits consiste à répercuter dans la base de connaissances les modifications issues de l'application d'inférences (cf. Figure III.2). Il maintient ainsi la cohérence de la base de faits et décharge le raisonnement de cette tâche.

Pour ce faire, le TMS mémorise les inférences réalisées sous la forme de dépendances entre faits, et propage, s'il y a lieu, les modifications dues à la validation ou l'invalidation de certains faits. Lorsqu'il rencontre une inconsistance, il effectue une phase de *rétrogression* qui consiste à choisir un des faits dont la validité, ou l'invalidité, n'a été que supposée par le raisonnement. Cette hypothèse est alors modifiée, si l'inconsistance persiste il incrimine une autre hypothèse et ainsi de suite jusqu'à retrouver la cohérence de la base.

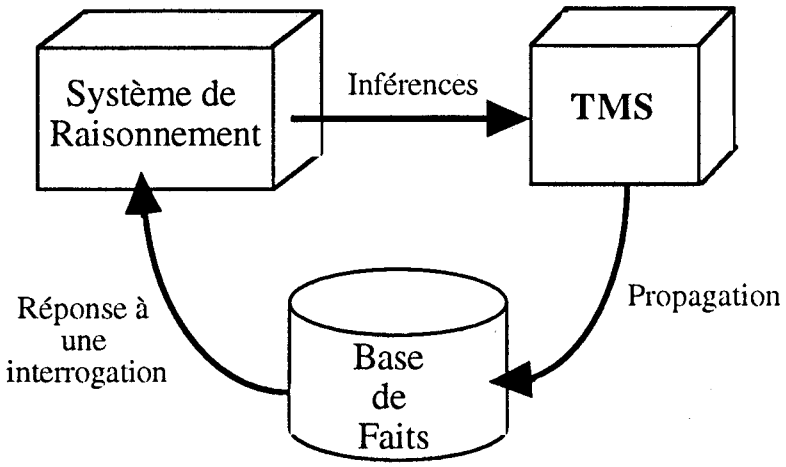

Figure III.2 : Architecture d'un système de raisonnement épaulé par un TMS pour maintenir la cohérence de la base de faits.

Le TMS considère la base de faits comme un réseau de **nœuds** reliés entre eux par leurs **justifications**. Chaque nœud décrit l'état d'un fait particulier qui peut-être :

- "le fait est cru", alors le nœud est **IN**.
- "le fait n'est pas cru", alors le nœud est **OUT**.

L'établissement de l'état d'un nœud est dépendant des justifications qui lui sont associées, elles peuvent être de deux types :

- **SL-justification (Support List)** : c'est une paire d'ensembles de nœuds appelés IN liste et OUT liste. La première contient les nœuds qui doivent être IN et la deuxième ceux qui doivent être OUT pour que le nœud considéré soit IN. Si au moins un des nœuds de la IN liste est OUT, ou au moins un des nœuds de la OUT liste est IN, la SL-justification est dite **invalide**.

  Une justification d'un nœud N est donnée sous la forme :

  < {IN liste}, {OUT liste}>: N

Ainsi,

- <{}, {}>: A, signifie que A est un axiome, ou prémisse.
- <{IN liste}, {}>: A, signifie que A est inféré par déduction classique.
- <{}, {OUT liste}>: A, signifie que A est valide par défaut.

- **CP-Justification (Conditional Proof)** : elle est utilisée lors de la remise en cause d'hypothèses dûe à une inconsistance. Elle se compose d'un nœud **conséquence**, des deux ensembles de nœuds appelés INhyp liste et OUThyp liste. La CP-justification est valide si la conséquence est valide quand tous les nœuds de la INhyp liste sont IN, et tous ceux de la OUThyp liste sont OUT. Cette justification est complexe à utiliser, elle l'est en général avec une OUThyp liste vide. Quand c'est possible, elle est remplacée par des SL-justifications. Pour conserver à l'exposé du TMS toute sa clarté, les CP-justifications seront ignorées par la suite.

Le graphe de dépendance se visualise très bien à l'aide de symboles graphiques (cf. Figure III.3) :

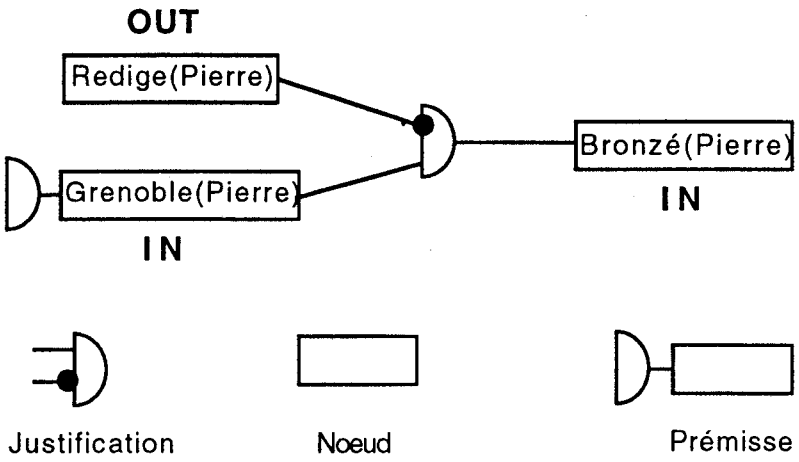

Figure III.3 : Exemple de représentation graphique d'un réseau de dépendances du TMS. Ce réseau traduit que si "Pierre est à Grenoble", et jusqu'à ce qu'il soit prouvé que "Pierre rédige sa thèse", on peut supposer "Pierre est bronzé".

La phase de propagation se déclenche quand une nouvelle justification est ajoutée au TMS. Cet ajout peut provoquer le changement d'état d'un ou plusieurs nœuds, et ceux-ci, à leur tour, sont susceptibles de modifier l'état des nœuds qu'ils justifient. Dans l'exemple, l'ajout d'une justification faisant passer le nœud "Rédige(Pierre)" à IN déclenche le passage du nœud "Bronzé(Pierre)" à OUT. Cette modification traduit l'abandon de l'hypothèse "Pierre ne rédige pas", car si "Pierre rédige" "il ne peut pas être bronzé".

Le nœud noté $\perp$ possède un statut particulier, il représente une contradiction. Sa validation décrit un état inconsistant de la base. Il est utile pour déclarer des nœuds exclusifs. En effet, s'il possède une justification dans laquelle la IN liste est non vide, les nœuds de cette liste ne peuvent être IN simultanément, sinon $\perp$ est IN et il y a inconsistance. Par exemple, si "Bronzé(Pierre)" et "Pâle(Pierre)" sont deux nœuds de la IN liste de la justification de $\perp$, ils sont alors déclarés contradictoires.

Lorsque le nœud $\perp$ est validé (IN), la base étant inconsistante la phase de rétrogression a lieu (cf. Figure III.4). Le principe est de remettre en cause certaines hypothèses qui supportent les nœuds de sa IN liste, et qui possèdent une OUT liste non vide, c'est-à-dire les défauts. L'hypothèse est remise en cause en validant l'un des nœuds de sa OUT liste par construction d'une nouvelle justification de ce nœud. Cette justification est une copie de celle de l'hypothèse, où l'on retire de la OUT liste le nœud à valider, et si elle existe, on retire l'hypothèse de la IN

liste. Comme l'hypothèse était IN, on est sûr que la nouvelle justification ainsi construite va valider le nœud choisi.

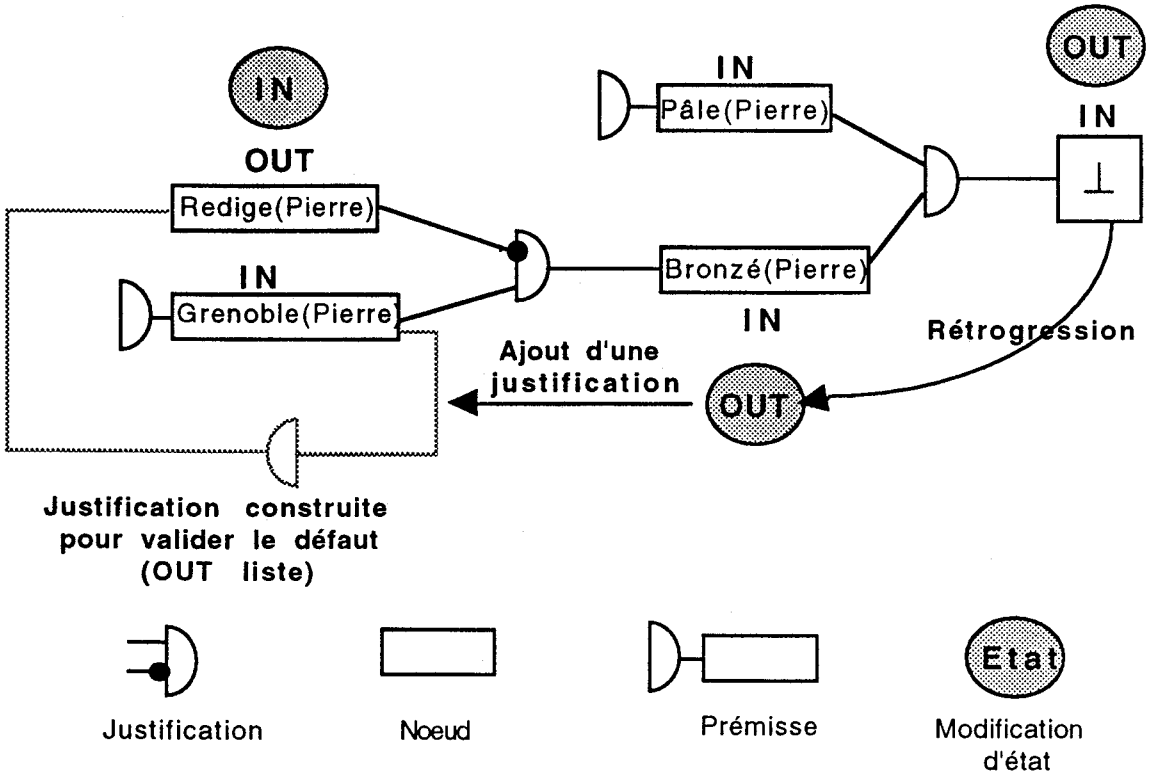

Figure III.4 : Phase de rétrogression du TMS. Les nœuds "Pâle(Pierre)" et "Bronzé(Pierre)" sont IN tous les deux, le nœud ⊥ est IN en conséquence. La base est donc inconsistante, la phase de rétrogression démarre en essayant d'incriminer l'un des deux nœuds, le critère de choix se fait à partir de ceux qui ont dans leurs justifications une OUT liste non vide. Le nœud "Bronzé(Pierre)" est donc sélectionné, on valide un nœud de sa OUT liste en construisant une justification à partir de celle de "Bronzé(Pierre)". Ce nœud est "Rédige(Pierre)" qui passe à IN, "Bronzé(Pierre)" est donc à OUT, et la consistance de la base est retrouvée puisque ⊥ passe à OUT.

A partir de cette gestion de graphe de dépendances, le TMS maintient la base cohérente au fur et à mesure que le système de raisonnement lui fournit un nouveau fait inféré, ce qui donne lieu à la création de nouvelles justifications, et d'un nœud si le fait est nouveau. Ces ajouts peuvent provoquer propagation et rétrogression.

Il est à noter des problèmes dûs à la présence possible de cycles particuliers (cycles impairs ou pairs). Les cycles impairs posent des problèmes de terminaison de la propagation, et les cycles pairs introduisent des ambiguïtés dans l'étiquetage du graphe. Diverses solutions sont proposées dont on trouvera de plus amples détails dans [Euze90], [Hato&91].

La principale critique concerne la gestion des hypothèses sur lesquelles se développe le raisonnement. La dominance des justifications cache le rôle des hypothèses et n'en permet pas une manipulation aisée. Le TMS ne travaille que sur un contexte hypothétique à la fois, pour passer d'un contexte à un autre il doit mettre en œuvre propagation et rétrogression. L'ATMS, présenté dans la partie suivante, est proposé par J. De Kleer pour permettre une pleine gestion de la notion d'hypothèse.

3.2. ATMS (Assumption-based TMS)

Le principe de l'ATMS est de gérer de façon explicite des ensembles d'hypothèses. Pour cela, l'ensemble global des hypothèses est organisé en environnements qui représentent les

sous-ensembles cohérents d'hypothèses (un environnement peut aussi être vu comme une conjonction d'hypothèses) à partir desquels le raisonnement se développe. L'ensemble des faits qui peuvent être déduits à partir d'un environnement spécifique forme un contexte. Ce dernier est construit par le raisonnement de façon monotone puisque, les hypothèses étant les seules faits révisables, une modification de l'ensemble des hypothèses entraîne un changement de contexte. L'ATMS est appelé **TMS à contextes** puisqu'il permet la gestion simultanée de plusieurs contextes à travers lesquels le raisonnement peut explorer différentes solutions en ajoutant et/ou en retirant des hypothèses.

Le système de raisonnement communique à l'ATMS les justifications correspondant à des inférences de nature monotone (les OUT listes ne sont donc plus utiles dans les justifications). Le système de raisonnement peut interroger l'ATMS sur la validité d'un nœud dans un contexte précis (cf. Figure III.5).

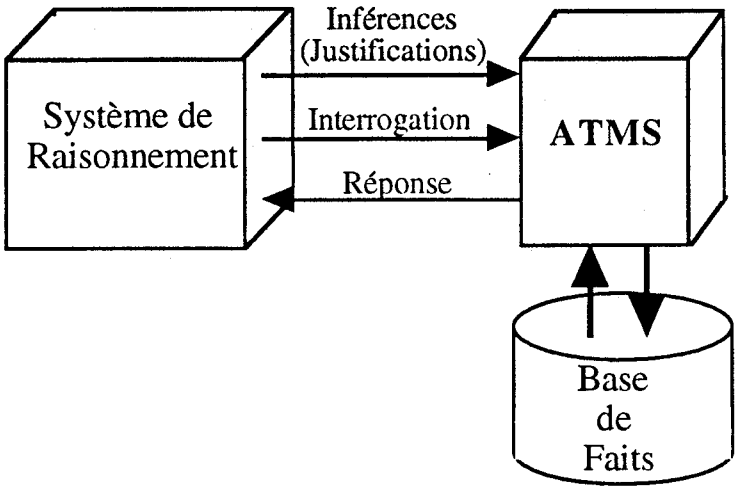

Figure III.5 : Architecture d'un système de raisonnement couplé avec un ATMS permettant la prise en compte de plusieurs contextes.

Une justification d'un nœud N fournit les nœuds à partir duquel peut-être dérivé N, elle est notée :

$<\{X_1,\ldots X_n\}>$: N, i.e. si $X_1,\ldots X_n$ sont valides alors N est valide.

Une hypothèse H est donc introduite par la justification :

$<\{H\}>$: H, i.e. H est valide dans tout contexte contenant H.

Un nœud N est valide s'il peut être dérivé d'un environnement E (conjonction d'hypothèses) et de l'ensemble de justifications noté J :

$E, J \vdash N$

Pour chaque nœud N construit ainsi, L'ATMS est chargé de gérer son étiquette, notée L(N) (i.e. *Label* de N), qui correspond à l'ensemble des environnements dans lequel le nœud est valide. La gestion de l'étiquette L(N) d'un nœud N consiste à s'assurer qu'elle soit consistante, correcte, complète, et minimale :

- **consistante** : les environnements de L(N) sont consistants.
  $\forall E \in L(N), \neg(E \vdash \bot)$
- **correcte** : N est dérivable de chaque environnement de L(N).
  $\forall E \in L(N), E, J \vdash N$
- **complète** : Chaque environnement consistant E, à partir duquel N est dérivable, est un sur-ensemble d'au moins un environnement E' de L(N).
  $(\forall E, E, J \vdash N) \Rightarrow (\exists E' \in L(N), E' \subset E)$

- **minimale** : Dans L(N), aucun environnement n'est inclus dans un autre.
  $\forall E \in L(N), \forall E' \in L(N), E' \not\subset E$

Un environnement inconsistant est appelé **nogood**, il est ajouté à l'étiquette d'un nœud spécial $N_\perp$ représentant l'ensemble des environnements menant à une inconsistance.

La minimalité d'une étiquette L(N) est suffisante car le raisonnement étant monotone tous les environnements consistants incluant l'un des environnements de L(N) permettent de dériver le nœud N. Lorsqu'une étiquette consistante E est inclue dans une autre étiquette consistante E', le contexte de E est inclus dans celui de E' (cf. Figure III.6).

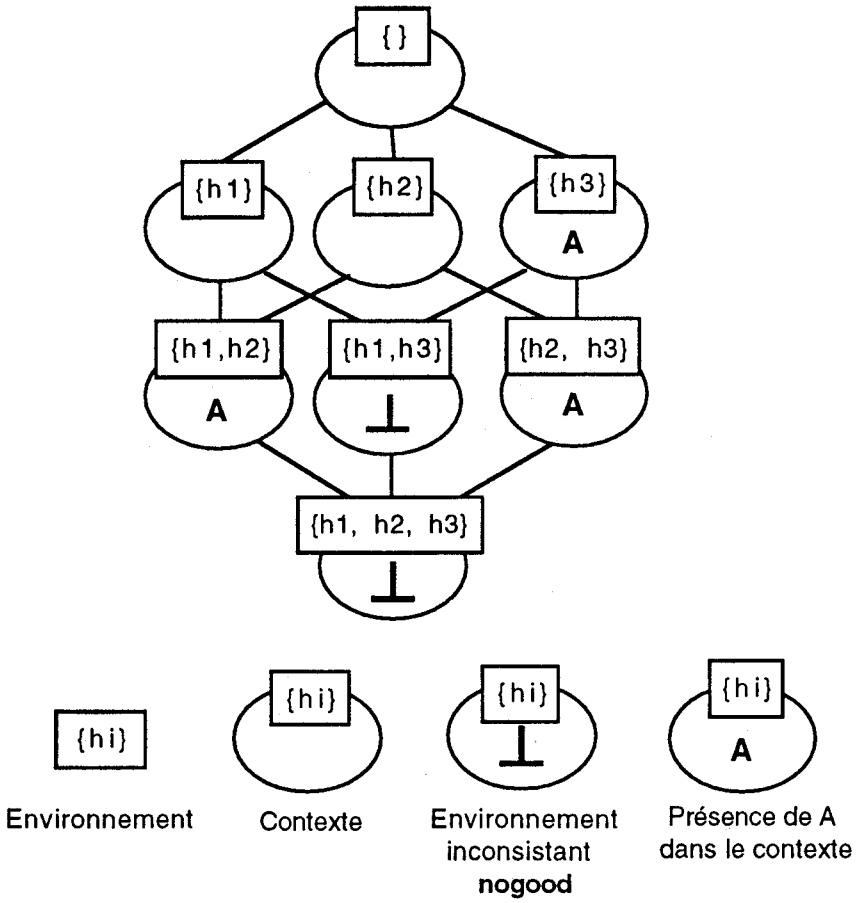

Figure III.6 : Exemple de contextes organisés par la relation d'inclusion qui est induite par celle des environnements. Les deux environnements constitués des hypothèses respectives h1 et h3, et h1, h2 et h3, sont **nogoods** car ils mènent à une inconsistance. Le nœud, ou fait, A est valide pour les environnements {h3}, {h1, h2} et {h2, h3}, son étiquette est donc {{h3}, {h1, h2}}, car d'après le critère de minimalité {h2,h3} est inutile.

La gestion des étiquettes se ramènent donc lors de l'ajout d'une justification J aux étapes suivantes (adapté de [Hato&91]) :

1) Calculer l'étiquette du nœud modifié, ou créé, par la justification J, en y ajoutant l'intersection des étiquettes des nœuds antécédents de la justification J, et en supprimant de la nouvelle étiquette tous les environnements inconsistants (consistance) et englobants d'autres environnements de l'étiquette (minimalité).
2) Si l'étiquette est inchangée alors fin.

3) Si le noeud est $N_\perp$ (nœud inconsistant) chaque environnement est **nogood**, ils sont supprimés, ainsi que leurs sur-ensembles, de toutes les étiquettes de nœud où ils apparaissent.

4) Sinon, les étiquettes des nœuds justifiés par le nœud courant sont mises à jour suivant le même processus.

Le test de fin 2) implique que même en cas de cycle la propagation s'arrête dès que les étiquettes sont dans un état stable.

Les étiquettes de chaque nœud étant fixées, le système de raisonnement peut savoir si un nœud N fait partie du contexte dans lequel il est en train d'évoluer en fournissant à l'ATMS son environnement hypothétique courant E. L'ATMS compare E à ses données :

- Le nœud N est **nécessairement présent**, si E englobe l'un des environnements de L(N).
- Le nœud est **nécessairement absent**, si $E \cup \{N\}$ englobe l'un des environnements de $L(N_\perp)$.
- Sinon, il est **couramment absent**.

En d'autres termes, le nœud est valide, invalide ou inconnu.

L'ATMS introduit la notion de validité contextuelle des faits, et permet au raisonnement de naviguer à travers des environnements hypothétiques différents en ajoutant et/ou supprimant des hypothèses, et donc de mener de front plusieurs raisonnement correspondant à des contextes différents. On retrouve dans KEE [Film88] la notion de contexte qui est appelée monde hypothétique, dans OMEGA [Atta&85] [Atta&86] et ART [Clay86] cette notion est appelée "point de vue".

L'inconvénient majeur de l'ATMS est que dans chaque contexte le raisonnement est supposé monotone. Cette supposition est la conséquence directe de la circonscription de l'ensemble des hypothèses, c'est-à-dire que l'on désigne pour chaque contexte les faits qui jouent le rôle d'hypothèse. Les autres faits du contexte ne peuvent pas être remis en cause. C'est là un inconvénient que ne connait pas le TMS puisque pour lui tous les faits sont révisables, il se charge de propager les modifications. Diverses extensions de l'ATMS vers un raisonnement non monotone ont été proposées dont on pourra trouver des descriptions dans [Hato&91] [Euze90].

Dans la partie suivante, la maintenance du raisonnement est abordée dans le cadre d'une représentation par objets. Dans un tel modèle, si le formalisme utilisé est assez éloigné des nœuds et justifications des systèmes précédents, les notions de modification de la connaissance et d'hypothèse y sont, par contre, également présentes. La mise en évidence de ces notions dans le formalisme à base d'objets permet d'adopter des solutions de gestion de la cohérence similaires.

4. Maintenance du raisonnement et objets

A l'image des deux systèmes précédents, une base de connaissances par objets peut tirer parti d'une gestion des modifications de la base et de l'introduction de la notion d'hypothèse. Cette partie se propose d'en montrer l'utilité et l'adaptation au contexte objet.

4.1. Mise à jour d'une base d'objets

Une représentation de connaissances par objet ne dispose que d'un nombre limité de mécanismes qui sont propres à la structuration des connaissances sous forme d'objets : hiérarchies de classes, descriptions de classes, instances, attributs... Ces mécanismes, tels que l'héritage, la classification, le déclenchement de calcul d'attributs, la gestion de relation inter-objets... se préoccupent en général eux-mêmes de la cohérence des résultats qu'ils fournissent. Les mécanismes généraux de raisonnement sur les objets sont monotones.

Les problèmes d'incohérences peuvent, par contre, apparaître dans la description même de la base, au niveau des hiérarchies et des classes. En effet, si des classes sont mal spécifiées, leur description étant incohérente, aucune instance ne peut leur être attachées. Ce problème n'est pas tant un problème de maintenance du raisonnement qu'un problème de validation de la description de la base. Pour ce faire, des techniques de typage de classes [Capp&93] ou encore, dans les langages terminologiques, de calcul de subsumants et subsumés d'une classe (*concept*), peuvent être mises en place pour assurer une construction valide de la base (cf. §II.3.1). Ces techniques sont cependant mises en déroute lorsque le système exploite des connaissances procédurales ou, plus généralement, des connaissances auxquelles il n'a pas accès complètement. Dans ce cas, la validation est partielle.

Par ailleurs, une source d'incohérence en relation avec le raisonnement provient de l'utilisation non monotone de la base. En effet, si les raisonnements sont monotones, les instances, qui représentent l'interface entre le système et l'utilisateur, sont susceptibles d'être modifiées. Cela devient d'autant plus vrai que la base est partagée par plusieurs utilisateurs, et que le nombre d'objets est grand.

Le problème d'incohérence n'apparaît pas si une instance modifiée est considérée comme une nouvelle instance, les mécanismes de raisonnement auxquels elle peut prétendre lui sont alors de nouveau appliqués. Cette méthode peut être coûteuse en temps puisque tout est à refaire. De plus, lorsque d'autres instances dépendent de l'instance modifiée, il est aussi nécessaire de leurs appliquer des modifications, une phase de propagation doit se mettre alors en place. Ce maintien de la cohérence, pour être performant, a tout avantage à exploiter les connaissances déjà acquises sur l'instance à modifier, puisque certains résultats indépendants des données altérées ne seront pas affectés.

Partant de ce constat, Jérôme Euzenat a introduit dans le système SHIRKA [Rech85] un système de maintien du raisonnement de type TMS [Euze87]. Celui-ci est chargé de mettre à jour les valeurs d'attributs d'une instance qui a été modifiée en soumettant à de nouvelles évaluations les attributs qui ont été affectés, directement ou indirectement, par la modification de l'instance. Pour ce faire, le TMS gère des dépendances entre attributs en associant à chacun d'eux les justifications qui décrivent les façons par lesquelles la valeur de l'attribut a été obtenue, et les dépendances qui désignent les justifications dans lesquelles la valeur de l'attribut intervient (cf. Figure III.7). Un processus de propagation se déclenche lors d'une mise à jour.

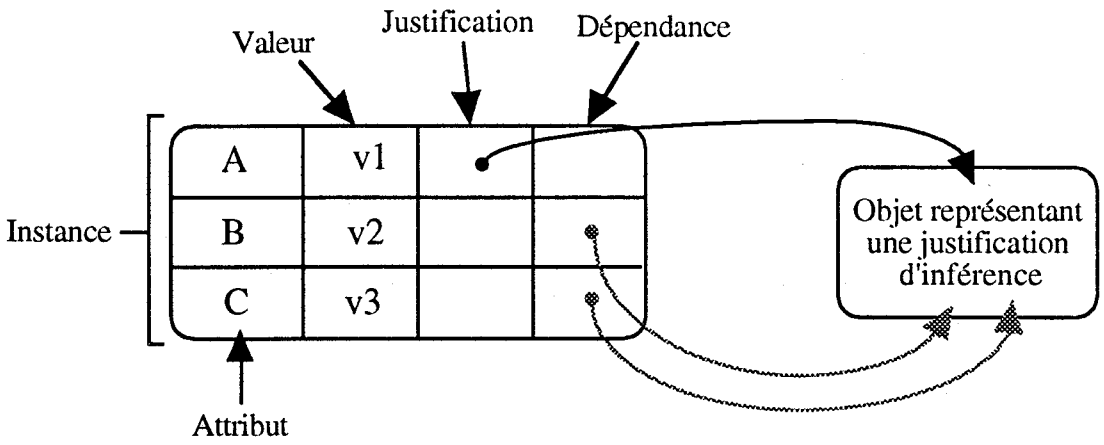

Figure III.7 : Exemple d'un réseau de justifications et de dépendances dans SHIRKA/TMS. L'exemple montre que la valeur de l'attribut A de l'instance a été obtenue par inférence à partir de ses attributs B et C. Cet objet justification pourrait être, par exemple, celui décrivant un appel à un attachement procédural.

Sur un principe similaire, Jérôme Gensel a introduit un TMS chargé de gérer les modifications des instances de TROPES [Gens90]. De plus, dans ce modèle objet, les instances étant classées à l'aide du mécanisme de classification, la modification d'une instance peut

remettre en cause sa position dans la hiérarchie, elle doit donc être aussi gérée. Un mécanisme de relocalisation d'instance est proposé pour jouer le rôle de mécanisme de maintien du raisonnement spécialisé dans la classification d'instance, il est chargé de repositionner convenablement l'instance dans la hiérarchie [Mari93]. Ce mécanisme tire parti du résultat de la classification précédente, et évite ainsi de reclasser complètement l'instance.

Il faut noter que le raisonnement étant monotone, seule l'utilisation de la base est non monotone, la gestion de la cohérence ne nécessite donc pas de phase de rétrogression et les justifications se suffisent de la IN liste du TMS. Le statut IN ou OUT d'un nœud correspond respectivement aux "non modifié" ou "modifié" d'un attribut, ou d'une instance dans le cas de la relocalisation.

Cette utilisation du TMS n'est pas exploitée pour permettre une manipulation de la notion d'hypothèse mais plutôt pour traduire une évolution des connaissances. L'extension d'une représentation par objets vers la mise en place d'une composante hypothétique est par contre le thème principal des parties suivantes.

4.2. Notion d'hypothèse et objet

S'il est important de permettre à la connaissance d'évoluer, il est tout aussi intéressant de pouvoir étudier simultanément plusieurs évolutions possibles de cette connaissance. Ces extensions servent un raisonnement hypothétique qui explore les divers contextes construits sur la base d'un ensemble d'hypothèses. Il s'avère que ce genre de cas est fréquent lorsque l'on modélise des connaissances "expertes", i.e. qui ne bénéficient pas toujours d'une théorie claire et établie, et à partir desquelles un raisonnement hypothétique consiste à ne prendre de décisions qu'après avoir exploré un ensemble d'hypothèses.

> "Dans de nombreuses situations, compléter sa connaissance est nécessaire lorsque l'incomplétude de celle-ci est manifeste et pose problème. L'observation d'un phénomène inexpliqué, c'est-à-dire d'un phénomène dont on ne peut prévoir et justifier l'existence à partir de ce que l'on connait, **est à la base de l'activité de découverte scientifique (Popper, 1973)**. Celle-ci a pour rôle de **construire des hypothèses** ***(explanans)*** **devant compléter notre connaissance** afin de rendre compte de tels phénomènes ***(explanandum)***." [Hat&91a]

Ce type de fonctionnement est l'un des atouts du langage hybride KEE qui est capable de gérer la base de connaissances comme plusieurs mondes hypothétiques, et qui intègre pour cela un système de maintien du raisonnement proche de l'ATMS. Chaque monde décrit une étape d'exploration du raisonnement. L'ensemble des mondes, à l'image de l'ensemble des contextes de l'ATMS, est organisé en un graphe d'héritage. Un monde créé à partir d'un autre hérite des connaissances du monde père.

Dans le cadre d'un modèle de représentation par objets, la structuration complexe de la connaissance implique une interprétation de la notion d'hypothèse en fonction des différentes entités de description du modèle. C'est-à-dire qu'il faut se demander où la connaissance peut être incomplète. On peut diviser une base de connaissances objets en deux parties : la connaissance descriptive, qui comporte les descriptions des concepts modélisés (i.e. les hiérarchies de classes), et la connaissance assertionnelle, qui rassemble les individus (i.e. les instances). On retrouve clairement ces deux niveaux de connaissances dans le système KRYPTON [Brac&83]. En effet, l'ensemble des opérations offertes par ce système est fournie sous la forme de deux *types abstraits* : la *ABox* permettant la manipulation d'assertions, et la *TBox* permettant la manipulation des termes sur lesquelles sont fondées les assertions de la *Abox*.

En ce qui concerne la connaissance descriptive, la cause d'incomplétude provient du décalage possible entre ce que l'on veut décrire et ce qui est effectivement décrit. En effet, dans les systèmes classificatoires, par exemple, la description des classes est très importante puisque le raisonnement se base dessus pour désigner les classes d'appartenance d'une instance. Si par construction d'une hiérarchie de classes l'inclusion ensembliste est assurée grâce à l'héritage des propriétés d'une classe vers ses sous-classes, d'autres propriétés de la hiérarchie peuvent être supposées sans être prouvées par les descriptions des classes.

Dans [Euze93a], diverses propriétés sémantiques d'une hiérarchie de classes sont étudiées dans le cadre de la classification : l'univocité, la déterminance, l'exhaustivité et l'exclusivité. L'**univocité** assure qu'une instance n'aura qu'une plus petite classe sûre (si une instance appartient à deux classes simultanément, il existe alors au moins une sous-classe commune telle que l'instance lui appartienne) ; la **déterminance** entraîne que la classe est une feuille de la hiérarchie. L'**exclusivité** traduit que les sous-classes directes d'une classe sont toutes disjointes (disjonction ensembliste) entre elles, c'est-à-dire qu'une instance ne peut appartenir à plus d'une sous-classe en même temps ; l'**exhaustivité** implique que l'union des ensembles représentés par ses sous-classes directes forment un recouvrement de l'ensemble représenté par la classe.

Les notions d'objectivité et de subjectivité sont introduites pour signaler le décalage possible entre ce qui est exprimé par les descriptions des classes et les propriétés attendues d'une hiérarchie de classes : une propriété sera définie **objective** si elle est déduite automatiquement, et **subjective** si elle est imposée manuellement par l'utilisateur et ne peut être déduite par le système. La subjectivité est liée directement à l'incomplétude des bases de connaissances qui ne peuvent prétendre "renfermer toutes les données qui permettraient d'établir objectivement les propriétés".

Prenons le cas de l'exclusivité subjective de deux classes d'une hiérarchie. Il y a alors un manque de connaissances au niveau des descriptions de classes. Diverses solutions à ce problème peuvent être adoptées :

a) Au niveau de la modélisation : on peut décrire la disjonction par l'ajout de propriétés aux descriptions des classes concernées. Ces propriétés "forcent" la disjonction recherchée. Cette solution est contestable si les propriétés ajoutées à la description de chaque classe, et donc aux instances représentées par celle-ci, n'ont d'autre rôle que de garantir la disjonction avec d'autres classes. En revanche, si elles sont naturelles, elles complètent la connaissance et sont alors tout à fait pertinentes.

- Si C et C' sont connues disjointes, mais ne le sont pas au vue de leur description respective, on peut ajouter un attribut A aux deux classes tel que :

  - $Dom(A, C) \cap Dom(A, C') = \emptyset$, ce qui implique $Ext(C) \cap Ext(C') = \emptyset$,

  où Dom(A, C) et Dom(A, C') représentent le domaine de l'attribut A dans les classes C et C' ; Ext(C) et Ext(C') les extensions des classes C et C'.

  Si A est un attribut superficiel, non pertinent pour le domaine modélisé, cette solution alors une "pirouette" de modélisation, dans le cas contraire A complète la description de la base de connaissances.

b) Au niveau du modèle conceptuel :

1- Une solution radicale est de poser une hypothèse générale sur le modèle indiquant que toute classe n'étant pas, à un degré quelconque, liée par la relation de spécialisation avec une autre en est disjointe.

Par exemple, on peut imposer que deux classes sœurs, c'est-à-dire deux sous-classes directes d'une même classe, sont disjointes entre elles. Une conséquence directe est que le modèle n'autorise pas ni **multi-instanciation**, ni le **multi-héritage**, dans une même hiérarchie de classes. Cette solution a l'avantage de ne pas rajouter de notion spéciale au niveau du langage, et d'imposer une discipline stricte dans la modélisation et la construction des hiérarchies. Elle peut, par contre, parfois paraître un peu trop contraignante en fonction de ce qui est à modéliser. De plus, le modèle doit offrir une réponse à la question suivante : "Comment gérer le cas où l'instance pourrait appartenir aux deux classes ?". Diverses solutions sont envisageables : signaler une incohérence et/ou ne pas prendre de décision de rattachement ; obliger l'utilisateur à choisir une classe ; gérer plusieurs mondes possibles en introduisant la notion d'instance hypothétique...

• On donc une hypothèse globale sur la hiérarchie, qui est :

Pour toutes classes C et C' d'une hiérarchie, si C et C' sont deux sous-classes directes d'une même classe, alors Ext(C) ∩ Ext(C') = Ø.

Une solution autorisant l'héritage multiple consiste à baser le critère de disjonction sur l'absence de sous-classes communes :

Pour toutes classes C et C' d'une hiérarchie, si C et C' n'ont pas de sous-classes communes, alors Ext(C) ∩ Ext(C') = Ø.

2- Une solution similaire, mais moins radicale, est d'introduire une nouvelle notion au modèle permettant d'exprimer la disjonction de certaines classes. Cette relation signale le fait que toute instance appartenant à telle classe ne peut appartenir à telle autre. Cette relation doit alors être prise en compte lors de l'établissement de l'appartenance à des classes qui vérifient cette relation. Elle doit aussi répondre au même problème que la précédente ; toutes les solutions proposées précédemment restent envisageables sauf bien sûr celle qui consistait à signaler une incohérence dans la base.

• On définit dans le langage un opérateur Disjoint tel que :

∀ C, C', Disjoint (C, C') ⇒ (Ext(C) ∩ Ext(C') = Ø).

Les solutions b-1 et b-2 paraissent répondre au problème d'expressivité rencontré, à savoir comment déclarer deux classes disjointes quand on ne possède pas la connaissance descriptive (description des classes) nécessaire. Elles doivent toutes deux offrir une solution au cas où une instance serait reconnue par un mécanisme de classification comme pouvant appartenir à deux classes connues disjointes. La solution des mondes possibles paraît la plus prometteuse et peut être paramétrée en fonction du mécanisme mis en œuvre par le raisonnement de façon à offrir aussi les autres solutions : choix d'un monde possible ; ou rejet de tous les mondes possibles, i.e. refus d'un raisonnement hypothétique (le conflit est signalée comme une incohérence de la base).

A travers cet exemple sur l'incomplétude possible de la connaissance descriptive, deux réactions du modèle de représentation sont possibles : déclaration d'inconsistance, ou production d'hypothèses. Ce deuxième cas traduit une répercussion du manque de connaissances descriptives au niveau des instances manipulées, i.e. de la connaissance assertionnelle.

En ce qui concerne la connaissance assertionnelle, deux cas de connaissances incomplètes peuvent se présenter :

- L'instance est incomplète, des valeurs d'attributs sont manquantes.
- La connaissance sur les classes d'appartenance d'une classe est insuffisante pour poursuivre un raisonnement. Ce peut-être dû au fait que l'instance est incomplète ou au fait que la description des classes est incomplète (voir ci-dessus).

Il est intéressant, après applications de raisonnements sur une instance, lorsqu'elle reste incomplète, d'émettre des hypothèses afin de la compléter et de poursuivre ainsi le raisonnement initial (qui peut être complexe). Cette complétion d'instance ne représente qu'une version possible de l'instance originale puisqu'elle est produite sous certaines hypothèses. Une telle instance est appelée instance hypothétique.

Un mécanisme mettant en œuvre ce raisonnement produira un certain nombre de versions acceptables de l'instance incomplète et, par propagation du caractère hypothétique des connaissances, de versions d'autres instances qui sont liées à la première par des liens de dépendance. Utilisées dans le contexte d'un raisonnement plus général, certaines de ses instances hypothétiques pourront être réfutées par la suite car elles ne vérifient pas les contraintes posées par ce contexte englobant ; par contre, si plusieurs versions sont possibles, une gestion des versions doit être effectuées. Ces versions peuvent traduire le fait que le raisonnement courant, mené par le système, admet plusieurs solutions au problème initial.

La production d'hypothèses peut donc s'effectuer soit au niveau des valeurs d'attributs d'une instance, soit au niveau de son appartenance aux classes. Ces deux niveaux ne sont certes pas indépendants puisque attacher une instance à une classe peut produire de nouvelles valeurs

d'attributs, qui peuvent être elles-mêmes hypothétiques. Et inversement, la prise en compte de valeurs d'attributs hypothétiques peut permettre au raisonnement de considérer de nouvelles hypothèses de rattachement de l'instance à des classes.

Une hiérarchie de classes peut devenir un support très intéressant pour la complétion d'une instance où l'on peut tirer parti des propriétés particulières de la hiérarchie pour produire un nombre limitée d'hypothèses, et donc d'instances hypothétiques. Pour ce faire, la notion d'environnement doit être gérée. Les principes de l'ATMS sont donc appliqués en restreignant la notion de contexte à la notion d'instance hypothétique et d'environnement à un ensemble d'hypothèses sur des valeurs d'attributs et/ou sur l'appartenance d'une instance à une classe.

Par la suite, l'intégration de la notion d'instances hypothétiques est introduite dans le cadre du travail de Jérôme Gensel sur la possibilité d'émettre des valeurs hypothétiques sur les attributs d'une instance dans le modèle TROPES.

5. Raisonnement hypothétique dans TROPES

Dans [Gens 90], la notion d'instance hypothétique est introduite dans le modèle TROPES et permet une gestion de la notion de contexte. Le raisonnement hypothétique a été introduit dans TROPES pour permettre l'exploration d'un ensemble de valeurs pour un ou plusieurs attributs d'une instance. L'objectif de l'utilisateur est d'observer les conséquences de son indécision à travers la classification effectuée par le système pour chacune des instances que l'on peut former à partir de ces ensembles de valeur. L'indécision n'est pas seulement introduite par l'utilisateur mais aussi par les mécanismes d'inférence de valeur d'attributs qui sont aussi susceptibles de fournir plusieurs valeurs possibles pour un attribut. Ces inférences sont déclenchées en fonction des besoins en valeur d'attribut, elles sont indépendantes des classes et se basent uniquement sur les valeurs d'attributs déjà obtenues. En effet, à chaque attribut est associée une tâche exprimant la stratégie de calcul de cet attribut [Orsi90] (cf. §II.3.4.3).

5.1. Gestion des instances hypothétiques

La création d'une instance hypothétique est donc le résultat d'introduction d'une ou plusieurs valeurs hypothétiques d'un attribut. Cette instance vient enrichir le **bassin hypothétique** en s'insérant dans le chaînage hypothétique reflétant les divers contextes (cf. Figure III.8).

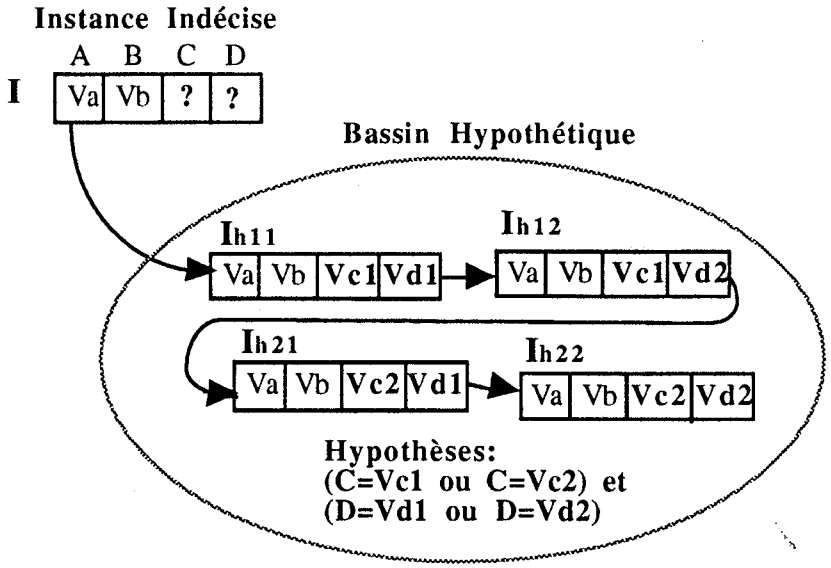

Figure III.8 : Bassin hypothétique de l'instance indécise I. Ce bassin est construit à partir des hypothèses sur les valeurs des attributs C et D. Quatre instances hypothétiques sont construites par recopie de l'instance indécise I, et complétées chacune par l'une des combinaisons possibles d'hypothèses.

Le bassin hypothétique représente l'ensemble des instances hypothétiques rattachées à une **instance indécise**, c'est-à-dire l'instance incomplète initiale sur laquelle des hypothèses ont été émises. La gestion du monde hypothétique, autant de bassins hypothétiques que d'instances indécises, se fait à travers l'émission, la validation, la suppression d'hypothèses sur la valeur d'attribut, ou directement sur une instance.

En ce qui concerne les instances hypothétiques, leur structure est similaire à celle de l'instance indécise, c'est-à-dire que chacune d'elles comporte pour chaque attribut sa valeur, les informations nécessaires au TMS (c'est-à-dire, la justification qui est une liste de références aux attributs qui ont permis le calcul de l'attribut considéré, et la descendance qui est la liste des attributs pour lesquels l'attribut considéré a participé à l'évaluation).

La prise en compte des justifications au niveau des instances hypothétiques permet de distinguer deux valeurs hypothétiques égales pour un même attribut et qui ont été évaluées par des moyens différents. C'est primordial lorsque l'on est autant intéressé par la solution que par la stratégie utilisée.

5.2. Classification et instance hypothétique

La classification d'instance proposée est couplée au gestionnaire d'instances hypothétiques. A chaque fois que la classification fait face à une indécision, des instances hypothétiques sont créées en fonction des choix possibles de valeurs d'attributs. Ceci permet d'effectuer une classification en profondeur qui effectue un retour arrière lorsque des valeurs d'attributs inférées s'avèrent incohérentes avec la description de la base. La classification d'instance incomplète enchaîne les différentes phases suivantes :

1- Appariement à une classe, le résultat en est le marquage de la classe comme sûre, possible ou impossible. Dans le cas du marquage "possible", l'appariement rend une explication de ce résultat sous forme d'une liste d'attributs à inférer. Ces attributs ne sont pas évalués dans l'instance mais déterminant pour établir l'appartenance à la classe. Le marquage "sûr" déclenche la descente dans une sous-classe, tandis que le marquage "impossible" provoque l'abandon de cette branche de la hiérarchie.

2- Inférences des attributs sélectionnés en fonction des valeurs d'attributs que l'instance possède déjà (l'utilisateur est considéré comme une inférence). Ces inférences peuvent rendre plusieurs valeurs possibles pour chaque attribut.

3- Création d'instances hypothétiques correspondant aux diverses valeurs possibles de chaque attribut.

4- Sélection d'une des instances hypothétiques.

5- Descente dans une sous-classe possible de l'instance hypothétique, sur laquelle on applique récursivement la même classification. S'il n'existe plus de sous-classe, cette instance est proposée comme solution (on peut encore lancer la classification des instances hypothétiques restantes si on veut toutes les solutions).

Quand des valeurs d'attributs inférées sont incohérentes, l'instance hypothétique qui en découle est supprimée du bassin hypothétique. Pour des questions d'efficacité, il est proposé que cette classification hypothétique soit paramétrée suivant diverses options comme, par exemple, le choix d'éliminer ou pas les valeurs hypothétiques d'attributs déterminées ultérieurement à celle d'un attribut remis en cause, d'éliminer les instances hypothétiques qui ne permettent plus de descendre dans la hiérarchie... Ces choix dépendent plus de la façon dont est construite la hiérarchie.

5.3. Analyse et critique de la construction du bassin hypothétique

Le chaînage séquentiel proposé dans [Gens90] est spécifique à l'utilisation qui doit être faite des instances hypothétiques lors du couplage entre le module de gestion d'instances hypothétiques et le raisonnement classificatoire. Ce dernier consomme les instances hypothétiques au fur et à mesure et, quand une instance hypothétique ne le satisfait pas (elle reste incomplète), il la supprime et passe à la suivante.

Cependant, derrière cette plate structuration des instances hypothétiques se cache une structure plus complexe traduisant la nécessité de gérer des environnements hypothétiques construits incrémentalement (cf. Figure III.9).

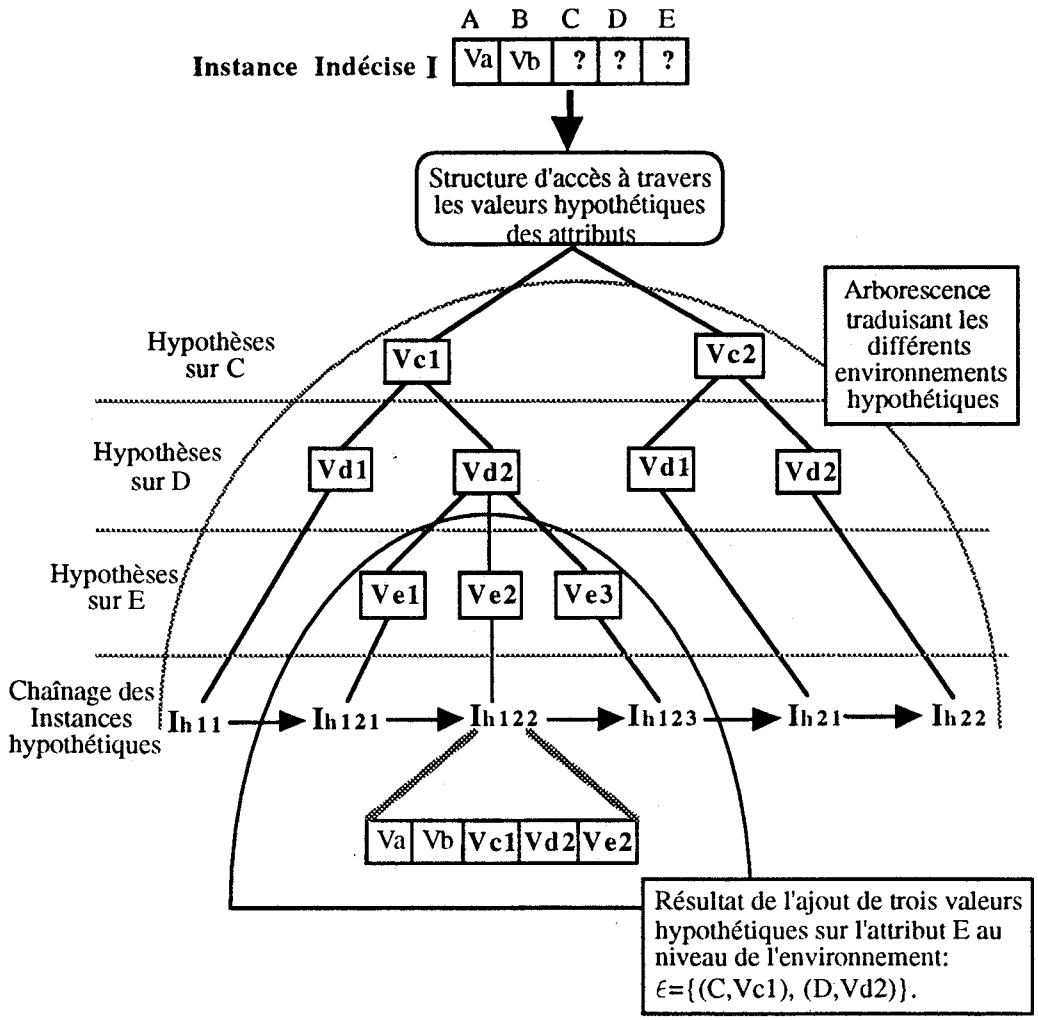

Figure III.9 : Mise en évidence des environnements hypothétiques sous-jacents. L'instance indécise I fait l'interface entre le bassin hypothétiques et le système de raisonnement. Elle possède les informations nécessaires au niveau de ses attributs pour coder cette structure logique d'environnements.

Si la notion d'environnement n'est pas explicite dans la structure des instances hypothétiques, l'ensemble des informations associées aux attributs hypothétiques permet de calculer à partir de la donnée d'un ensemble de paires (attribut hypothétique, valeur hypothétique), c'est-à-dire un environnement, la position des instances hypothétiques concernées dans le chaînage. Les calculs sont complexes, ils tirent parti de l'ordre dans lequel les attributs hypothétiques sont introduits, du nombre de valeurs hypothétiques par attribut... pour recréer dynamiquement une partie de l'arborescence des environnements construits. Les informations nécessaires aux calculs sont centralisées dans l'instance indécise.

Cette méthode est adaptée lorsque les hypothèses sont posées directement sur les attributs de l'instance indécise comme pour les attributs C et D de l'exemple, les environnements sous-jacents sont alors tous de même niveau. En effet, dans l'exemple de C et D, les hypothèses sont équivalentes à l'expression booléenne $(C=Vc1 \cup C=Vc2) \cap (D=Vd1 \cup D=Vd2)$, qui se réécrit en $(C=Vc1 \cap D=Vd1) \cup (C=Vc1 \cap D=Vd2) \cup (C=Vc2 \cap D=Vd1) \cup (C=Vc2 \cap D=Vd2)$. On retrouve alors quatre environnements : $E1 \cup E2 \cup E3 \cup E4$. Ils donnent naissance aux quatre instances hypothétiques respectives : Ih11, Ih12, Ih21 et Ih22.

Par contre, lorsque les hypothèses sont posées au niveau d'une instance hypothétique, donc d'un environnement particulier, comme pour l'attribut E de l'exemple, la gestion devient beaucoup plus complexe. Implicitement, il y a création d'environnements de niveaux inférieurs ; dans l'exemple E2 se voit attacher trois environnements correspondants aux valeur hypothétiques Ve1, Ve2 et Ve3. L'arborescence se déséquilibrant, les calculs de position d'instances pour un environnement perdent leur généralité et se compliquent.

On peut, de plus, s'étonner de voir disparaître l'instance hypothétique qui est à l'origine des émissions d'hypothèses. En effet, dans l'exemple, Ih12 est remplacée par les trois instances Ih121, Ih122 et Ih123. Cette instance ne joue-t-elle pas le même rôle que l'instance indécise pour les instances Ih11, Ih12, Ih21 et Ih22 ? Ce choix se justifie par le fait qu'une fois qu'un attribut s'est vu affecté un ensemble de valeurs hypothétiques, le choix devient exhaustif, c'est-à-dire que l'attribut aura nécessairement une de ces valeurs. Dans le cas contraire, l'instance hypothétique émettrice (Ih12) n'est pas recevable et peut être supprimée. Partant de cette supposition, il n'est effectivement pas nécessaire de la garder lors de la production de Ih121, Ih122 et Ih123.

Cette technique est par contre fâcheuse si l'on souhaite que de nouvelles hypothèses puissent être émises à partir de Ih12, par exemple sur un autre attribut que E, ou encore sur lui mais par d'autres moyens d'obtention.

La plupart de ces décisions sur la gestion d'instances hypothétiques sont déterminées par leur exploitation dans le mécanisme de classification, qui n'utilise qu'une instance hypothétique à la fois. L'exploration des hypothèses se fait alors en profondeur dans l'arborescence des environnements hypothétiques. Dans ce cas, il serait même préférable de ne produire des instances hypothétiques qu'au fur et à mesure des besoins, une à une.

Cette gestion d'instances hypothétiques perd de sa généralité de par la contrainte d'exhaustivité sur l'ensemble des valeurs hypothétiques d'un attribut, et de sa souplesse par la structuration plate des instances hypothétiques. La principale raison de ses choix provient de présupposés sur la production et l'exploitation des hypothèses. Ceux-ci consistent essentiellement à soutenir un raisonnement de classification d'instance incomplète au cours duquel les seules possibilités d'émission d'hypothèse sont fournies par les moyens d'obtentions de valeurs d'attributs, extérieurs au processus de classification.

5.4. Proposition d'une nouvelle approche du bassin hypothétique

Cette partie propose une nouvelle approche de la modélisation du bassin hypothétique visant à généraliser le travail présenté précédemment (cf. §III.5.3). En effet, un bassin hypothétique associé à une instance est perçu comme un ensemble d'unités de connaissances, les **versions hypothétiques de l'instance**, structuré par la notion d'environnement inhérente à toute gestion d'hypothèses. Outre le fait que cette notion reflète fidèlement la progression d'un raisonnement hypothétique, elle permet une représentation factorisée de l'ensemble des instances hypothétiques d'un bassin puisqu'elle prend en compte les connaissances partagées par ces instances.

5.4.1. Représentation de l'évolution multiple d'une instance

L'intégration de la notion d'hypothèse au sein du modèle à objets, présentée dans cette partie, exploite une structuration qui s'inspire des treillis de contextes décrits dans l'étude de l'ATMS. Cette structuration est basée sur l'ordre partiel induit par l'inclusion ensembliste des environnements (ensemble d'hypothèses).

De ce fait, le principe général est de représenter l'ensemble des états d'une instance en regroupant les connaissances en fonction des environnements hypothétiques construits. Ainsi, à chaque environnement est associé un contexte qui fournit l'état de connaissances de l'instance dérivable à partir de cet environnement (cf. Figure III.10).

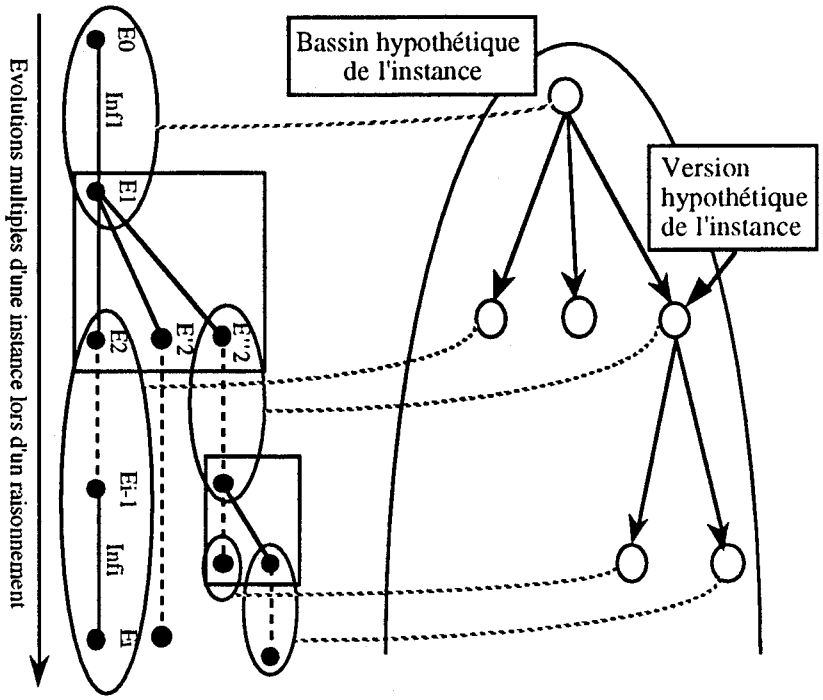

Figure III.10 : La notion de version hypothétique est le support d'un raisonnement développant une instance à partir de connaissances sûres et hypothétiques. L'ensemble des versions d'une instance, son bassin hypothétique, suit l'évolution multiple de l'instance.

L'association entre un environnement et un contexte prend naissance au sein d'une entité appelée **version hypothétique**. De plus, l'ensemble des versions hypothétiques d'une instance, structuré par la relation d'ordre partiel sur les environnements, constitue le **bassin hypothétique** de l'instance.

5.4.2. Dimension hypothétique d'une instance

Bien qu'un système de type ATMS ne gère que de façon implicite la notion de contexte, il n'en reste pas moins que le résultat de cette gestion revient à la mise en place d'un mécanisme d'héritage entre contextes. En effet, partant du principe que toute connaissance visible à partir d'un environnement est aussi visible à partir de tout environnement englobant, l'ATMS gère chaque connaissance en l'étiquetant avec le(s) plus petit(s) environnement(s) à partir duquel elle est visible. Un tel environnement est alors dit minimal pour cette connaissance. De ce fait, la donnée d'un environnement permet de restituer le contexte correspondant en sélectionnant toutes les connaissances étiquetées par un environnement minimal qui est inclus dans l'environnement désigné. A travers cette gestion d'étiquette, les connaissances sont partagées par tous les contextes auxquels elles appartiennent.

Aussi, l'ensemble de versions étant structuré par la relation d'ordre entre environnements, la gestion d'un bassin hypothétique peut exploiter ce principe de visibilité pour adopter une représentation factorisée et mettre en place un mécanisme d'héritage entre versions. Sa réalisation est basée sur l'ajout d'une nouvelle relation, appelée **version-de**, qui est mise en place à chaque fois qu'une version est construite par adjonction d'hypothèses à l'environnement d'une autre.

Une version devant restituer l'état de connaissances de l'instance dans un environnement hypothétique particulier, cet état est alors obtenu par héritage des connaissances dans la hiérarchie des versions. Aussi, pour reprendre la métaphore spatiale d'Amedeo Napoli [Napo92], l'ajout de la relation d'ordre partiel **version-de** introduit une nouvelle dimension dans la base de connaissances : la **dimension hypothétique** (cf. Figure III.11).

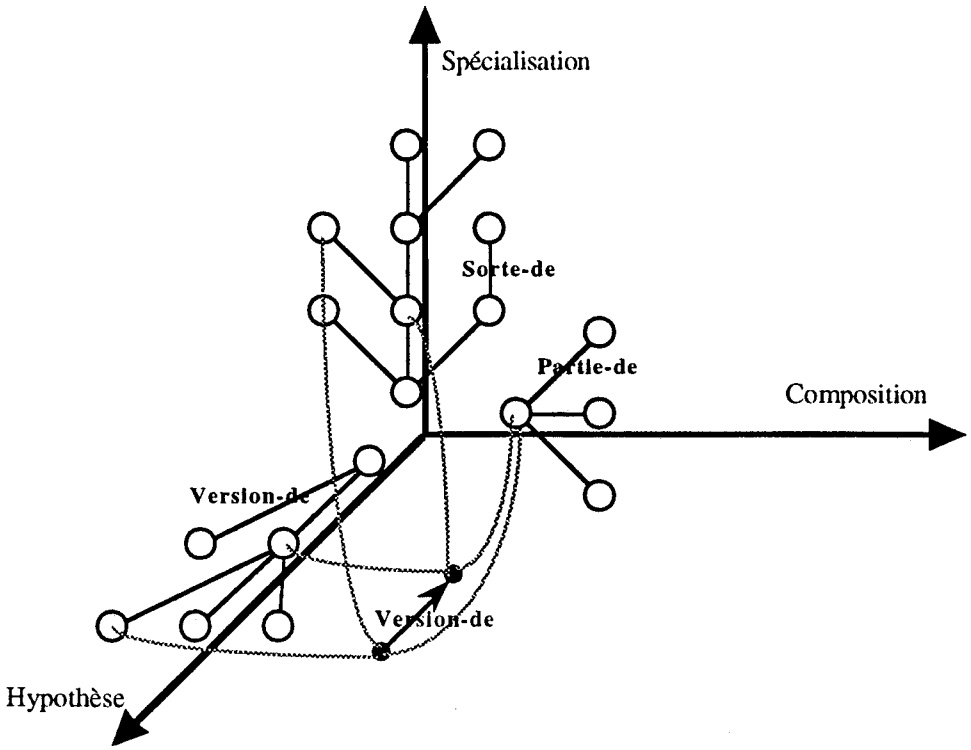

Figure III.11 : La hiérarchie de versions hypothétiques d'une instance, matérialisée par la relation **version-de**, vient ajouter la dimension hypothétique à l'espace des objets comprenant déjà les dimensions de spécialisation et de composition.

La partie suivante décrit la notion de bassin hypothétique, la relation qu'entretient une instance avec ce bassin, l'information portée par chaque version et les règles de partage d'informations entre une version et sa "sur-version".

5.4.3. Instance, version et bassin hypothétique

Le principe adopté consiste à séparer la notion d'instance de celle d'hypothèse, et de voir l'instance comme une entité évoluant dans la dimension hypothétique (cf. Figure III.12).

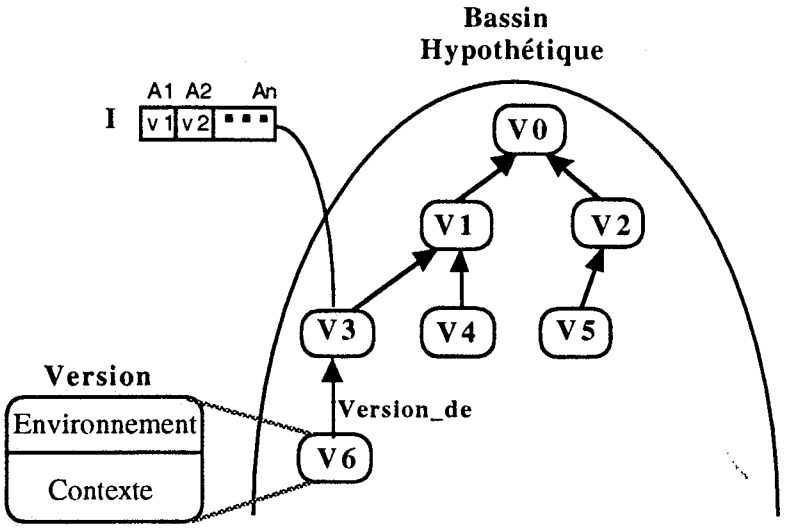

Figure III.12 : Le bassin hypothétique est une hiérarchie de **versions** organisée par la relation **version-de**. Quand l'instance est liée à une version elle épouse le modèle fourni par la version.

Ainsi, l'instance se voit lier dans son bassin hypothétique à la version correspondant à l'environnement courant du raisonnement. Comme dans une hiérarchie de classes, l'instance évolue dans la hiérarchie de versions en fonction des besoins du raisonnement hypothétique. Une instance est l'interface entre le raisonnement et le bassin hypothétique, toute modification de l'instance a des répercussions sur le bassin à partir de la version à laquelle elle est actuellement liée. Notamment, l'introduction de nouvelles hypothèses déclenche la création de nouvelles versions.

5.4.3.1. Structure d'une instance

La structure d'une instance est enrichie d'une information concernant la version à laquelle elle est liée dans le bassin hypothétique. Les informations portées par une instance sont :

- **Son lien de rattachement** : la plus petite classe d'appartenance de l'instance dans la hiérarchie de classes.
- **Sa version courante** : Si cette information n'est pas spécifiée, l'instance n'est pas hypothétique et ne possède donc pas de bassin hypothétique.
- **L'ensemble de ses attributs** : Si l'instance est liée à une version, l'information concernant ses attributs est restituée à partir de la version.

De cette façon, une structure d'instance ne diffère de celle d'une instance classique que par l'ajout d'un lien avec sa version courante.

5.4.3.2. Structure d'une version

Toutefois, les grandes lignes de cette structure sont données par trois types d'informations : celles propres à la version, celles concernant l'environnement, et celles concernant le contexte.

- Informations **propres à la version** :
  - **Son lien avec sa sur-version** : désignant la version du niveau supérieur à laquelle elle est liée par la relation **version-de**. Si le lien est **nil**, la version est la racine de la hiérarchie.
  - **Sa descendance** : c'est-à-dire les liens désignant l'ensemble des versions qui sont des sous-versions directes de cette version. Cela correspond donc aux liens inverses de **Version-de**. Si la descendance est vide, la version est une feuille de la hiérarchie de versions.
- Informations concernant l'**environnement** : ces informations fournissent les hypothèses qui ont donné lieu à la création de la version et se distinguent de celles de sa sur-version. Ces informations peuvent concerner le rattachement de l'instance à une classe, ou tout type d'hypothèse pouvant être émise sur un attribut.
- Informations concernant le **contexte** : celles-ci représentent les nouvelles connaissances acquises sur l'instance à partir de l'environnement courant de cette version.

5.4.3.3. Rattachement d'une instance à une version

Le rattachement de l'instance à une version déclenche un processus de récupération des informations permettant de restituer l'instance dans l'état représenté par cette version. A tout moment de son évolution hypothétique, l'instance centralise l'ensemble des informations héritées à travers la hiérarchie de versions, c'est-à-dire le long du chemin liant la version courante à la version racine du bassin hypothétique. L'ensemble des versions composant ce chemin représente les raffinements de connaissances successifs par lesquels l'instance est passée pour aboutir à son état courant. Aussi, l'héritage doit sélectionner dans la hiérarchie de versions les informations les plus raffinées.

La mise en place de l'héritage exploite un principe d'étiquetage de chaque connaissance de l'instance avec la référence à la version dont elle provient. Cet étiquetage des informations portées par une instance est alors établi dynamiquement en fonction de la progression de l'instance dans le bassin hypothétique (cf. Figure III.13).

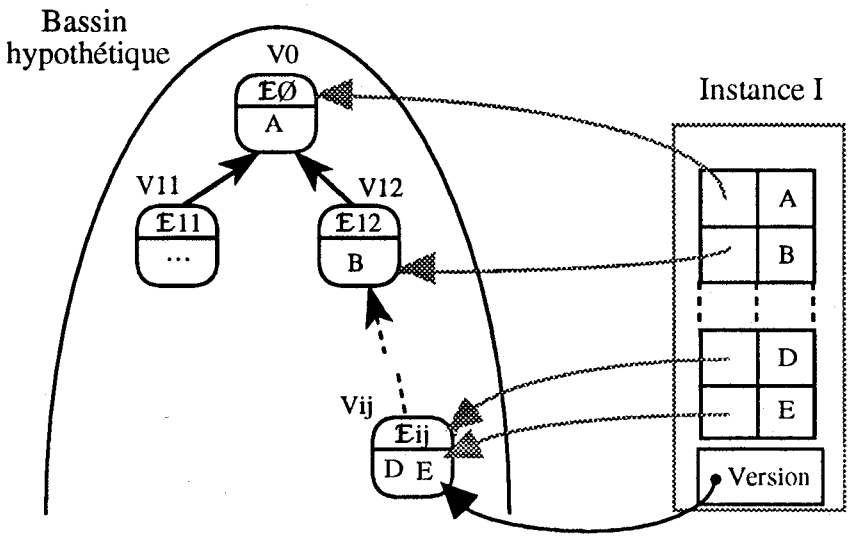

Figure III.13 : Mise en place au niveau de l'instance de l'étiquetage des éléments de connaissance en fonction de leur origine dans la hiérarchie de versions. Cette annotation systématique peut être vue comme une trace de l'héritage des connaissances à travers le bassin hypothétique.

L'héritage bénéficie en retour de cet étiquetage pour mettre à jour les connaissances de l'instance en fonction de ses déplacements dans la hiérarchie. En effet, lorsque l'instance progresse d'une version à une autre dans le bassin, deux opérations sont possibles :

- éliminer les connaissances de l'instance qui ne sont pas attachées à la nouvelle version ou à l'une de ses sur-versions, et
- ajouter toutes les informations des versions en remontant à partir de la nouvelle version courante dans la hiérarchie de versions jusqu'à atteindre une version déjà prise en compte dans l'instance.

Techniquement, l'annotation des connaissances dans l'instance consiste simplement en une réutilisation des liens de justification de chacun des éléments de représentation de l'instance. En effet, chacun d'eux comporte dans sa justification la référence à la version du bassin dont il provient. En contre-partie, le bassin contient dans les versions toutes les informations concernant les justifications réelles (description des inférences) de chaque élément. Cette indirection, transparente pour l'instance, évite une redondance entre les versions et l'instance des informations gérées par le système de maintien du raisonnement.

6. Conclusion

Le problème des connaissances incomplètes peut se présenter de différentes façons : informations oubliées, informations implicites ou informations inconnues. Si la première requiert l'intervention d'un utilisateur, les deux secondes correspondent à deux types de raisonnement, non monotone et hypothétique. Ces deux types de raisonnements permettent d'étendre un raisonnement de type déductif à un raisonnement sur des connaissances incomplètes. Ils nécessitent par contre la mise en place d'un système de maintien du raisonnement, de type TMS pour le premier et de type ATMS pour le second.

Dans le contexte des représentations par objets, il est intéressant de bénéficier de telles techniques pour, d'une part, gérer les modifications de la base et, d'autre part, introduire la notion d'hypothèse au sein des objets. Ce dernier point semble une voie d'extension

prometteuse d'un raisonnement sur les objets. En effet, la possibilité d'émettre des hypothèses permet de poursuivre un raisonnement pour explorer plusieurs solutions simultanément. Les systèmes ART, KEE et OMEGA adoptent une telle approche et permettent de gérer plusieurs mondes hypothétiques d'objets suivant un principe proche de l'ATMS.

Le travail sur TROPES montre l'intérêt d'une notion d'instance hypothétique. Son application à la classification permet d'étendre ce mécanisme au cas d'indécision sur les valeurs d'attributs qui peuvent être inférées. Le caractère particulier des hypothèses dont l'impact est localisé à la notion d'instance distingue le gestionnaire d'hypothèses proposé d'une solution de type ATMS. La notion de contexte étant réduite à celle d'instance hypothétique, la gestion des étiquettes ne s'avère plus nécessaire et se traduit naturellement par la possibilité d'exister ou non d'une instance affectée par la présence de valeurs d'attributs hypothétiques.

Ce choix est, de plus, étayé par le fait qu'un système de maintien du raisonnement de type TMS est plus approprié et déjà mis en place pour prendre en compte l'utilisation non monotone d'une base de connaissances par objets. La gestion de la cohérence et la gestion d'hypothèses sont donc deux problèmes traités orthogonalement, un des résultats attendus de ce principe étant de pouvoir assurer aussi les mises à jour sur un bassin hypothétique.

Si l'étude de l'intégration de la gestion d'hypothèses au sein d'une base d'objets est menée de manière générale, la solution proposée est, quant à elle, spécifique à l'application des instances hypothétiques au raisonnement de classification en profondeur. Les conséquences en sont que la structuration des hypothèses adoptée devient difficile et lourde à exploiter dès lors que le bassin hypothétique doit refléter des dépendances entre hypothèses. C'est notamment le cas lorsque l'instance indécise n'est plus la cible privilégiée de production d'hypothèses mais que des calculs d'attributs émettent des valeurs hypothétiques à partir d'une instance elle-même hypothétique.

Ce problème provient essentiellement de la perte de la notion d'environnement qui constitue l'unité logique de structuration des contextes de l'ATMS. C'est donc vers une solution se basant sur cette notion qu'un gestionnaire d'instances hypothétiques retrouvera sa généralité, et permettra de rendre compte d'un raisonnement de production d'hypothèses plus riche.

La proposition d'introduire la notion de version hypothétique comme unité de représentation du bassin hypothétique d'une instance répond à ce besoin de généralité. Pour soutenir le mécanisme de complétion d'instance qui sera proposé par la suite, un système de gestion de versions hypothétiques est utilisé.

Partie 2

IDENTIFICATION ET CONSTRUCTION D'OBJETS

Chapitre IV
LA DUALITE
IDENTIFICATION/CONSTRUCTION D'OBJETS

1. Introduction

Les mécanismes de classification d'instance et d'inférence de valeurs d'attributs suggèrent l'exploitation de la hiérarchie de classes à des fins de construction d'instances guidée par des données d'entrée. Alors qu'un certain nombre d'attributs (déterminants) pourraient être exploités pour effectuer une classification, d'autres (déterminés) pourraient être calculés en fonction du contexte des classes sélectionnées durant la phase de classification. Cette distinction entre attributs **déterminants** et **déterminés** laisse penser que la simple donnée des premiers permet le calcul des seconds et offre, par là même, un mécanisme pour compléter une instance partiellement valuée. Toutefois, cette distinction est souvent illusoire puisque tout attribut dans la définition d'une classe est généralement considéré comme déterminant pour la classification. Par contre, lors du rattachement d'une instance à une classe, cette distinction entre attributs déterminants et déterminés est tout à fait acceptable puisque l'instance est explicitement déclarée comme appartenant à la classe. Les attributs déterminés sont alors calculés suivant la description de la classe et en fonction des attributs déjà évalués. La classe présente donc un double visage suivant son exploitation : classification ou rattachement.

Cette partie s'intéresse à l'étude d'un mécanisme qui utilise la classification d'une instance en vue de la compléter. Pour cela, la notion d'instance incomplète est étudiée, et le double emploi de la notion de classes, modèle pour l'identification ou la gestion d'instances, est souligné.

2. Instance incomplète

La notion d'instance incomplète est abordée dans le contexte d'un système à objets muni d'un mécanisme de classification. Par souci de généralité, l'existence d'une instance est indépendante des classes (cf. §II.2.1). En revanche, son espace d'évolution est fourni par la notion de concept comme dans TROPES (cf. §II.4.1). Ainsi les informations de description des instances peuvent être locales à une classe, ou globale à un concept.

2.1. Notion d'instance incomplète

Une instance I appartenant à un concept Ω est dite **incomplète** lorsque cette instance possède au moins une valeur d'attribut inconnue ("?"), l'attribut étant tout de même reconnu comme une propriété de l'instance I. Une instance peut obtenir le statut "incomplet" soit de façon directe par la donnée explicite d'un attribut de valeur inconnue, soit de façon indirecte lors du rattachement à une classe.

Soit $\mathcal{A}_\Omega$={ A, B, C, E, F...} l'ensemble de tous les attributs que peut posséder une instance du concept Ω. Une instance de Ω ne possède pas nécessairement tous les attributs de $\mathcal{A}_\Omega$.

Dans la base de connaissances, l'état de connaissances d'une instance I de Ω peut être précisé en deux étapes : la donnée de l'ensemble des attributs de $\mathcal{A}_\Omega$ qui sont connus pour être des attributs de I, puis pour chacun de ces attributs la donnée d'une valeur ou non.

Par la suite, l'expression $I\{(A_1, V_{A1}) ; \ldots ; (A_i, V_{Ai})\}$ décrit l'état de connaissances associé à l'instance I en précisant les couples attributs-valeurs connus pour I. Par exemple, l'expression $I\{(A, V_A), (B, V_B), (C, V_C), (D, V_D)\}$ décrit l'état de connaissances de I en

précisant que I possède les attributs A, B, C et D avec les valeurs respectives V_A, V_B, V_C et V_D. De plus nous notons, **C**(A, B, C, D...) le fait qu'une classe **C** de Ω spécifie les attributs A, B, C, D... dans sa définition.

Avec ces notations, nous avons alors :

- l'instance I est incomplète dans l'**absolu** lorsqu'au moins une des valeurs d'attributs est égale à la valeur indéfinie "?".

  Par exemple, I{(A, V_A), (B, V_B), (C, ?), (D, V_D)} indique que l'instance I est incomplète car la valeur de l'attribut C est inconnue. L'instance I est dite incomplète dans l'**absolu**.

- l'instance I est **incomplète relativement à la définition de la classe C** lorsque l'appariement de I à la classe **C** indique que **C** comporte au moins un attribut que ne possède pas l'instance I. Si I est rattachée à **C**, I devient aussi incomplète dans l'absolu puisque le rattachement impose l'introduction d'attributs dans I dont la valeur est inconnue.

  Par exemple, si l'instance I est donnée par I{(A, V_A), (B, V_B), (C, V_C)} est appariée à la classe **C** définie par **C**(A, B, C, D) alors I est incomplète relativement à **C** puisque D n'apparaît pas dans I. Si I est rattachée à **C** alors son état est donné par I{(A, V_A), (B, V_B), (C, V_C), (D, ?)}.

Il est à remarquer qu'une instance peut être complète relativement à une classe et être cependant incomplète dans l'absolu. Par exemple, si l'instance incomplète I((A, V_A), (B, V_B), (C, ?), (D, V_D)) appartient à la classe **C**(A, B, D), l'instance I est, par contre, complète vis-à-vis de la classe **C**.

S'il est possible de donner le statut incomplet à une instance, le statut complet pose, par contre, des problèmes. En effet, à moins de définir explicitement quels sont les attributs que doit posséder une instance d'un concept pour être complète (et ainsi circonscrire le concept à un ensemble fini d'attributs), il est impossible de donner objectivement le statut complet à une instance. En revanche, une classe définit exactement l'ensemble des attributs que doit comporter au minimum une instance lui appartennant. C'est pourquoi le statut complet n'est utilisé que relativement à la définition de classe.

Que ce soit relativement à une classe ou dans l'absolu, le statut incomplet d'une instance est important puisqu'il permet de focaliser l'attention sur un manque d'information spécifique : la valeur d'un attribut particulier. La détection d'un tel cas peut être interprétée comme une demande d'obtention de la valeur d'attribut. Dans ce cas, la nuance entre les statuts "incomplet dans l'absolu" et "incomplet relativement à une classe" est importante. En effet, le statut de relativité désigne une classe dont la description peut fournir les informations nécessaires à l'inférence de la valeur d'attribut manquante. Ces informations pourront être exploitées si l'instance est connue comme appartenant à la classe. Quant à lui, le cas d'un attribut déclaré uniquement incomplet dans l'absolu ne peut se traiter qu'à partir de moyens d'inférence globaux au concept de définition de l'instance.

2.2. Classification d'instance incomplète

La classification d'une instance est basée sur l'appariement de cette instance avec les classes de la hiérarchie. Cet appariement teste l'adéquation entre les informations portées par l'instance et celles décrites par une classe. Aussi, ce mécanisme d'appariement est à même de détecter les cas où l'instance est incomplète relativement à une classe. Cette incomplétion de l'instance n'est pas significative quand la classe est rejetée (marquée impossible) puisqu'une contradiction entre les valeurs d'attributs de l'instance et les descriptions d'attributs de la classe est rencontrée ; et qu'en conséquence l'instance ne peut pas appartenir à la classe. Par contre, si aucune contradiction n'est trouvée, l'appartenance est donc possible, et l'instance est incomplète relativement à cette classe. Cette incomplétion de l'instance ne peut être affirmée que si celle-ci est connue comme appartenant à cette classe, ou simplement si elle est déjà incomplète dans l'absolu. Par conséquent, la classification d'une instance dans une hiérarchie de classes met en évidence les cas où l'instance pourrait être incomplète relativement à la définition d'une

classe. Ces cas sont donnés par l'ensemble des classes déclarées comme possibles par la classification.

Par exemple, l'appariement de l'instance I((A, V_A), (B, V_B)) à la classe C(A, B, C), en supposant que V_A et V_B sont valides pour la classe C, donnera le résultat possible, avec l'explication "si I appartient à C alors I est incomplète relativement à C car l'attribut C est inconnu dans I". On peut noter une nuance dans le résultat de cet appariement si I avait été donnée par I((A, V_A), (B, V_B), (C, ?)), étant incomplète dans l'absolu, l'explication aurait été alors "si I appartient à C alors I est incomplète relativement à C car la **valeur** de l'attribut C est inconnue dans I".

La classification peut donc s'avérer un support intéressant pour guider l'acquisition des valeurs d'attributs d'une instance. En effet, les attributs sont sélectionnés en fonction de la progression de la classification et donc de la hiérarchie de classes. C'est le cas dans SHIRKA qui offre un mécanisme de classification d'instance. Celui-ci consiste à parcourir la hiérarchie de classes de façon incrémentale, des classes les plus générales aux classes les plus spécifiques suivant le lien de spécialisation. A chaque étape, la classification marque comme possibles, sûres, ou impossibles les sous-classes directes de la classe où est rattachée l'instance considérée, et propose comme nouvelles classes de rattachement de l'instance celles qui sont soit possibles, soit sûres (cf. §II.3.2). Le choix d'une classe possible implique que l'instance soit alors incomplète relativement à celle-ci ; les valeurs d'attributs manquantes sont alors inférées ou demandées à l'utilisateur.

La figure IV.1 montre l'ordonnancement des demandes de valeurs d'attributs suivant la progression de la descente effectuée lors de la classification d'une instance dans la hiérarchie de classes. A chaque niveau, l'instance est considérée comme incomplète par rapport aux classes déclarées possibles, la demande des valeurs d'attributs inconnues est déclenchée. Par exemple, l'instance I, qui possède initialement deux attributs A et B ayant pour valeur respectivement V_A et V_B, se voit complétée progressivement lors de la classification. Au deuxième niveau, la classe C1 est déclarée impossible alors que C2 est déclarée possible. L'attribut D n'étant pas compris dans la spécification de I, une requête est émise (soit à l'utilisateur, soit à un mécanisme d'inférence) pour obtenir une valeur de cet attribut pour l'instance I. La valeur V_D obtenue en retour de cette requête satisfait la description de la classe C2 qui est alors déclarée sûre, la classification peut alors être poursuivie. L'appariement à la classe C3 permet d'obtenir la valeur V_E de l'attribut E pour l'instance I suivant le même processus.

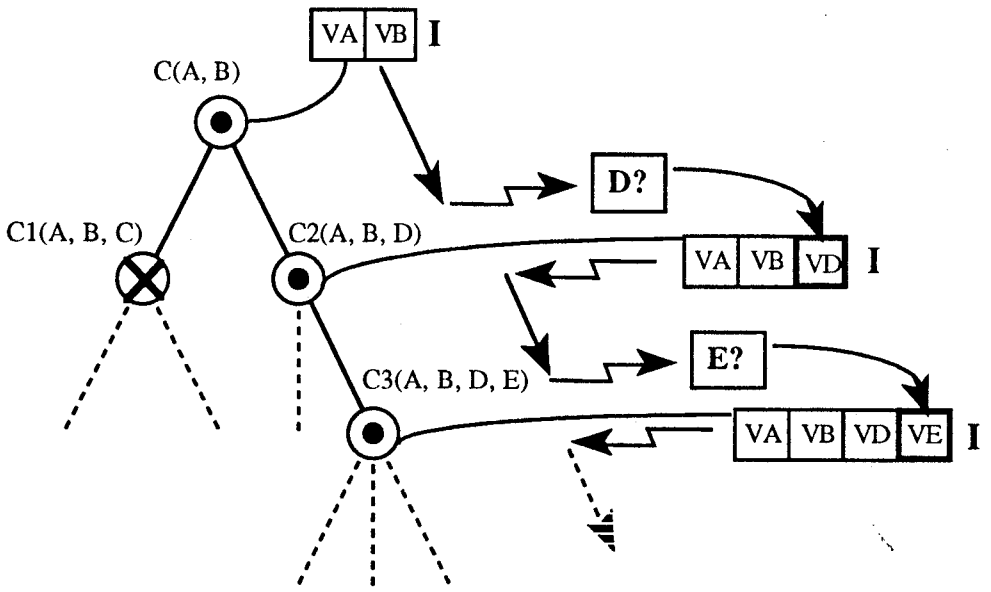

Figure IV.1 : Ordonnancement des demandes de valeur d'attribut en fonction de la classification de l'instance I. Au niveau de la classe C2 l'attribut D est demandé puis déterminé ; de même pour l'attribut E au niveau de la classe C3.

L'acquisition progressive d'informations sur une instance offerte par un contexte classificatoire montre que la classification propose un cadre adéquat pour la construction d'objet, et dépasse donc le simple raisonnement de reconnaissance. Toutefois, à l'image de SHIRKA, si la classification permet de focaliser l'attention sur l'information manquante, elle nécessite l'intervention d'un agent extérieur pour l'obtention de cette information. Sinon le processus déductif que la classification met en œuvre se voit bloqué (cf. Figure IV.2).

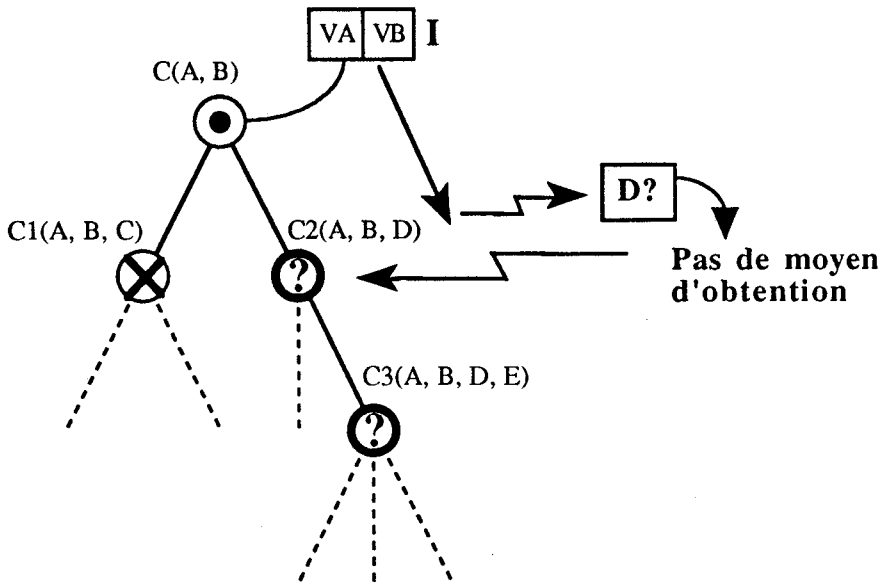

Figure IV.2 : Lors de la classification de l'instance I, l'attribut D est inconnu pour cette instance. Puisqu'il n'existe pas de moyen d'obtention de la valeur de D pour I, le processus de classification est bloqué. La classe C2 est déclarée possible.

Ainsi présentée la classification d'instance incomplète ne se différencie de la classification usuelle d'instance que par sa coopération avec un agent extérieur pour demander des valeurs d'attributs (cf. Figure IV.3). La hiérarchie de classes n'est pas exploitée pour la construction de l'instance ; c'est-à-dire qu'une classe de la hiérarchie n'est utilisée que comme identifiant et non pas comme un constructeur de l'instance.

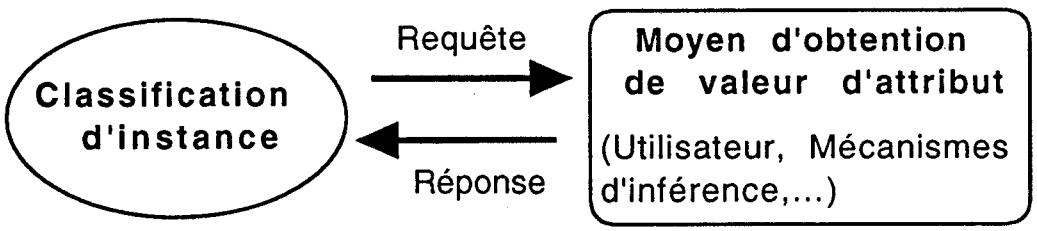

Figure IV.3 : Coopération entre le processus de classification d'une instance incomplète et un agent chargé de fournir la valeur d'un attribut manquant.

Toutefois, il est classique dans la représentation par objet de considérer la classe comme un moule sur lequel est formée toute instance lui appartenant. Ainsi, au sein de la description d'une classe peuvent être décrits les moyens d'obtention d'une valeur d'attribut qui seront utilisés lorsque cette valeur sera manquante dans le contexte d'une instance. Ces moyens d'obtention s'expriment en fonction des informations définies dans le contexte de cette classe (et de ses sur-classes par héritage).

Cette propriété de la classe n'est pas exploitée lors de la classification mais lors du rattachement de l'instance à la classe. Nous étudions par la suite les deux rôles possibles d'une classe afin d'en dégager les caractéristiques et d'établir un compromis entre ces deux rôles.

3. Double rôle d'une classe

Un grand nombre de raisonnements dépassent le cadre simple de la reconnaissance d'une instance complète. En effet, les raisonnements sont souvent plus complexes et alternent les phases de reconnaissance d'objets et de production d'information sur ces objets (l'une pouvant entraîner l'autre, et vice versa), c'est pourquoi il est intéressant d'exploiter la description des classes non seulement pour reconnaître l'instance mais aussi pour la compléter. Le langage utilisé peut offrir les moyens d'indiquer comment inférer des valeurs. Il y a deux grands courants d'idées qui dépendent essentiellement de l'importance qui est donnée au rôle d'une classe : unité d'identification ou de gestion d'instances.

3.1. La classe, modèle pour l'identification

Les systèmes qui favorisent la capacité d'identification de la classe appuient leur raisonnement classificatoire sur le fait que les informations associées à une classe représentent la condition suffisante (et les conditions nécessaires) d'appartenance à la classe. Pour la classification, une classe représente une règle de définition aristotélicienne[1] :

$\forall$ I, $P_1(I) \wedge P_2(I) \wedge P_3(I)... \rightarrow \mathbf{C}(I)$, où P_1, P_2, P_3... représentent les propriétés (les attributs et les contraintes sur ces attributs) de l'instance I, et $\mathbf{C}(I)$ signifie $I \in \mathbf{C}$, $\mathbf{C}$ étant la classe définie par les propriétés P_i.

3.1.1. Langages terminologiques

C'est le cas de la famille des langages terminologiques descendants de KL-ONE [Brac&85]. Tous ces langages s'articulent autour des notions de **concept** (**primitif** ou **défini**) et de **rôle**, et fournissent comme mécanismes la reconnaissance d'individus (classification d'instances) et la classification de concepts (classifications de classes) (cf. §I.3.2.3).

Ainsi, fidèle aux idées de Ronald J. Brachman, notamment en ce qui concerne les défauts ou exceptions [Brac85], la plupart des langages terminologiques fournissent un langage réduit de description de classes (*concepts*) par comparaison à ce qui se fait dans les autres langages de *frames*. L'expression d'une classe est principalement tournée vers les mécanismes de classification et de subsomption et l'effort est porté sur l'efficacité de ces mécanismes ainsi que leur complétude et leur correction. Le système CLASSIC est un bon exemple des concessions qui ont été faites sur l'expressivité du langage pour conserver dans la mesure du possible ces propriétés [Brac92]. Notamment la description d'interdépendance entre rôles (attributs) dans un même *concept* est réduite à l'égalité entre ces rôles (constructeur **SAME-AS**) ; sous la contrainte supplémentaire que ces rôles soient monovalués, ils sont alors appelés *attributs*.

D'autres langages terminologiques comme NIKL [Kacz&86], LOOM [MacG&91] et BACK [Nebe88] préférent augmenter le pouvoir d'expression du langage et sacrifier la complétude. Ce choix est souvent guidé par des considérations pragmatiques qui rendent compte d'utilisations réelles.

Ainsi, contrairement aux possibilités offertes par l'attachement procédural ou les valeurs par défaut à l'intérieur d'une description de classe d'un langage de *frames* classique, la description d'un *concept* dans les langages terminologiques n'offre pas de grandes possibilités d'inférences de valeurs d'attributs (*rôle*). Un *concept* étant défini par des conditions nécessaires et suffisantes, la classification d'une instance (individu, ou *concept individuel*) incomplète ne peut en aucune façon utiliser la description d'un *concept* pour compléter l'instance sans avoir établi auparavant l'appartenance de celle-ci au *concept* ; c'est-à-dire que l'instance doit être

---

[1] On trouvera dans [Hart&91] les définitions de quatre types de règles: contraintes sur les types, définition aristotélicienne, définition contingente ou schématique, et causalité.

complète relativement à ce *concept* et vérifier ses contraintes. Un tel système est donc impuissant face au problème de l'instance incomplète.

Toutefois, pour permettre d'inférer de nouvelles valeurs d'attributs (*rôle*) lors de la classification d'une instance, le problème est résolu en introduisant dans le modèle la notion de **règle** (CLASSIC [Brac&91]), ou **implication** (LOOM), qui se déclenche dès lors qu'une condition est vérifiée par l'instance. Cette condition peut être l'appartenance à une classe (*concept*), le fait déduit peut-être la valeur d'un attribut (*rôle*). Ce mécanisme de règles permet la déduction de nouvelles connaissances en coopération avec le processus de classification.

L'exemple suivant, repris de [Brac&91], est une règle de CLASSIC :

assert-rule [Vin-Chardonnay (FILLS couleur blanc)]

Cette règle est utilisée à chaque fois qu'un individu est déclaré comme appartenant au *concept* de vin **Chardonnay**, son *rôle* **couleur** est alors évalué comme **blanc**. Ce qui peut permettre à la classification de reprendre et d'établir que l'instance considérée qui est un Chardonnay appartient aussi au *concept* **Vin-Blanc** (Une bonne définition du vin blanc étant **Vin** avec le *rôle* **couleur** fixé à **blanc**).

3.1.2. Modèle TROPES

En dehors des langages terminologiques, le modèle TROPES, qui propose un mécanisme de classification automatique [Mari&90], considère aussi les contraintes définissant une classe comme des conditions nécessaires et suffisantes (cf. §II.4). Pour inférer des valeurs d'attributs lors de la classification, l'algorithme de classification tire parti d'une structuration multi-points de vue et de la notion de passerelle entre classes de deux points de vue différents. En effet, l'introduction d'une telle passerelle peut jouer un rôle similaire aux règles des langages terminologiques. Une passerelle est déclarée comme ayant une classe source et une classe destination ; elle traduit le fait qu'une instance appartenant à la classe source appartient à la classe destination. Dans ce cas si l'appartenance à la classe source est établie, l'instance bénéficie des informations portées par la classe destination, notamment sur la valeur de certains attributs (cf. Figure IV.4).

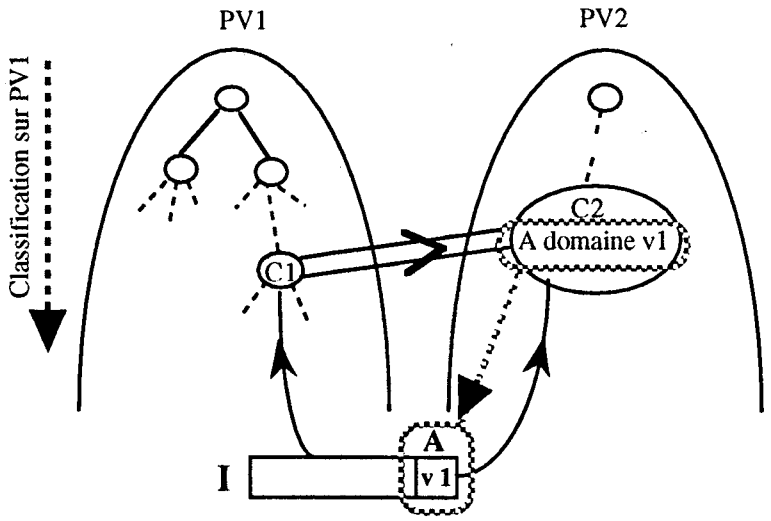

Figure IV.4 : Exemple d'inférence d'une valeur d'attribut lors d'une classification sur un point de vue dans le modèle TROPES. L'instance I est classée sur le point de vue PV1, l'appartenance à la classe C1 a été établie. Une passerelle indique que toute instance de la classe C2 du point de vue PV2 appartient aussi à la classe C1 du point de vue PV1. L'attribut A non déterminant dans le PV1 et ne possédant pas de valeur dans l'instance I, peut être inféré car son domaine dans C2 est réduit à la valeur v1, cette valeur est ajoutée à l'instance I.

Pour compléter une instance, le "détachement" procédural a été proposé comme extension du modèle TROPES (cf. §II.3.4.3). Celui-ci permet des calculs de valeurs d'attributs plus complexes mais ne permet pas a contrario de décrire explicitement le lien entre l'appartenance d'une instance à une classe et le calcul d'un attribut.

Ce mécanisme est adapté lorsque le calcul d'un attribut n'est pas dépendant de la classe d'appartenance de l'instance à laquelle il est lié, mais plutôt quand il s'agit d'une information inhérente à l'instance. Une fois encore, à l'instar des règles des langages terminologiques, le "détachement" procédural offre un mécanisme de calcul d'attribut indépendant de la hiérarchie de classes utilisée lors de la classification.

3.1.3. Conclusion

Les modèles tels que les langages terminologiques et TROPES, dédiés à la classification, correspondent au schéma de coopération entre le processus de classification et celui d'obtention de valeur d'attribut (règles, "détachement" procédural, etc.) présenté dans la figure IV.3 de la partie §IV.2.2.

La solution d'adjoindre des mécanismes comme les règles ou le détachement procédural est une solution satisfaisante répondant au problème d'alternance de phases d'identification et de production de connaissances. Toutefois, elle prône la séparation de l'expression de la connaissance pour l'identification de celle pour la production. La description des classes s'appauvrit en conséquence, ce qui peut être coûteux lorsque le calcul de certains attributs dépend effectivement de l'appartenance à une classe.

En effet, en associant à une classe les moyens de calcul d'attribut qui lui sont propres, le système s'épargne le travail de recherche d'un moyen de calcul d'un tel attribut puisqu'il y a directement accès par le lien d'appartenance de l'instance à la classe. Dans le cas des modèles décrits ici, la sélection est dépendante du nombre et de l'organisation des moyens de calcul d'un attribut que chaque modèle adopte, ainsi que de l'état de l'instance.

Par contre, ce type de modèle souligne bien le fait que tout attribut participant à la définition d'une classe est déterminant lors de la classification, et qu'aucun d'eux ne peut prétendre être déterminé à l'aide des descriptions de classes explorées lors du processus de classification. C'est pourquoi, ces modèles préconisent l'utilisation d'informations contingentes (externes) à la hiérarchie de classification pour, en particulier, déterminer des valeurs d'attributs.

Enfin, il est à noter que dans une certaine mesure ces systèmes peuvent prétendre voir la classe comme un modèle pour la construction puisque les attributs décrits dans une classe sont des conditions nécessaires. Ainsi une instance attachée à une classe doit se plier à sa description. Cependant le caractère classificatoire de ces systèmes a considérablement réduit le pouvoir de description d'une classe pour la construction d'instance. En ce qui concerne les langages terminologiques, le manque d'expressivité a été sévèrement critiqué dans [Doyl&91] : "Restreindre la classification à des informations purement définitionnelles réduit considérablement son utilité dans des applications pratiques". Mais il semble que la tendance actuelle aille dans le sens de J. Doyle et R.S. Patil comme l'atteste le workshop sur les langages de subsomption de termes [Pate&90].

3.2. La classe, modèle pour la construction

L'utilisation de la classe comme moule de construction d'instance est certainement la plus répandue dans les modèles objets qui distinguent classes et instances. Cette exploitation de la classe passe par le rattachement explicite d'une instance à la classe, ou souvent encore l'instanciation. Par opposition aux modèles précédents qui cherchent à classer l'instance dans la hiérarchie de classes, la classe d'appartenance de l'instance est fixée. L'instance bénéficie alors des informations associées à cette classe.

Les langages de programmation à objets comme SMALLTALK-80 [Robs&81], C++ , EIFFEL..., sont des représentants typiques de ces modèles. Le mécanisme principal est l'héritage de structure et de méthode. Celui-ci est déclenché lors de la manipulation d'instance

(création, modification, interrogation, etc.). Une instance est liée irrémédiablement à sa classe de création : elle n'a pas la possibilité de migrer vers une autre classe.

> "La classe est l'entité conceptuelle qui décrit l'objet. Sa définition sert de modèle pour construire ses représentants physiques appelés *instances*. Une instance est donc un objet particulier qui est créé en respectant les plans de construction donnés par sa classe." [Masi&89]

Dans le domaine des représentations de connaissances, c'est souvent ainsi qu'est considérée la classe. C'est le cas des systèmes comme LOOPS, KEE, STROBE... qui s'inspirent largement des langages de programmation par objets [Stef&85]. C'est aussi le cas de nombreux langages de *frames* comme FRL [Robe&77], KRL [Bobr&77], RLL [Grei&80], etc.

L'exploitation commune de la classe se fait à travers une règle de définition contingente, ou schématique :

$\forall$ I, $\mathbf{C}(I) \rightarrow P_1(I) \wedge P_2(I) \wedge P_3(I)\ldots$, où P_1, P_2, P_3... représentent les propriétés (les attributs et les contraintes sur ces attributs) définies pour la classe $\mathbf{C}$ et que l'instance I doit posséder puisque $I \in \mathbf{C}$ (i.e. $\mathbf{C}(I)$).

Mais, plus qu'un cadre logique aussi simple, ce type de modèles propose de nombreuses extensions d'aspirations conceptuelles, et pragmatiques, telles que les idées développées par M. Minsky dans [Mins75] sur la notion de ***frame*** (cf. §I.3.1.2). L'expression d'une classe, ou d'un *frame*, s'enrichit substantiellement selon les différents modèles existants. Il serait ambitieux et inutile de vouloir donner une liste exhaustive des apports de chacun des systèmes à l'expression des objets, il est préférable d'en donner simplement les grandes lignes.

3.2.1. Les défauts

Déjà présente dans les premiers langages de *frames* comme FRL et KRL, la facette de défaut associée à un attribut dans une description de classe permet d'affecter dans une instance une valeur à cet attribut s'il n'en a pas. L'attachement procédural associé à la facette de réflexe *$si-besoin* d'un attribut peut être également vu comme un mécanisme par défaut puisqu'il permet de calculer la valeur de l'attribut lorsqu'elle est absente. C'est une généralisation de la facette *$défaut*.

L'introduction de ces facettes, et de la facette d'affectation de valeur *$valeur*, dans les schémas de classes les rend concurrentiels dans leur fonctionnalité. En effet, lors de l'accès en lecture à un attribut, le comportement de l'héritage doit être spécifié pour choisir quelle sera la facette utilisée pour déterminer la valeur de l'attribut. Plusieurs parcours du graphe d'héritage sont possibles suivant qu'on favorise l'une de ces facettes, et suivant la façon d'appliquer l'ordre de priorité des facettes : sur une classe puis sur ses sur-classes une par une, ou globalement sur la hiérarchie de classes. On trouvera une description de trois stratégies de parcours proposées par P. Winston dans [Masi&89]. Ces parcours sont appelés en I, en N et en Z. L'importance d'une facette sur l'autre dépendra donc du parcours utilisé.

Une conséquence directe de l'introduction de ce genre de facettes est la mise en place d'un raisonnement par défaut donc non-monotone. Ainsi, le défaut introduit la notion d'hypothèse au sein même de la description des objets pour offrir une réponse au problème d'instances incomplètes. Une base de connaissances est alors sujette à des révisions d'hypothèses qui peuvent la laisser dans un état incohérent. Il faut être capable d'en maintenir la cohérence pour en assurer le bon fonctionnement. Ceci peut-être fait en couplant au système de représentation un système de maintien du raisonnement (SMR) chargé d'éliminer toute inconsistance (cf. §III.3). On peut citer parmi les sytèmes qui intègrent un SMR : KRS [Marc86], KEE [Film88], SHIRKA/TMS (version de SHIRKA couplée avec un SMR) [Euze87].

Une autre façon de maintenir la cohérence de la base est de la laisser tout simplement à la charge du concepteur. Celui-ci doit inclure à l'aide des réflexes, qui contrôlent ajout et suppression de faits, les procédures de vérification et de gestion de la consistance de la base. Cette solution est peu satisfaisante dans le sens où un certain nombre d'opérations de gestion de

la cohérence ne sont pas dépendantes du domaine d'application mais plutôt de la sémantique des *frames* ; elles sont donc automatisables. Par contre, cette méthode peut être plus efficace puisqu'elle permet de ne se préoccuper que des objets susceptibles de produire des incohérences.

Enfin, il est à noter que la notion de défaut peut être utilisée non pas pour décrire une propriété typique d'une catégorie d'objets mais aussi comme un moyen d'initialisation d'attributs dans une instance. Cette valeur n'a plus alors le statut d'hypothèse mais caractérise plutôt la nature évolutive de l'attribut. Par exemple, les attributs décrivant la hauteur et largeur d'un objet **fenêtre** auront des valeurs par défaut permettant de créer un prototype de fenêtre. Ils évolueront par la suite en fonction de la taille de la fenêtre souhaitée.

3.2.2. Les connaissances procédurales

Dans [Wino75], T. Winograd propose l'attachement procédural comme juste milieu entre le déclaratif et le procédural. A l'image de la facette *$si-besoin*, les facettes *$si-ajout* et *$si-enleve* peuvent être associées à un attribut. La facette *$si-ajout* permet de déclencher la procédure qui lui est attachée, lorsqu'il y a modification de la valeur de l'attribut. De la même façon, la facette *$si-enleve* a le même comportement lors de la suppression de la valeur de l'attribut (cf. §II.2.2).

Cette intégration du procédural dans les *frames* a un impact direct sur le mode d'exploitation, on parle alors de **programmation dirigée par les accès** :

> "Contrairement aux classes, les frames n'ont pas de comportement propre et sont manipulés par des *primitives*. Cependant, des procédures peuvent être attachées localement aux attributs pour définir des méthodes de calcul, de gestion ou d'utilisation de ces derniers. Ces procédures traduisent des raisonnements locaux et décentralisés et se déclenchent lors de l'accès aux attributs. Toute action est donc la conséquence d'un accès aux données et la programmation est dite *dirigée par les accès*." [Masi&89]

Cette intégration de connaissances procédurales dans les descriptions de classes donne à un modèle de représentation de connaissances par objets une puissance d'expression semblable à celle des langages de programmation à objets.

Toutefois, comme le montre l'étude effectuée dans [Orsi90] sur l'attachement procédural, cette puissance est aussi source de confusion. En effet, il joue plusieurs rôles :

- *l'inférence de valeurs d'attributs*, où il décrit alors une propriété de l'objet ;
- *la production de nouveaux objets*, où il décrit une relation entre objets ;
- *l'introduction d'informations de contrôle*, ce qui permet de gérer les objets et de maintenir la cohérence de la base.

L'utilisation des procédures rend la connaissance qui y est codée inaccessible au système de représentation, et donc non exploitable par ses mécanismes de raisonnement. Cette utilisation doit être faite à bon escient pour conserver les avantages de la déclarativité ; c'est-à-dire que l'attachement procédural doit répondre à un manque d'expressivité du modèle de représentation et être utilisé en dernier recours.

3.2.3. Les relations

Alors que le passage des réseaux sémantiques aux langages de *frames* avait réduit les relations aux relations de spécialisation entre classes et d'appartenance d'une instance a une classe, de nombreux travaux s'appliquent à réintroduire cette notion au sein des descriptions d'objets. La raison principale est que, bien souvent, sous le couvert des descriptions d'attributs et à grands renforts d'attachements procéduraux [Forn&90], se cache la notion de relation, ce qui la rend difficile à exprimer et à gérer. En effet, différents rôles sont associés à la notion d'attribut lors de la description d'objet [Rous88] :

- *l'attribut en tant que description de la structure* :
  Un attribut peut être perçu comme un élément constitutif de l'objet, il joue alors le rôle de lien entre un composant et un composé.

- *l'attribut en tant que description d'une propriété* :
  L'attribut est ici le reflet d'une des propriétés de l'objet, comme la couleur d'une voiture, ou le type d'une matrice (rectangulaire, symétrique...). Ces propriétés permettent de préciser le type de l'objet.
- *l'attribut en tant que description d'une relation* :
  Lorsque la valeur associée à un attribut est elle-même un objet, les deux objets sont alors liés par cet attribut qui joue le rôle d'une relation. L'attribut **épouse** de l'objet **Homme** qui prend sa valeur dans l'objet **Femme**[1] en est un exemple. Si on désire que cette relation maritale soit symétrique, un attribut **époux** doit exister pour la femme. De plus, des procédures de gestion de la relation doivent être prévues pour mettre à jour ces attributs lorsque l'un d'eux est modifié (YAFOOL propose une gestion de relation similaire : définition dans un objet d'un lien (un attribut relation) et du lien réciproque).

La façon de considérer un attribut dépend essentiellement de l'utilisation que l'on veut en faire. Par exemple, si un attribut ne joue le rôle que d'une propriété, ce n'est pas alors la peine de le considérer et de le gérer comme une relation. L'important dans sa manipulation est de pouvoir lui associer une valeur et d'en vérifier le type. Un attribut décrivant une relation de composition impose une gestion particulière des liens qui garantit une certaine intégrité du tout (composants et composé) ; ceci suffit pour la distinguer des autres relations qui sont bien souvent dépendantes de l'application même.

Dans une représentation par objets, l'intégration de relations peut se faire en étendant l'ensemble des relations prédéfinies, par exemple, la composition qui permet d'introduire la notion d'objet composite [Blak&87] [Duc90] [Mari91] [Magn94]. Elle peut aussi se faire en offrant au concepteur la possibilité de définir lui-même ses relations dans lesquelles il peut spécifier leurs comportements vis-à-vis des objets qu'elles mettent en relation (gestion des dépendances entre les objets, définition des règles d'héritage ou de partage de valeurs d'attributs entre les objets...). Dans ce dernier cas, la relation est souvent modélisée à l'aide de classes dont la description fournit les éléments nécessaires à la gestion de la relation, c'est le cas des systèmes OTHELLO [Forn&89], SRL [Fox&86], SHOOD [Esca&90]...

3.2.4. Les contraintes

La notion de contraintes (cf. §II.3.4.2), quant à elle, vient naturellement enrichir l'expression d'une classe puisqu'elle offre un moyen d'indiquer à la fois une propriété que doit vérifier un objet et les différentes façons d'inférer des valeurs d'attributs pour que l'objet satisfasse la propriété. Pour ce faire, le langage de description des objets est augmenté par un ensemble de primitives de contraintes, et la gestion des contraintes est à la charge d'un module spécialisé qui doit rendre compte du résultat des vérifications de contraintes et s'occuper des propagations des valeurs d'attributs des instances sur lesquelles sont posées les contraintes. La dualité des contraintes comme étant à la fois entités de description et moyen d'inférence d'informations explique l'intérêt de son intégration dans une représentation par objets.

Le système THINGLAB [Born81] est une extension du langage de programmation SMALLTALK qui caractérise parfaitement l'apport de l'intégration de contraintes. Dans ce système, un objet est composé de parties sur lesquelles est spécifié un ensemble de contraintes. Cette transformation de l'objet initial de SMALLTALK le rend beaucoup plus proche de la notion d'objet des langages de représentation. En effet, les contraintes fournissent une façon naturelle d'exprimer et de maintenir des relations entre les parties d'un objet.

Depuis, la notion de contrainte tend à devenir un concept standard d'un modèle objet comme l'atteste le produit commercial PECOS [Puge92] qui fournit un ensemble de primitives permettant d'exprimer des problèmes de satisfaction de contraintes (CSP) au sein d'un langage à objets basé sur LE-LISP [Chail&86]. Dans le modèle TROPES, J. Gensel [Gens93] montre comment le module de gestion d'objets et un module de gestion de contraintes comme PECOS peuvent être couplés ensemble en représentant les contraintes par des classes, et comment

---

[1] Voila qui rétablit l'équilibre entre l'homme et la femme: ce sont tous deux des objets!

l'interaction entre ces deux modules permet des inférences de valeurs d'attributs d'objets (cf. §VI.2).

L'apport de l'introduction des contraintes dans un modèle objet est donc double puisqu'il permet :

- L'augmentation du pouvoir de description des attributs d'une classe.
- L'expression et la maintenance d'interdépendances entre les attributs d'une classe.

Classiquement, ces fonctionnalités sont trop souvent cachées dans les procédures attachées aux attributs par les réflexes et, par conséquent, leur mise en place est à la charge du concepteur d'une base. L'extension d'un modèle par les contraintes est un pas nécessaire vers la déclarativité puisqu'il permet de séparer les mécanismes de gestion des contraintes des contraintes elles-mêmes.

3.2.5. Conclusion

Le foisonnement d'idées réunies autour de la notion de classe est largement influencé par des prérogatives de programmation comme l'attestent la programmation dirigée par les accès due à l'attachement procédural, l'utilisation du défaut pour initialiser des attributs, la présence de méthodes au sein des classes comme dans YAFOOL, LOOPS... Mais d'un autre côté surgissent des notions, comme les relations et les contraintes, qui enrichissent la notion d'objet dans l'optique de la représentation de connaissances. L'expérience de THINGLAB en est un bon exemple.

Ces divers points de vue sur l'objet expliquent en grande partie les confusions dans l'utilisation de diverses notions. L'attachement procédural y contribue largement puisqu'il permet dans un modèle à objets d'exprimer procéduralement ce qui ne peut l'être déclarativement. Son rôle n'est pas toujours clair : permet-il d'exprimer des calculs de valeurs d'attributs ? Permet-il de généraliser la notion de valeur par défaut ? Permet-il d'exprimer des dépendances entre attributs ? Etc.

Dans un contexte de représentation de connaissances, la notion de défaut (facette *$si-besoin* ou *$defaut*) introduit une notion beaucoup plus générale, celle d'hypothèse. Son statut dans une description de classe est donc différent des autres éléments. Le pragmatisme avec lequel est traité le défaut lors de l'héritage en démontre la singularité. Comme le souligne R. Brachman [Brac&85], c'est aussi une entrave aux mécanismes comme la classification qui exploitent la hiérarchie de classes pour raisonner. C'est pour cela qu'il propose de les exprimer en dehors de la hiérarchie de classes, et de ne laisser dans la description d'une classe que les conditions nécessaires. Sans aller jusque là, et pour préserver la facilité d'expression qu'offre le défaut, il est important de considérer et de gérer une valeur déduite par défaut comme une valeur hypothétique [Mari89], donc révisable.

Dans [Napo&92] (cf. §II.3.1.1), un ensemble de règles auxquelles doivent se contraindre les facettes d'un attribut est décrit pour donner à la relation "d'héritage" la sémantique d'une relation de subsomption, appelée en l'occurrence *O-subsomption*. Cette démarche permet de construire une hiérarchie de classes, incluant des facettes de défaut et d'attachement procédural, de façon cohérente et exploitable par les mécanismes de subsomption. Chaque règle décrit le comportement de chaque type de facette pour la subsomption. Ainsi entre les facettes décrivant le type et le domaine d'un attribut la subsomption est décrite en terme de sous-typage et inclusion de domaines. Les facettes *$defaut*, *$si-besoin*, et *$valeur* nécessitent des règles particulières. Les règles suivantes doivent être appliquées pour qu'un attribut **a1** subsume un attribut **a2** :

- Une valeur par défaut doit vérifier les contraintes sur le type de l'attribut.
- Une valeur introduite par la facette *$valeur* dans l'attribut **a1** est une valeur fixée. La valeur de **a1** ne peut être masquée dans **a2**. Les facettes *$si-besoin* et *$defaut* ne peuvent être associées à **a2** quand une facette *$valeur* est associée à **a1**.
- Une facette *$defaut* ne peut être associée à **a2** si une facette *$si-besoin* est associée à **a1**.

Par cette proposition, on voit clairement un effort d'unifier les travaux sur les langages terminologiques et ceux sur les langages de *frames*. Etant donné l'effort en sens inverse qui consiste à enrichir l'expressivité des langages proposant la classe comme modèle pour l'identification, on peut espérer un consensus sur l'utilisation et la description d'une classe dans le contexte de la représentation des connaissances par objets.

4. La classe comme modèle mixte

Etant données les propriétés différentes que confère chacun des deux aspects précédents de la classe, il est intéressant d'essayer de les concilier au sein d'un même modèle. Le premier aspect permet de confronter et de gérer l'évolution d'une instance dans une hiérarchie de classes, le deuxième aspect offre de nombreuses possibilités pour la construction de l'instance via la description d'une classe. L'avantage d'un modèle mixte est de pouvoir mettre en place des mécanismes de construction incrémentale d'instance par exploration de la hiérarchie de classes.

Cette partie souligne l'intérêt de cette approche en présentant deux catégories d'applications qui exploitent un tel mécanisme, puis présente deux modèles à objets qui offrent un mécanisme de classification permettant de concilier construction et identification d'instances.

4.1. Intérêt de l'approche

L'intérêt du compromis entre les deux aspects de la classe est illustré par deux types d'applications qui mettent en œuvre un mécanisme alternant phase d'identification et de production : la première consiste à modéliser un processus de conception par affinement et instanciation et la deuxième utilise le raisonnement par classification du modèle SHIRKA pour le diagnostic médical.

4.1.1. Conception par affinement et instanciation

Un problème de conception consiste à construire un objet satisfaisant un ensemble de contraintes. Dans ce genre de problème, l'évolution de l'objet est primordiale puisqu'elle décrit le processus de conception. Initialement incomplète, la connaissance sur l'objet comprend les données connues et les exigences du concepteur. Par la suite, on cherche à trouver un modèle correspondant à ces données qui fournit une méthode pour construire l'objet. Dans un contexte de représentation par objets, la recherche de ce modèle correspond à la recherche de la classe la plus spécifique. L'exploration de la hiérarchie de classes décrit alors le processus de conception ; la classe représente le modèle sur lequel l'objet peut être conçue. C'est pourquoi les techniques de programmation par objets fournissent un environnement approprié pour mettre en place un système d'aide à la conception.

C'est dans des termes semblables que [Gero&88] propose un modèle théorique pour la conception. Ce modèle s'articule autour d'un concept appelé ***prototype*** qui est présenté comme le schéma conceptuel pour la représentation de connaissances de conception. Ce *prototype* est représenté naturellement par une classe ; les entités conçues en sont alors les instances. Le processus de conception revient à sélectionner par classification une classe correspondant à la situation courante et à l'instancier.

La représentation d'un *prototype* de conception par une classe nécessite différents types d'attributs en fonction du rôle que chacun d'eux joue dans le processus de conception. Les attributs de définition représentent les données ou paramètres du processus de conception. Les attributs d'environnement rendent compte des éléments externes qui peuvent agir sur la conception. Et enfin, les attributs résultats spécifient les buts à atteindre et sont dérivés des attributs de définitions. Les *prototypes* sont organisés dans une hiérarchie représentant les différents niveaux d'abstraction ; le passage d'un niveau à un autre apparaît lors de l'affinement. Le processus de conception peut commencer à n'importe quel niveau d'abstraction. Ce processus consiste en trois étapes :

- **Préparation et identification des spécifications de la conception courante** : cette phase consiste à fournir les attributs, leurs valeurs ou domaines de

valeurs admissibles. Cet ensemble de données forme un filtre qui sera utilisé pour rechercher les *prototypes* adaptés au problème spécifique.

- **Sélection d'un *prototype*** : la seconde étape consiste à sélectionner à travers la hiérarchie de *prototypes* un *prototype* de conception approprié aux spécifications, qui peut potentiellement satisfaire le filtre conçu à l'étape précédente. Quand un *prototype* est sélectionné, une instance est créée pour représenter les spécifications de conception dépendantes du contexte.
  Il est possible que plusieurs *prototypes* soient sélectionnés, on peut alors essayer plusieurs *prototypes* ou en choisir un en particulier.
- **Définition et évaluation de l'entité conçue** : La définition de l'entité intervient lors de l'instanciation du *prototype*, en affectant aux attributs les valeurs données lors des spécifications. Les attributs de définition qui n'ont pas été fixés initialement sont alors réclamés à l'utilisateur, ou calculés, alors que les attributs résultats sont évalués, c'est-à-dire qu'ils sont calculés et comparés aux contraintes définies pour eux dans le *prototype*.
  Si la valeur de certains attributs sont elles-mêmes un *prototype*, le même processus de conception lui est appliqué.

Cette approche du problème de conception montre l'intérêt d'utiliser les objets aussi bien pour l'identification que pour la construction. L'utilisation de la représentation par objets nécessite un grand nombre de moyens : classification (ou filtrage), instanciation, traitement d'objets composites, contraintes sur les objets, calculs de valeurs d'attributs...

Le processus de conception montre clairement, par son étape de sélection d'un *prototype*, l'intérêt d'un mécanisme de classification. Toutefois, tel qu'il est présenté, il ne se présente pas comme une évolution de l'objet en cours de conception mais comme un mécanisme de filtrage sur les classes. Ceci est dicté principalement par l'utilisation d'un modèle objet ne permettant que l'instanciation, la migration d'instance y étant prohibée. Aussi, au lieu de créer l'instance et de la classer par la suite, on doit créer un filtre rassemblant les connaissances de l'utilisateur sur l'entité à concevoir, puis sélectionner par affinement une classe et enfin l'instancier. Conceptuellement, le filtre représente une vue incomplète de l'instance finale puisque les informations qu'il porte sont des valeurs ou des contraintes sur des attributs de cette instance.

Ce travail est une proposition de modélisation de la conception d'objet réel par raffinement et instanciation, il nous intéresse car il est conceptuellement proche de la construction d'instance dans un modèle par objets. Le modèle SHOOD, étudié par la suite, applique au niveau même de la gestion de ses objets un principe similaire.

4.1.2. Diagnostic par affinement et instanciation

Une hiérarchie de classes peut être perçue comme un "arbre de discrimination" [Ferb86] sur lequel peut s'appuyer un raisonnement de diagnostic. Par exemple, une hiérarchie de maladies est naturellement représentée par une hiérarchie de classes. Chaque classe contient les informations permettant d'accepter ou de rejeter une instance correspondant à la situation courante. Le processus d'analyse consiste donc à classer cette instance, tout en la "remplissant", le long de la hiérarchie.

Par leur nombre et leur complexité, les informations dans une classe gagnent à être elles-mêmes structurées en plusieurs concepts. Par exemple, dans une hiérarchie pour le diagnostic médical, des informations sont propres à l'élaboration des signes cliniques, d'autres à la description des maladies (le type d'organe atteint, nerf ou muscle ; la localisation). Ces regroupements d'informations constituent les concepts sur lesquels un médecin raisonne. La prise en compte de ces concepts donnent lieu à un éclatement du domaine en plusieurs hiérarchies chargées de les représenter. Le raisonnement classificatoire consiste alors à explorer ces différentes hiérarchies.

Une telle modélisation est proposée pour le diagnostic en électromyographie (étude des pathologies neuromusculaires) dans le système MYOSIS [Cruy&92]. Le raisonnement s'organise autour de la classification d'objets complexes. Les objets sont dits complexes

lorsque certains de leurs attributs font références à d'autres objets. Ce type d'attributs, appelés aussi complexes, permet de décrire un lien entre différents concepts d'électromyographie ; chacun de ces concepts étant représenté par sa propre hiérarchie de classes dans le système SHIRKA.

Le diagnostic consiste alors à classer une instance dans l'une des hiérarchies (Signe-Clinique). Lorsqu'un attribut fait référence à une autre hiérarchie (Hypothèse-Maladie), une instance de ce concept est créée et est à son tour classée. Ce processus se répète autant de fois que nécessaire avec d'autres hiérarchies (Figure IV.5).

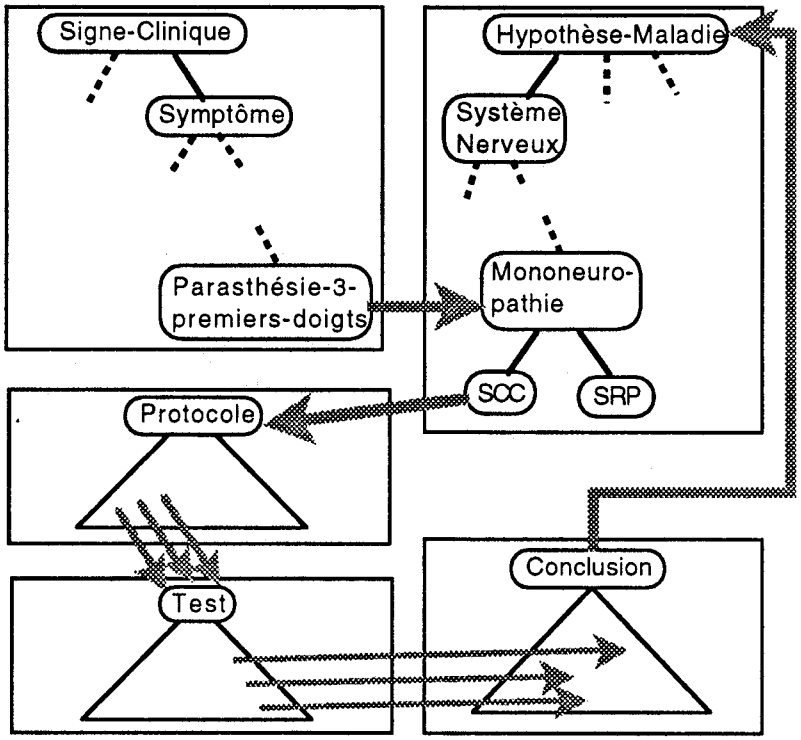

Figure IV.5 : Modélisation du raisonnement de diagnostic électromyographique à travers plusieurs hiérarchies de classes représentant les différents concepts du domaine. La classification se propage d'une hiérarchie à l'autre par le biais des attributs complexes, c'est-à-dire ceux qui font référence à d'autres objets. Par exemple, lorsque la classe **Paresthésie-3-premiers-doigts** est sélectionnée, elle propage par son attribut **evoque** la classification d'une nouvelle instance d'hypothèse de maladie à partir de la classe **Mononeuropathie**.

Dans l'exemple, le raisonnement électromyographique passe par la création et la classification en séquence d'une instance de signe-clinique, d'une instance d'hypothèse de maladie, d'une instance de protocole chargée de vérifier l'hypothèse, d'un ensemble d'instances de tests composant le protocole, d'un ensemble d'instances de conclusions partielles résultant des tests. La classification des conclusions partielles provoquent la création d'une conclusion générale. Cette conclusion générale décrit le résultat de l'hypothèse de maladie testée, et peut soumettre à nouveau une hypothèse de maladie par le même processus si besoin est.

Ces classifications lancées à partir d'un concept dans un autre concept sont représentées à l'aide de l'attachement procédural fourni en SHIRKA par la facette *$sib-exec* ("si besoin exécuter"). La procédure employée est une séquence composée de deux primitives du système SHIRKA : création d'instance, puis classification d'instance. L'exemple suivant (cf. Figure IV.6) présente la classe **Paresthésie-3-premiers-doigts** dont l'attachement procédural sur l'attribut **evoque** est chargée de lancer la création et la classification d'une instance d'hypothèse de maladie à partir de la classe **Mononeuropathie**. Certains attributs de cette nouvelle instance sont donnés lors de l'exécution de l'attachement procédural (**nerfs-suspects**, **segments-suspects**, etc.). Le résultat de cet attachement est l'instance créée.

```
{ Paresthésie-3-premiers-doigts
   sorte-de          =    paresthésie;

   lieux--apparition $domaine   median;
   regions           $domaine   main;

   évoque            $un  Hypothèse-Maladie
                     $sib-exec {
                          créer-classer-hyp
                              classe              $valeur mononeurothérapie;
                              nerfs-suspects      $valeur median;
                              segments-suspects   $valeur "paume-poignet";
                              type-atteinte       $valeur "focale";
                              instance            $var-> évoque
                          }
}
```

Figure IV.6 : Exemple d'attachement procédural dans une classe de signe clinique pour la création et la classification d'une instance d'hypothèse de maladie. La procédure **créer-classer-hyp** est chargée de créer une instance et de la classer à partir de la classe décrite dans son paramètre **classe**.

Dans cette application, il est clair que les classes sont des modèles mixtes. La classification s'appuie sur elles pour guider le raisonnement, mais aussi pour construire un ensemble d'instances qui constitue l'ensemble des étapes suivies et la solution au problème posé.

Indéniablement utilisée pour compléter une instance, la classification proposée ici se singuralise par le fait qu'elle se propage sur d'autres instances par l'entremise des attachements procéduraux **creer-classer-hyp** associés aux attributs complexes. Pour ce faire, une hypothèse cruciale a été posée qui assure le bon fonctionnement du mécanisme de classification : les attachements procéduraux n'échouent jamais du fait de la modélisation adaptée. En effet, même si l'hypothèse de maladie s'avère mauvaise, elle sera créée et donnera lieu à des conclusions signalant ce fait. L'instance représentera alors une hypothèse de maladie récusée. C'est-à-dire qu'à chaque fois que l'on confronte une instance incomplète à une classe comportant un tel attachement procédural, on est assuré de compléter l'instance et de pouvoir l'attacher à cette classe.

Par ailleurs, pour que le mécanisme de classification soit pertinent, les hiérarchies de classes doivent être bien formées ; c'est-à-dire qu'elles doivent assurer que la sélection des classes est bonne. Il ne doit pas y avoir en compétition deux classes permettant un attachement procédural différent. Auquel cas, il y aurait conflit pour le calcul de l'attribut, en l'occurence de l'instance à créer et classer. Ce conflit peut toujours être résolu par l'utilisateur qui trancherait pour l'une ou l'autre des solutions (à supposer que l'une des deux constitue une solution). Une autre solution consisterait à créer et à continuer avec une version de l'instance pour chaque classe.

Le domaine considéré doit être suffisamment bien connu pour permettre cette modélisation. En effet, les hiérarchies utilisées doivent être pertinentes pour la mise en place du raisonnement par classification d'objets complexes présenté ici. Car du point de vue de la représentation par objets, cela se traduit par l'utilisation d'une classe possible comme si elle était sûre pour l'instance classée. C'est ainsi que les informations portées par cette classe peuvent être utilisées pour compléter l'instance classée. L'hypothèse sur laquelle se fonde la modélisation est que, dans la hiérarchie de classes, il n'y a qu'une classe possible de ce type convenant à l'instance. Des modifications intempestives ou simplement une extension de la base de connaissances peuvent remettre en cause cette hypothèse, et mettre à défaut le mécanisme de raisonnement dont une partie du contrôle est dispersée dans les attachements procéduraux.

4.2. Systèmes proposant identification et construction d'instances

Les modèles à objets SHOOD et FROME constituent deux exemples de systèmes qui permettent de concilier évaluation des attributs et classification d'instances. Ils sont successivement présentés dans les parties suivantes.

4.2.1. Le modèle à objets SHOOD

SHOOD [Esca93] est un système de représentation par objets dans la lignée des langages de *frames*. Il s'inspire de ROME (cf. §IV.4.2.1) en ce qui concerne le traitement des représentations multiples d'un objet, et de SHIRKA par le mécanisme de classification d'instance qu'il propose. SHOOD est présenté comme un modèle spécialisé pour la conception assistée [Nguy&92].

Le modèle de SHOOD est organisé autour d'un méta-modèle réflexif qui constitue le noyau objet standard. S'il doit faire face au problème de la profusion d'instances due à l'instanciation systématique de cette méta-connaissance, il permet, par contre, d'étendre et d'adapter le modèle aux besoins spécifiques d'un domaine particulier. C'est dans cet esprit qu'il s'applique au domaine de la conception.

Les atouts qu'il offre actuellement à un tel domaine sont la prise en compte de l'évolution des objets, la possibilité de définir des relations sémantiques entre les objets via des relations de dépendances [Esca&90], la représentation de méthodes [Rieu&92] inspirée de CLOS [Gabr&91], la définition de contraintes violables ou non sur un attribut, etc.

De plus, pour décrire un mécanisme d'affinement comme pour le modèle de conception précédent, SHOOD intègre un mécanisme de classification qui permet à une instance d'explorer une hiérarchie de classes et de déduire par inférence (attachement procédural) des valeurs d'attributs de cette instance [Liot93]. Le problème de conception d'un objet réel est alors ramené au problème de construction incrémentale d'instances par classification. Ainsi, cette proposition de classification dans SHOOD s'inscrit pleinement dans la problématique du traitement d'une instance incomplète et de l'utilisation des attachements procéduraux qui permettent de la compléter.

- **Mécanisme de classification d'instance**

Le mécanisme de classification explore la hiérarchie de classes et donne à chaque classe rencontrée le statut soit **possible**, soit **impossible**. Ce statut est fonction du résultat de l'appariement entre l'instance et la classe ; à la manière du modèle TROPES [Mari&90], le résultat de l'appariement donne lieu à la déduction du statut d'autres classes en fonction d'un ensemble de règles de cohérence caractérisant la sémantique de la relation de spécialisation et de la disjonction de classes.

Le statut impossible signifie que l'instance ne vérifie pas les exigences de la classe. Celles-ci consistent pour tous les attributs en la vérification de type, de domaine, de prédicat (appelé **contrainte forte**), et aussi, en l'obligation d'avoir au moins une valeur pour certains attributs (appelés **attributs obligatoires**). Le statut possible est, quant à lui, le statut complémentaire du statut impossible, c'est-à-dire qu'une classe devient possible si aucune des exigences n'est violée. Il n'y a pas de statut sûr comme dans TROPES ou SHIRKA, ce cas est inclus dans le statut possible.

- **Inférence d'attribut**

En ce qui concerne l'attachement procédural dans SHOOD, il consiste à associer un ensemble de méthodes, appelées **inférences**, à la description d'un attribut dans une classe. Ces inférences sont chargées du calcul de la valeur de l'attribut. L'ordre dans lequel elles sont données correspond à un ordre de priorité d'exécution lors d'une demande de calcul. L'exécution des inférences s'achève dès que l'une d'entre elles fournit un résultat acceptable vis-à-vis des contraintes portées sur l'attribut ou quand toutes les inférences ont échoué. Dans ce dernier cas, l'attribut reste sans valeur.

Les *inférences* peuvent avoir un statut de calcul global ou local, la distinction se faisant en fonction du type de l'attribut calculé. Cela se traduit par la mise en place de deux méta-classes : **Méta_attribut** et **Méta_att_inf_local**.

La première est sur-classe de la deuxième, elle est choisie par défaut. Elle caractérise des attributs dont les inférences sont censées rendre toujours le même résultat pour la même instance ; c'est-à-dire que le résultat ne dépend pas de la ou les classes d'appartenance de l'instance mais des attributs qu'elle possède au moment du calcul. C'est le cas, par exemple, de l'attribut **âge** qui pourrait être calculé par une inférence qui utilise la date de naissance ou une autre qui utilise le numéro de sécurité sociale. L'âge reste le même, seule la façon de le calculer est différente ; elle dépend des informations disponibles.

Les attributs déclarés comme instances de **Méta_att_inf_local** représentent les attributs dont la valeur est dépendante de la classe de rattachement de l'instance. Par exemple, le calcul de la masse d'une barre sera différent si elle est avec ou sans axe, avec ou sans trou, etc. Le calcul ne rend pas le même résultat. Les inférences ne sont valides que localement à la classe à laquelle elles sont associées.

- **Emergence de plusieurs instances lors de la classification**

L'émergence de plusieurs instances provient du traitement des attributs inférés localement. En effet, dans le cas d'un attribut inféré localement, l'algorithme de classification peut rencontrer une situation de conflit : un **conflit** apparaît lorsqu'une instance I possède déjà une valeur inférée pour un attribut A et que la classe C en cours d'appariement propose aussi, par l'utilisation de l'une de ses inférences, une valeur pour l'attribut A.

La solution adoptée pour résoudre un conflit consiste à reproduire l'instance et à reporter le nouveau résultat sur la copie de l'instance. Ainsi, dans le cas de l'instance I et du conflit sur l'attribut A, le processus de classification exploite le conflit à la fois pour donner le statut à la classe C et pour décider de la génération d'une instance adaptée I' :

- La classe C est déclarée impossible pour I, et une instance I' est générée par copie de I et mise à jour de la valeur de l'attribut A.
- C est déclarée possible pour I', et toutes les sur-classes de C possédant une inférence pour A sont déclarées impossibles pour I'.
- Lorsqu'il y a multi-instanciation, l'inférence choisie doit être commune à toutes les classes d'appartenance directes de l'instance.

Ce mécanisme est illustré par un exemple de résultat de classification repris de [Liot93] (cf. Figure IV.6). L'objet classé s'avère être une barre avec axe et trou. Cet objet est représenté par l'instance I1.

Dans cet exemple de classification, l'appariement de I1 avec la classe Barre déclenche l'inférence locale Calcul_Masse_barre. Lors de la descente de I1 dans la classe Barre_Avec_Axe, un conflit sur le calcul de Masse apparaît. L'instance I1 est déclarée impossible pour cette classe et l'instance I2 est alors générée à partir de I1 pour être attachée à la classe Barre_Avec_Axe. La valeur de masse de I2 est celle inférée par l'inférence locale Cal_Masse_Axe. La classe Barre obtient le statut impossible pour I2.

Puis, la descente de I1 dans la classe Barre_Avec_Trou provoque la génération de I3 et le calcul Masse_Trou. Enfin, l'instance I4 est générée lors de la classification de I2 qui est en conflit avec la classe Barre_Avec_Trou. L'inférence utilisée pour ce calcul est Cal_Masse_Axe_trou, commune aux deux classes, puisqu'il y a multi-instanciation pour I4.

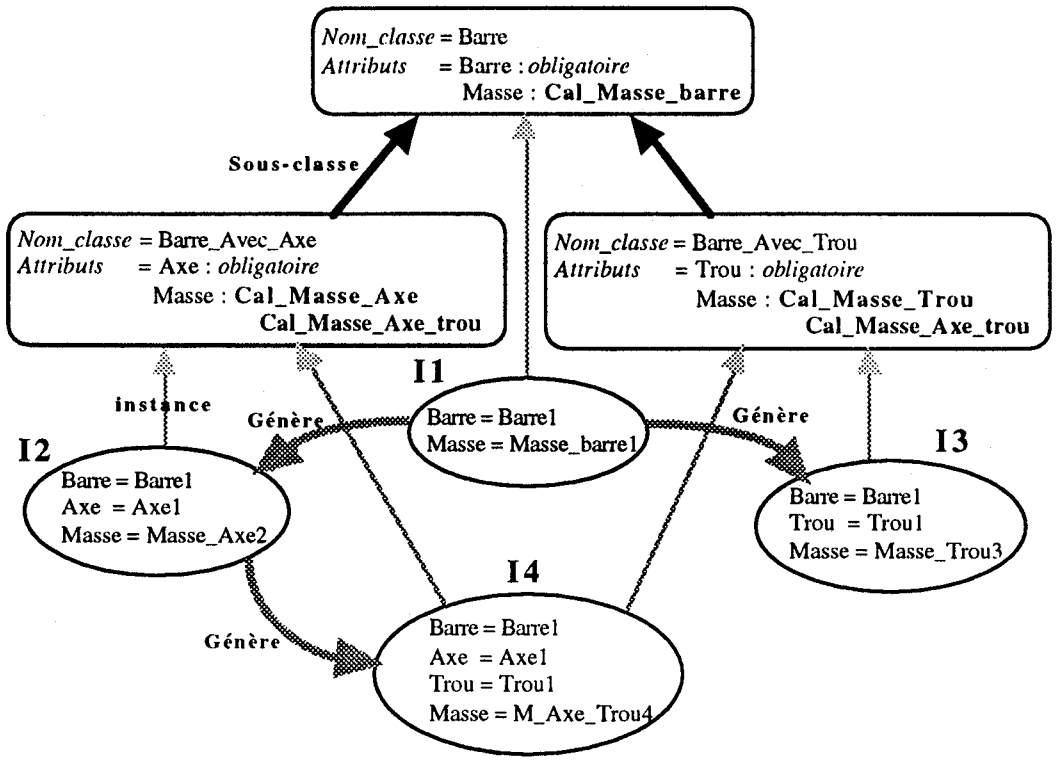

Figure IV.6 : Exemple du principe de classification d'instance avec inférences locales de SHOOD. Cet exemple montre le traitement du conflit sur le calcul de l'attribut Masse lors de la classification de l'instance I1. En résultat, les trois instances I2, I3 et I4 ont été successivement créées pour résoudre les différentes situations de conflit.

Il est à noter que la catégorie des attributs inférés globalement ne pose pas de problème lors de la classification puisque la première inférence rencontrée qui réussit est considérée comme la bonne. L'attribut se voit alors affecter la valeur calculée, et les autres inférences sont ignorées.

L'approche utilisée par SHOOD est intéressante puisqu'elle permet d'explorer la hiérarchie avec une instance incomplète pour en donner des versions complétées suivant les différentes inférences employées. Toutefois, trois points majeurs sont critiquables : la perte de la sémantique de la relation de spécialisation, la nécessité d'avoir une inférence commune à toutes les classes de rattachement directe de l'instance pour le calcul d'un attribut, et enfin le procédé de création et réadaptation de la version d'une instance. Ces trois points sont abordés par la suite.

- **Critique sur la sémantique de la spécialisation**

La première critique concerne la façon de marquer les classes lors de la gestion des conflits. Elle va à l'encontre de la sémantique d'inclusion ensembliste, pourtant annoncée, qui est associée à la relation de spécialisation de classes. En effet, suivant la gestion de conflit de l'instance I2 de l'exemple, la classe Barre_Avec_Axe est possible pour cette instance alors que sa sur-classe Barre est impossible. Cela signifie que I2 n'est pas une barre. Plus généralement on peut remarquer que pour chaque instance I1, I2, I3 et I4, les seules classes possibles sont les classes de rattachement direct.

La propriété d'inclusion ensembliste de la relation de spécialisation est perdue. Deux raisons sont à l'origine de cette perte de sémantique.

D'une part, la notion d'inférence locale joue le rôle d'une contrainte sur l'attribut : pour qu'une instance appartienne à une classe possédant des attributs avec inférences locales, les attributs correspondants dans l'instance doivent être calculés à partir des inférences locales de la

classe. Même si la valeur retournée par l'inférence est la même que celle possédée par l'instance, cette dernière est rejetée à cause d'un conflit sur l'origine différente des valeurs .

D'autre part, la modélisation adoptée dans l'exemple des barres ne correspond pas à une hiérarchie de classes d'un système à objets basé sur une relation de spécialisation stricte. En effet, dans cet exemple, une barre avec axe est **presque** une barre, **sauf** qu'il y a un axe en plus et que le calcul de la masse doit prendre en compte cet axe. La classe Barre n'exprime pas un concept plus abstrait que ceux exprimés par Barre_Avec_Axe ou Barre_Avec_Trou mais un concept différent. Le lien entre la classe Barre et ces deux sous-classes est un lien de type *prototype*.

- **Critique de la contrainte d'une inférence commune à toutes les classes**

La seconde critique concerne la contrainte posée dans le modèle SHOOD sur la nécessité d'utiliser pour le calcul d'un attribut que les inférences locales communes à toutes les classes de rattachement direct (lors d'une multi-instanciation).

Cette contrainte est posée afin d'assurer que le résultat d'un calcul par inférence locale soit valide pour toutes les classes de rattachement direct. Toutefois, elle a des conséquences néfastes sur la construction des hiérarchies de classes. En effet, lors de la création d'une classe, pour couvrir tous les cas de calcul local d'un attribut, il faut prendre en compte l'ensemble des inférences locales des classes qui peuvent participer à une situation de multi-instanciation avec cette nouvelle classe. Inversement, l'introduction d'une nouvelle inférence locale dans la classe créée doit être répercutée sur la description de ces classes.

Ce partage d'informations entre classes, indépendant de l'héritage propre à la spécialisation, nécessite une vision globale de la hiérarchie de classes, et réduit de ce fait les propriétés de modularité propre à un modèle à objets. Il est toujours dangereux pour la cohérence d'une base de connaissances qu'une modification locale de la description de la base puisse avoir des répercussions globales.

Puisque cette contrainte semble un prix cher à payer sur la puissance d'expression du modèle, il est intéressant d'envisager les conséquences de son retrait (cf. Figure IV.7).

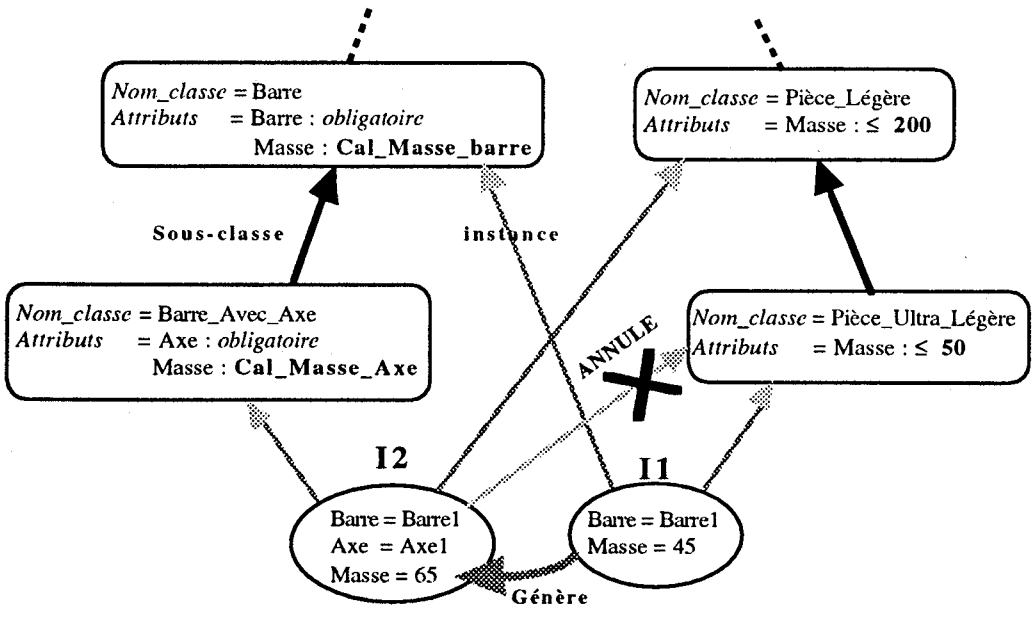

Figure IV.7 : Scénario **impossible** dans SHOOD à cause de la contrainte obligeant toutes les classes de rattachement d'une instance à posséder la même inférence pour le calcul d'un attribut. Dans ce scénario, l'instance I2 est générée à partir de I1 et de la classe Barre_Avec_Axe. Puisque I2 s'avère contredire la classe de rattachement Pièce_Ultra_légére de I1 de par la nouvelle valeur de la masse, son lien de rattachement avec cette classe devrait être annulé, et I2 migrerait alors vers la sur-classe Pièce_Légére.

Si l'on suppose qu'une inférence locale peut être utilisée alors que certaines classes de rattachement ne la possèdent pas dans leur description, il reste que toute valeur inférée doit néanmoins vérifier la description de ces classes. En effet, celles qui n'auraient pas participé au calcul d'inférence de l'attribut en conflit pourraient alors voir violées leurs contraintes par cette nouvelle valeur. Il convient en cas de violation des contraintes d'appartenance de traiter ces cas comme un nouveau type de conflit.

Pour résoudre ce nouveau type de conflit, une solution tout à fait possible consiste à générer une nouvelle version de l'instance intégrant la nouvelle valeur et remettant en cause les rattachements invalidés. L'exemple qui précède (cf. Figure IV.7) est un **scénario fictif** d'un tel traitement.

Ainsi, la contrainte qui était avant imposée au niveau de la description d'une base pourrait s'exprimer en termes de cohérence de la base et étendre la notion de conflit.

- **Critique sur la production des versions d'instance**

Enfin, la dernière critique soulevée concerne la façon de générer des versions d'instances. La démarche suivie consiste à adapter l'instance en générant de nouvelles versions qui remettent en cause les informations qui participent à un conflit. Or, adapter les instances entre elles (I2 est adaptée de I1, I4 est adaptée de I2, etc.) est un principe qui oblige à défaire ce qui dépend des calculs d'attributs abandonnés.

Certes, ce problème est minimisé dans SHOOD par la contrainte exigeant l'existence de la même inférence dans toutes les classes directes, mais cela au prix d'une diminution d'expressivité du modèle (l'exemple précédent de la figure IV.7 ne peut être exprimé). Aussi, un rejet éventuel de cette contrainte rendrait difficile l'adaptation des versions d'instance en fonction des conflits. Une analyse plus fine du processus de génération de versions permet de mettre en évidence la notion d'hypothèse sous-jacente à la notion de conflit.

En effet, d'une part, le traitement nécessaire pour l'adaptation d'une version est équivalent au traitement d'une remise en cause des informations à l'origine d'un conflit. C'est-à-dire que les informations incriminées, en l'occurrence le résultat d'une inférence locale, peuvent être perçues comme des connaissances par défaut, elles ont donc un statut hypothétiques. D'autre part, le souci de générer plusieurs versions montrent qu'il est important de conserver les solutions correspondant aux différentes hypothèses explorées.

L'interprétation du conflit en terme d'hypothèse suggère que le processus d'adaptation des versions d'une instance peut être évité si la gestion des hypothèses est explicite. En effet, puisque l'utilisation d'une inférence locale est susceptible de provoquer un conflit, autant la considérer de prime abord comme une hypothèse et générer une version pour accueillir son résultat. Ainsi, dans l'exemple de la figure IV.6, quatre hypothèses d'inférence ont été posées sur le même attribut, donnant lieu à quatre versions différentes d'instances : I1, I2, I3 et I4. Puisqu'elles portent sur le même attribut *Masse*, ces hypothèses constituent des extensions possibles d'une même instance, dont on ignore encore la valeur de l'attribut *Masse*. C'est donc à partir de cette instance que devrait être générée chacune des versions. Dans ce cas, l'adaptation des versions ne serait alors plus nécessaire puisque leur création constituerait un développement possible et monotone de l'instance initiale.

- **Conclusion de l'étude du modèle SHOOD**

L'adaptation du modèle SHOOD au domaine de l'aide à la conception passe par l'introduction de concepts et de mécanismes spécifiques qui traduisent les besoins des spécialistes en conception. Les divers types d'attachement procédural permettent de faire la différence entre les calculs inhérents à un attribut, et ceux qui dépendent du type de l'instance. La classification proposée permet d'explorer une hiérarchie de classes pour trouver celles qui sont susceptibles de correspondre au modèle de construction d'un objet. La génération de plusieurs versions de l'instance permet de considérer plusieurs solutions de construction de l'instance lors de la classification.

Si le principe de la combinaison d'identification et de construction d'instance proposé est prometteur, sa mise en place effective soulève des problèmes majeurs. En effet, la classification avec résolution de conflit entraîne une modification sournoise de la relation de spécialisation en une relation de type *prototype*, et remet donc en cause les fondements même du modèle. De plus, la solution choisie pour le traitement des conflits par génération de versions d'instances semble pouvoir être améliorée en tirant parti de la notion d'hypothèse inhérente à la notion de conflit.

4.2.2. Du langage ROME au modèle FROME

Une des originalités du langage ROME est de proposer une représentation d'objet évolutif [Carr&90] ; c'est-à-dire qu'une instance pourra voir son rattachement à une classe évoluer. Cette proposition est le résultat d'un constat : la capacité d'évolution d'une instance peut être utile en conception, pour le diagnostic, pour l'identification, et plus généralement durant un processus de décision [Carr&88]. En cela la hiérarchie de classes est un support intéressant pour décrire les évolutions possibles par affinement de la caractérisation de l'objet.

Le langage de *frames* FROME est, quant à lui, une extension du méta-modèle du langage objet ROME [Carr&91]. Un *frame* de FROME est un objet ROME qui est constitué d'une agrégation d'objets *slot*, les attributs du *frame*.

- **L'évolution des objets dans ROME**

Généralement, l'instanciation est la seule façon de spécifier la classe d'un objet. Elle impose une contrainte très forte sur l'instance puisqu'elle est irrémédiablement liée à sa classe d'instanciation. Aussi, la seule façon de faire évoluer un objet dans un langage à objets est de détruire l'instance et de la recréer dans la classe où elle doit être attachée.

Pour remédier à cela, ROME propose l'introduction d'un nouveau lien qui permet de décrire l'évolution de l'instance. Ce lien, appelé **lien de représentation**, vient compléter le lien d'instanciation qui, lui, reste statique et ne peut être défait. Le lien d'instanciation décrit les caractérisations minimales et irrévocables de l'objet. Le lien de représentation permet de rattacher l'instance aux classes vers lesquelles on désire la faire évoluer, et permet de plus de décrire plusieurs classes de rattachement, constituant différents points de vue sur le même objet.

L'introduction de ce nouveau lien se fait simplement par ajout d'une classe **R-OBJET** dans le méta-modèle, classe de toutes les instances évolutives et sous-classe de **OBJET**. Dans cette classe, l'attribut **r-classe** décrivant les liens de représentation est introduit.

Dans la figure IV.8, on peut voir cette partie du méta-modèle, ainsi qu'une hiérarchie de classes décrivant des composants électroniques comme des objets évolutifs (sous-classes de R-OBJET), et au plus bas niveau deux instances R1 et R2. R1 est une résistance qui a évolué jusqu'à la classe RCOURANTE grâce au lien de représentation.

L'évolution d'une instance doit se plier à la contrainte de représentation imposée par le modèle : une classe de représentation d'un objet évolutif ne peut être qu'une sous-classe de sa classe d'instanciation ; c'est-à-dire qu'un objet ne peut évoluer qu'en dessous de sa classe d'instanciation.

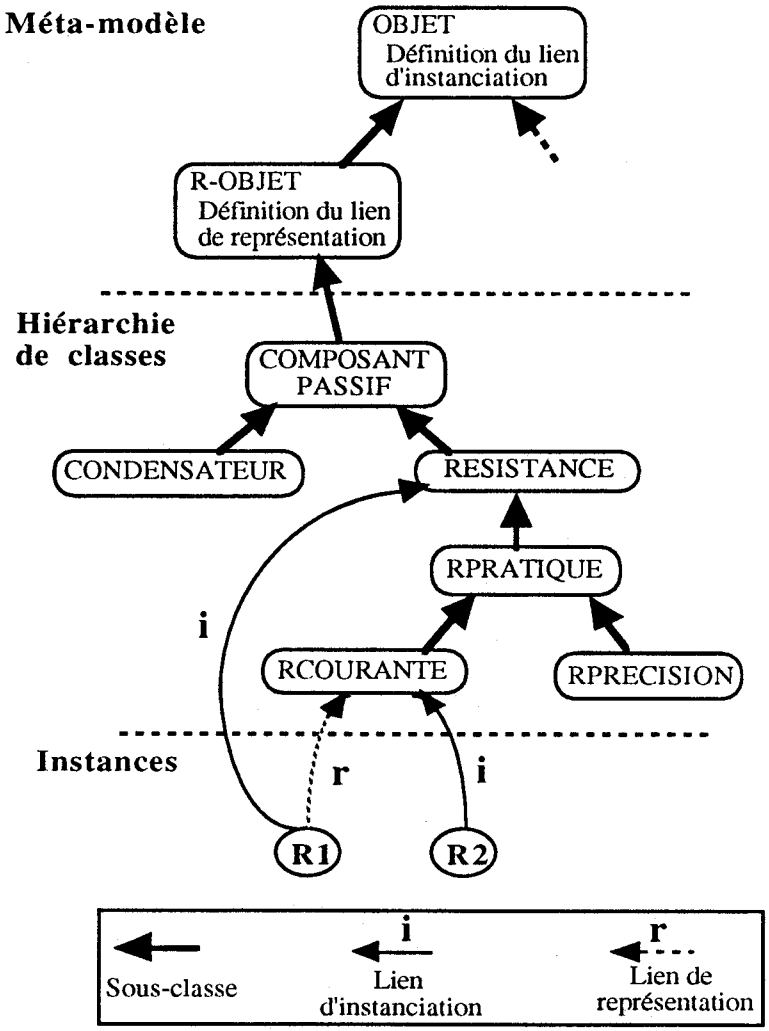

Figure IV.8 : Le modèle ROME introduit une classe R-OBJET qui dénote les objets évolutifs. Ainsi, la résistance R1, initialement instance de RESISTANCE, peut évoluer par le lien de représentation vers la sous-classe RCOURANTE. Cette modélisation est utilisée pour décrire l'évolution d'un objet lors du passage d'une phase théorique à une phase pratique de conception électronique [Carr&87].

La modification dynamique du lien de représentation se fait par deux opérations implémentées comme des méthodes. Le message "**r- <nom-classe>**" détruit le lien de représentation entre l'objet receveur et la classe spécifiée ; le message "**r+ <nom-classe>**", quant à lui, crée un lien de représentation entre l'objet receveur et la classe spécifiée.

- **Le raisonnement classificatoire dans FROME**

Bien sûr, FROME a hérité de ROME cette capacité à faire évoluer les objets [Carr&91] [Dekk&92]. Les deux opérations précédentes s'expriment alors par :

- [aFrame **isa** <C>] qui lie "aFrame" à la classe C ;
- [aFrame **no-more** <C>] qui détache "aFrame" de la classe C mais reste membre de la sur-classe de C.

FROME étant un langage de *frames*, les descriptions de classes définissent un ensemble de propriétés à vérifier. Ainsi le rattachement d'une instance à une classe est dépendant de la validation de ces propriétés.

L'ensemble de ces caractéristiques font du système FROME un contexte adapté à la mise en place d'un raisonnement classificatoire. Dans [Dekk94], une importante étude a été menée dans ce sens.

S'inspirant des langages terminologiques (cf. §I.3.2.3), ce travail s'intéresse à la relation existant entre la description d'une classe et l'appartenance d'une instance à une classe. De la même façon que les langages terminologiques distinguent les *concepts définis* et les *concepts primitifs*, deux types de classes sont distingués : classe définie et classe descriptive. La description des premières représentent pour les instances les conditions nécessaires et suffisantes d'appartenance tandis que celles des secondes se limitent aux conditions nécessaires.

Toutefois, l'originalité de l'approche est d'étendre cette distinction au sein même de la description d'une classe. En effet, des informations de différents types peuvent être apportées dans la description d'une classe :

- les *slots nécessairement existants* : pour appartenir à la classe, une instance doit nécessairement posséder ces slots.
- les *slots nécessairements évalués* : pour appartenir à la classe, une instance doit nécessairement posséder une valeur pour chacun de ces slots.
- les *slots déduits* : une instance qui appartient à cette classe possède nécessairement ces slots.

Lorsqu'une classe est *définie*, les slots nécessairement existants et les slots nécessairement évalués définissent les conditions nécessaires et suffisantes de l'appartenance d'une instance à une classe. Dans ce cas, les slots déduits définissent les propriétés contingentes qui seront nécessairement déduites après que l'appartenance de l'instance à la classe ait été établie.

L'intérêt de cette approche est donc de permettre à la classification, d'une part, d'établir l'appartenance de l'instance à une classe *définie* en considérant uniquement les conditions nécessaires et suffisantes définies par sa description et, d'autre part, de pouvoir inférer de nouvelles informations sur l'instance à partir des *slots déduits* de sa description. Puisqu'une telle classe peut contribuer aussi bien à l'identification qu'à la construction d'une instance, elle correspond à un modèle de classe mixte.

Par cette proposition, le système FROME permet de traiter de façon cohérente le problème de construction d'une instance par classification. Toutefois, la distinction explicite dans la description d'une classe entre les informations qui sont soit déterminantes, soit déterminées pour la classification impliquent une construction de la hiérarchie qui dénotent une connaissance de la façon dont on va l'exploiter. Le problème de l'impact d'un type de problème de construction d'instance sur la conception d'une hiérarchies de classes est un sujet traité dans le chapitre suivant (cf. §V).

5. Conclusion

Ce chapitre souligne la présence de deux grandes familles de représentation par objets se distinguant par le rôle qu'ils peuvent donner à la notion de classes. L'une d'elles assigne à la classe le rôle d'unité identification d'instances, tandis que l'autre lui assigne le rôle d'unité de construction d'instances. Ces deux aspects duaux de la notion de classe permettent dans un cas d'identifier les objets appartenant à une classe au vu de leurs propriétés et dans l'autre de construire une instance sur le modèle de la classe à laquelle elle appartient.

A la frontière entre ces deux familles se trouvent des modèles qui cherchent à exploiter la classe comme support à la fois de classification et de construction d'instances. L'objectif de ces derniers est de pouvoir concilier les avantages des deux premières familles afin de mettre en œuvre un raisonnement de construction d'instance par exploration d'une hiérarchie. L'intérêt d'un tel raisonnement est souligné par l'application qui en est faite dans des domaines comme l'aide à la conception et au diagnostic.

Ce raisonnement, appelé par la suite raisonnement d'identification/construction d'instances, est abordé en augmentant les possibilités de description d'une classe et en adaptant le mécanisme de classification d'instances. Cette adaptation consiste à permettre d'exploiter des mécanismes d'inférence d'attribut lors de la classification d'une instance incomplète.

Le système FROME définit clairement au niveau de la description des classes une séparation entre les connaissances pertinentes pour l'identification et celles qui ne le sont pas. Cette dernière catégorie offre donc la possibilité d'intégrer au sein d'une classe des mécanismes d'inférences d'attributs exploitables dans le contexte d'une classification d'instance. Le système SHOOD introduit la possibilité d'exploiter la classification d'une instance pour construire différentes versions de la même instance. Au delà de cette idée originale, le modèle SHOOD souffre des limitations d'une sémantique parfois mal définie.

Chacun de ses systèmes ramène le problème d'identification/construction d'instances à celui de la classification d'une instance incomplète. Cependant, il convient de se poser la question suivante : cette approche ne restreint-elle pas la problématique de l'identification/construction d'instances à celle de l'identification ?

Le chapitre suivant se propose, d'une part, de répondre à cette question et, d'autre part, de présenter une nouvelle approche du problème d'identification/construction d'instances.

Chapitre V
VERS UN MECANISME D'IDENTIFICATION/CONSTRUCTION D'INSTANCES

1. Introduction

Ce chapitre a pour propos de présenter une nouvelle approche du problème d'identification/construction d'instance ; c'est-à-dire du problème de la construction d'une instance par exploration d'une hiérarchie de classes. Cette problématique rejoint donc celle des systèmes qui proposent la classe comme modèle mixte : unité d'identification et de construction d'instances (cf. §IV.4).

L'approche proposée est motivée par la recherche d'un idéal vers lequel devrait tendre tout système de représentation des connaissances :

- *Les connaissances doivent être représentées sans préjuger de la façon dont elles vont être exploitées.*

L'intérêt de ce principe idéaliste est de faciliter la *réutilisation*, ou plus généralement la *multi-utilisation* des connaissances décrites dans une base. La notion de *multi-utilisation* signifie que la même base de connaissances peut être exploitée pour résoudre des problèmes différents.

Dans le contexte du problème d'identification/construction d'instances, l'application de ce principe se traduit par :

- *Si la résolution d'un problème d'identification/construction d'instance est le résultat des connaissances représentées, la façon dont sont représentées les connaissances ne doit pas être une conséquence du type de problème que l'on souhaite résoudre.*

En d'autres termes, l'un des objectifs que l'on cherche à atteindre vise à éviter d'avoir à décrire les connaissances que l'on possède sur les objets en fonction d'un type particulier de problèmes que l'on cherche à résoudre.

Le résultat attendu de ce chapitre consiste, d'une part, en la définition d'un contexte adéquat à la mise en place d'un mécanisme général d'identification/construction d'instances et, d'autre part, en la définition du type de raisonnement sur lequel repose ce mécanisme.

Après avoir soulevé les différents problèmes des systèmes qui proposent la classification avec inférence pour résoudre des problèmes d'identification/construction d'instance (cf. §V.2), les bases d'une nouvelle approche sont décrites (cf. §V.3), tant du point de vue représentation que du point de vue raisonnement. L'ensemble de cette étude définit le cadre général de la mise en place d'un mécanisme d'identification/construction d'instances.

2. Inconvénients de la classification avec inférence d'attributs

Dans un premier temps, cette partie montre que les solutions proposées pour résoudre un tel problème ont une incidence sur la façon dont est construite une hiérarchie de classes. L'introduction dans une hiérarchie de classes de connaissances, implicites ou explicites, liées à la résolution d'un type de problème peut s'avérer un facteur limitatif pour la *réutilisation* ou la *multi-utilisation* de cette hiérarchie.

Dans un second temps, cette partie soulève les limitations du raisonnement classificatoire comme mécanique de résolution de problème d'identification/construction d'instance.

2.1. Aspect représentation

La construction d'une hiérarchie de classes devrait être réalisée pour décrire un ensemble de propriétés que possèdent un ensemble d'instances. Cette construction sera jugée pertinente pour la résolution d'un problème particulier si les propriétés des objets qu'elle propose peuvent être effectivement exploitées pour apporter une réponse au problème posé.

Cet énoncé général concernant la construction et l'exploitation d'une hiérarchie de classes est généralement ramené pour des raisons pratiques à l'interprétation suivante : une hiérarchie de classes doit être construite pour résoudre un type de problème particulier. Une conséquence de cette interprétation est que la hiérarchie de classes intègre alors aussi bien des connaissances relatives aux propriétés des objets qu'elle représente que des connaissances relatives à la façon dont un problème doit être résolu. Le problème de l'identification/construction d'instances est une illustration de cette interprétation.

2.1.1. Modélisation guidée par le problème de construction

Dans les modèles où la classe est utilisée comme support de construction d'instances (cf. §VI.3.2), une hiérarchie de classes est constituée de deux types de classes : classe abstraite et classe concrète. L'étude de cette distinction permet de mieux comprendre l'impact que peut avoir le type de problème que l'on cherche à résoudre sur la construction même d'une hiérarchie de classes.

Dans les modèles où la classe est un support à la fois d'identification et de construction d'instances, la distinction entre classes concrètes et abstraites n'existe pas explicitement. Toutefois, le recours à un nouveau critère de distinction, basé sur la capacité d'une classe à répondre à un problème de construction d'instances, permet d'expliquer comment la prise en compte d'un problème de construction influe sur la constitution d'une hiérarchie de classes. Les inconvénients d'un système à objets, qui adopte cette modélisation des objets comme principe de représentation, sont illustrés à travers le modèle FROME et l'exploitation de SHIRKA dans l'application MYOSIS.

2.1.1.1. Introduction aux notions de classe abstraite et classe concrète

Dans les modèles à objets proposant les classes comme un moule de construction d'instance, plusieurs types de classes coexistent dans une même hiérarchie. En effet, seules certaines classes sont pertinentes pour une application spécifique, les autres ont pour rôle de factoriser et de rassembler les descriptions des premières.

> "Souvent, [...], certaines classes ne sont créées que pour en faire des "réservoirs d'héritage" pour les autres classes. En LOOPS et en Smalltalk-80, elles sont alors dites *abstraites*. Instancier de telles classes n'a généralement pas de sens et ne présenterait de toute façon guère d'intérêt." [Masi&89]

Ainsi, dans les classes abstraites, certaines méthodes n'ont que leur sélecteur de spécifié ; le traitement étant particulier aux sous-classes, il y est délégué. Ces méthodes des classes abstraites sont dites *virtuelles*, ou *différées*.

Le langage ROME [Carré&87] distingue par deux méta-classes, **R-CLASSE** et **I-CLASSE**, les classes abstraites et les classes concrètes. Elles sont définies ainsi :

- **Classe abstraite** : C'est une classe qui n'a que la fonctionnalité d'abstraction. C'est-à-dire qu'elle représente un concept général indépendamment de toutes classifications ultérieures ; elle regroupe tous les objets correspondants à ce concept ; elle peut décrire une caractérisation factuelle (variables d'instance, attributs, champs) et conceptuelle (sélecteurs-méthodes) commune à tous ces objets. Une classe abstraite n'est pas susceptible d'être instanciée.
- **Classe concrète** : C'est une classe qui a la fonctionnalité d'abstraction, mais aussi celle d'instanciation (ou génération).

Pour appréhender cette notion de classe concrète, il est nécessaire de connaître l'utilisation qui est faite des classes ; c'est effectivement cette utilisation qui pourra leur donner leur statut concret ou abstrait.

Dans l'exemple ci-dessous (cf. Figure V.1), le raisonnement de conception électronique se décompose en une phase théorique dont la classe RESISTANCE fait partie, et en une phase pratique dont la classe RPRATIQUE est la classe de plus haut niveau.

Dans la première phase, la classe RESISTANCE est suffisamment spécifique pour que les traitements de cette phase soient bien spécifiés. Dans cette classe, les méthodes ne sont pas virtuelles.

Lorsque la deuxième phase est entamée, la classe RPRATIQUE est trop générale, un type particulier de résistance doit être choisi, RCOURANTE ou RPRECISION. Le choix de l'une de ces deux classes fournit les comportements nécessaires de l'instance pour le bon déroulement de la phase pratique. Ainsi, les classes concrètes décrivent des états stables des instances pour le raisonnement effectué.

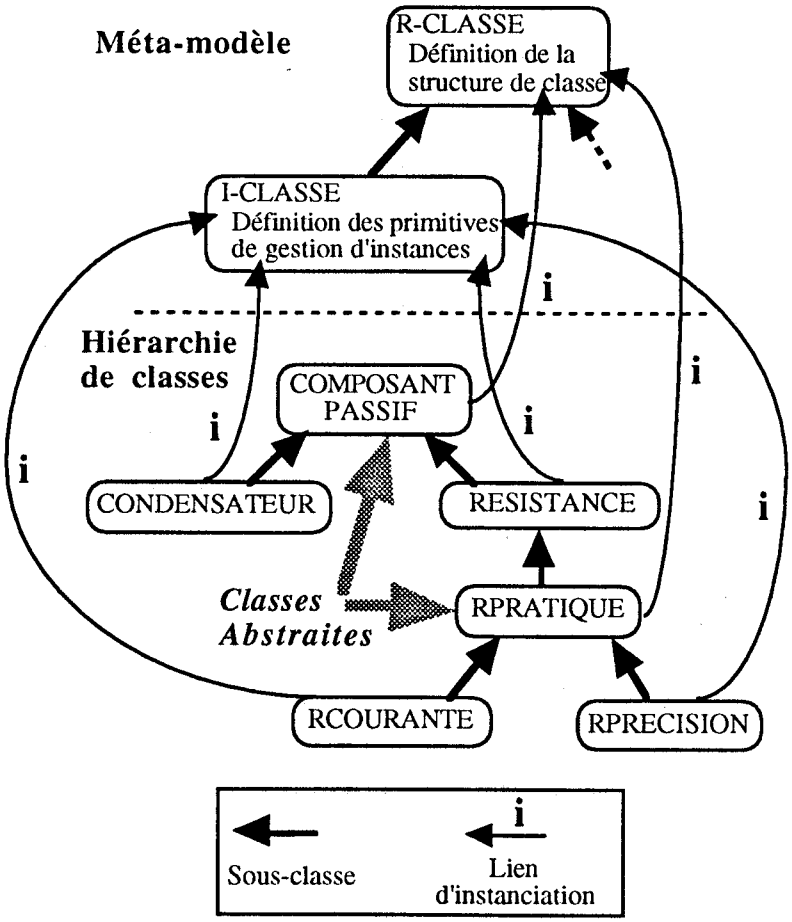

Figure V.1 : Définition de classes abstraites et concrètes en ROME à l'aide des méta-classes respectives R-CLASSE et I-CLASSE. Les classes RPRATIQUE (pour résistance phase pratique) et COMPOSANT-PASSIF sont abstraites (instances de R-CLASSE), elles ne peuvent être instanciées. En contre partie, les autres classes sont instanciables (instances de I-CLASSE).

D'un point de vue ensembliste, lorsqu'on utilise une telle hiérarchie de classes, les classes abstraites représentent l'union de leurs sous-classes concrètes. Ainsi, l'ensemble des COMPOSANT-PASSIF effectifs est strictement égal à l'union des RESISTANCE et des CONDENSATEUR, seules sous-classes instanciables de la phase théorique. La création d'une

instance de COMPOSANT-PASSIF correspond donc à la création d'une instance soit de RESISTANCE, soit de CONDENSATEUR.

D'un point de vue conceptuel, cette distinction met en évidence les frontières de l'univers modélisé. En effet, seules les classes concrètes sont instanciables, donc l'ensemble global des instances potentielles est modélisé par l'union des instances potentielles de chaque classe concrète. Outre son rôle de factorisation d'informations, une classe abstraite peut être perçue comme un choix exhaustif de classes concrètes ; c'est-à-dire l'ensemble de toutes ses sous-classes concrètes. Plus on descend dans la hiérarchie de classes, plus les classes abstraites que l'on y trouve réduisent l'éventail de classes concrètes possibles.

Il est intéressant de noter que le statut d'une classe peut évoluer en fonction de la phase d'utilisation de la hiérarchie de classes. L'exemple des composants électroniques le montre par la prise en compte des classes concrètes des deux phases du raisonnement. Dans la phase théorique, la classe RESISTANCE est concrète, mais dans la phase pratique elle ne correspond plus à la description d'une résistance réelle. Elle n'est donc plus réellement concrète.

La distinction entre types de classes est une information contextuelle puisqu'elle est dépendante de l'objectif à atteindre. Elle n'a de réalité que dans le contexte d'un raisonnement particulier, voire même, simplement le temps d'une phase de ce raisonnement. La définition de classe concrète pourrait alors être reformulée ainsi : une classe est concrète si elle décrit un état de l'instance comportant les informations nécessaires et suffisantes à l'une des phases du raisonnement mise en place par une application. Contrairement à une classe abstraite, une classe concrète est construite pour répondre aux exigences de l'application.

2.1.1.2. Influence du problème d'identification/construction sur la hiérarchie de classes

Les systèmes, comme SHOOD, SHIRKA (à travers l'application MYOSIS, cf. §IV.4.1.2) ou FROME, qui permettent d'exploiter les mécanismes d'inférences d'attributs pour compléter une instance lors de sa classification, favorisent une description des connaissances orientée par la notion de problème. Le propos de cette partie est de montrer les conséquences d'une telle orientation.

- **Critère de distinction classe abstraite/concrète**

Dans un système dans lequel la classe sert aussi bien de support de construction que d'identification d'instances, la distinction entre classes abstraites et classes concrètes ne peut plus se faire sur le critère d'instanciation ou non d'une classe. En effet, toutes les classes de la hiérarchie ont la capacité d'accueillir des instances : une classification peut débuter à n'importe quel niveau d'abstraction, et l'instance est susceptible d'être rattachée à n'importe quelle classe.

Dans ce type de modèle, la perte d'une distinction entre classes concrètes et classes abstraites peut s'expliquer par le fait que chaque classe d'une hiérarchie peut participer au raisonnement de classification d'instances. Par conséquent, du point de vue de ce raisonnement, toute classe d'une hiérarchie est concrète. Elle est effectivement censée décrire les informations nécessaires et suffisantes que doit comporter une instance pour lui appartenir. Ainsi, l'univers d'instances modélisé par une hiérarchie de classes ne connait pas a priori les limites explicitement décrites par les classes concrètes des langages précédents (cf. §V.2.1.1).

Toutefois, lorsqu'une hiérarchie de classes permet aussi la construction d'objets, ce rôle n'est généralement pas partagé par l'ensemble des classes de la hiérarchie. Du point de vue de la construction d'une instance, certaines classes sont abstraites car elles ne décrivent pas une phase de construction d'instance, tandis que d'autres sont concrètes car elles apportent les connaissances nécessaires à la construction de l'instance. Ces connaissances se traduisent par l'introduction dans la description de la classe concrète de moyens d'évaluation d'attributs.

Pour résoudre un problème d'identification/construction d'instance, la solution adoptée consiste à exploiter la classification d'instance pour trouver parmi les classes de la hiérarchie celle qui est concrète pour le problème de construction posé. Il faut donc nécessairement, d'une part, qu'une classe de la hiérarchie soit concrète pour ce problème et, d'autre part, qu'elle

puisse être atteinte par le raisonnement d'identification. Pour réunir ces conditions, la hiérarchie de classes doit avoir été conçue pour résoudre ce type de problème.

En effet, la hiérarchie de classes est construite autour d'un ensemble de modèles d'instances, les classes concrètes, qui sont suffisamment précises pour permettre de compléter une instance particulière à partir de la donnée d'un ensemble minimal d'attributs. Ces derniers servent à guider l'exploration de la hiérarchie jusqu'aux classes concrètes. C'est donc la description de cet ensemble d'attributs qui permet de constituer la hiérarchie au niveau abstrait, décomposant ainsi l'ensemble des instances jusqu'aux classes concrètes. Cet ensemble réduit d'attributs dénote l'ensemble des entrées du problème d'identification/construction.

Cette modélisation des objets permet de garantir le succès d'une démarche classificatoire pour résoudre certains problèmes d'identification/construction d'instances. Toutefois, elle favorise la représentation des objets selon un *point de vue* particulier : la résolution d'un type de problèmes. Il convient de s'interroger sur les conséquences de cette approche.

- **Introduction explicite de connaissances liées au problème**

Le système FROME adopte une solution qui met en évidence la structure particulière d'une hiérarchie de classes exploitée pour résoudre un problème d'identification/construction d'instances. Ce système permet effectivement de définir une classe en distinguant les attributs qui participent à son identification et les attributs qui participent à sa construction. Dans ce cas, un problème d'identification/construction d'instance est explicitement décrit dans la hiérarchie en fonction de ses entrées et ses sorties. La hiérarchie de classes peut donc être scindée en deux points de vue : celui de l'identification et celui de la construction (cf. Figure V.2).

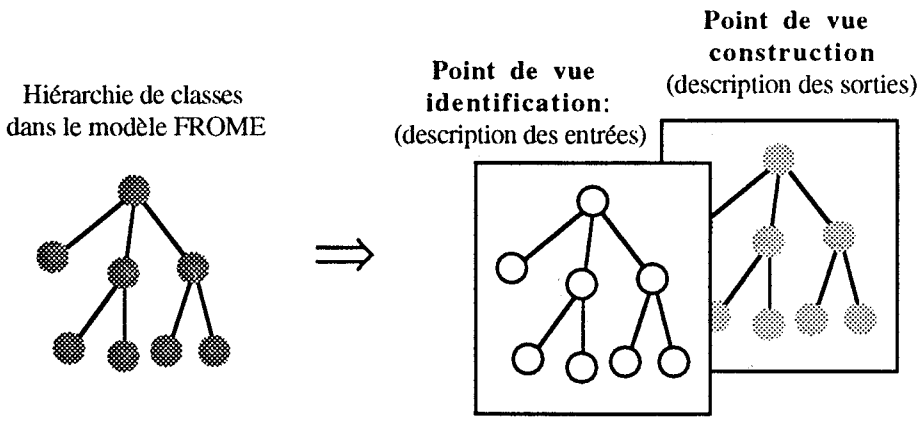

Figure V.2 : la distinction faite par le modèle FROME entre deux types d'attributs d'une classe permet à la classification d'exploiter une partie de la description de la classe pour établir l'appartenance de l'instance et une autre partie pour compléter l'instance. Cette division de la description des classes d'une hiérarchie peut être perçue comme deux points de vue sur l'ensemble des objets. Le premier permet, grâce à la classification, d'identifier le problème de construction d'instance posé au vu des entrées, le second permet de compléter l'instance en fonction des résultats obtenus sur le premier.

L'approche de FROME possède deux avantages : elle permet de traiter correctement le problème de l'exploitation de mécanismes d'inférences d'attributs lors de la classification d'une instance et impose une rigueur méthodologique dans la construction d'une hiérarchie. Toutefois, ce dernier avantage s'avère aussi un inconvénient car la rigueur imposée à la construction d'une hiérarchie repose sur la description de la façon dont doit être résolu un type de problème particulier. En effet, l'introduction d'un attribut dans la description d'une classe de la hiérarchie amène la question suivante : cet attribut est-il une entrée ou une sortie du problème d'identification/construction d'instance ?

La réponse à cette question pour chaque attribut introduit dans la hiérarchie implique l'orientation de celle-ci. L'exemple suivant (cf. Figure V.3) montre que cette orientation n'est pas toujours souhaitable.

Cet exemple repose sur une hiérarchie de classes chargée de décrire les connaissances concernant les équations du second degré. Notamment, elle décrit par rapport au calcul du déterminant les différents cas pouvant apparaître lors de la résolution d'une telle équation.

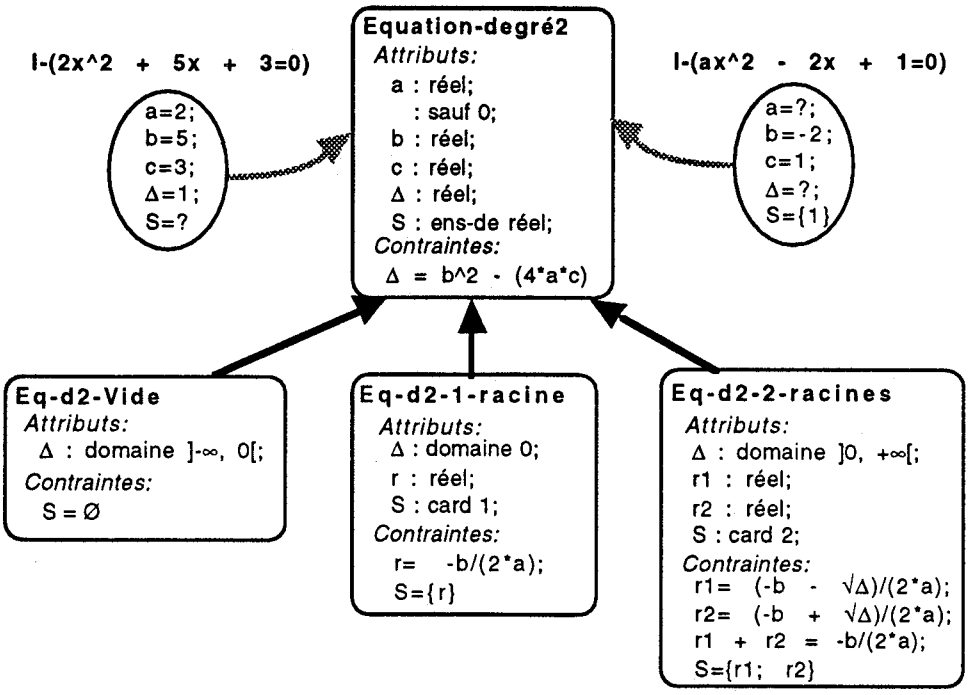

Figure V.3 : Exemple d'une hiérarchie de classes dans laquelle la distinction proposée par FROME entre attributs d'entrées et attributs de sorties limiterait l'exploitation des connaissances décrites. Les deux instances d'équation I-($2x^2 + 5x + 3 = 0$) et I-($ax^2 - 2x + 1 = 0$) illustrent deux problèmes différents d'identification/construction d'instances qui pourraient être résolus par cette hiérarchie. Ces problèmes (duaux) ne se définissent pas en fonction des mêmes entrées et sorties et pourtant à partir des mêmes attributs.

Pour exploiter cette hiérarchie en vu de compléter une instance d'équation du second degré, il faudrait indiquer dans FROME les attributs déterminants pour la classification. Pour ce faire, il est nécessaire de savoir quel problème d'identification/construction la hiérarchie doit permettre de résoudre.

Dans ce cas, plusieurs possibilités sont envisageables. En effet, on peut vouloir résoudre l'équation ou encore construire l'équation en fonction de la solution. Les deux instances présentes dans l'exemple (cf. Figure V.3) montrent qu'en fonction du choix qui sera fait l'une au moins des deux ne pourra pas être complétée par classification. Pourtant, la hiérarchie de classes possède les connaissances pour compléter les deux instances. L'orientation que donne à une hiérarchie de classes le choix du statut d'entrée ou de sortie pour un attribut, limite alors son utilisation.

- **Introduction implicite de connaissances liées au problème**

Dans le cas de l'application MYOSIS (cf. §IV.4.1.2), les entités électromyographiques ne sont pas explicitement représentées en termes d'entrées et de sorties du problème de diagnostic. La raison principale à cela réside dans le fait que le système de représentation utilisé, SHIRKA, ne permet pas de distinguer au sein d'une classe les attributs qui participent à l'identification de ceux qui participent à la construction des instances.

Néanmoins, il serait illusoire de penser que la représentation des objets proposée est ainsi plus générale qu'elle ne l'eut été en utilisant le modèle FROME. En effet, la modélisation sous-jacente à cette application est orientée, suivant le même principe, par le processus de classification nécessaire au raisonnement electromyographique. Chaque hiérarchie de l'application est conçue pour que la classification d'une instance permette l'acquisition progressive des données d'entrée du problème de diagnostic. Ce processus d'acquisition prend fin lorsque la classification atteint une classe concrète (pour ce problème) qui permet alors de construire une solution ; c'est-à-dire inférer les attributs de sortie du problème.

La classe **Paresthésie-3-premiers-doigts** est un exemple de classe concrète de la hiérarchie des signes cliniques. Dans cette classe, l'attribut **évoque** est l'attribut de sortie de cette phase du diagnostic. Pour atteindre cette classe par classification, il faut nécessairement que l'utilisateur ait fourni les attributs d'entrée du problème.

Une analyse plus poussée de la modélisation des objets dans l'application MYOSIS consiste à étudier l'impact du mode de classification utilisé sur la constitution des hiérarchies de classes. Le système SHIRKA propose deux modes de classification : avec ou sans inférence. Contrairement au premier, le second mode permet d'exploiter les mécanismes d'inférence d'attributs associés aux classes lors de la classification d'une instance. Ce mode, utilisé dans l'application MYOSIS, permet lors d'une phase de raffinement de limiter l'ensemble des informations recueillies auprès de l'utilisateur à l'ensemble des attributs apparaissant comme non-inférables dans les sous-classes possibles.

De façon générale, l'emploi d'un tel mode de classification impose une construction particulière des hiérarchies de classes dans laquelle l'effort consiste en grande partie à éviter des déclenchements inopinés de mécanismes d'inférences. Dans le cas de l'application MYOSIS, il est d'autant plus important de prendre ce type de précautions que les mécanismes d'inférences employés consistent à propager le processus de classification d'une hiérarchie de classes à une autre.

Un nouvel aspect doit donc être pris en compte dans la modélisation des objets : le contrôle des déclenchements d'inférence lors de la classification. L'étude de la construction des hiérarchies de MYOSIS montre l'influence de ce mode sur la modélisation et permet de dégager des règles générales de construction qui permettent (presque) de garantir un bon fonctionnement de l'application :

- la sur-classe d'une classe concrète doit réunir l'ensemble des attributs décrivant les données d'entrée du problème représenté par cette classe concrète.

  => cette sur-classe joue le rôle de condition de déclenchement d'inférence. Si elle est déclarée "sûre" alors le prochain raffinement permettra de déclencher les inférences de la classe concrète en disposant de toutes les données nécessaires, si elle est "possible" (ou "impossible") le raffinement au niveau de la classe concrète n'a pas lieu.

- si deux classes concrètes sont sous-classes directes d'une même classe, au moins un attribut de cette dernière doit fournir un critère garantissant qu'une seule de ces deux sous-classes pourra être possible lors d'une phase de raffinement.

  => Pour que la sur-classe soit sûre (et donc que le raffinement puisse avoir lieu sur ses sous-classes), tous ses attributs doivent être évalués dans l'instance. Par conséquent, l'attribut (ou les attributs) fournissant le critère de disjonction ensembliste des sous-classes est nécessairement évalué lors du raffinement. Dans la hiérarchie des signes-cliniques de MYOSIS, un attribut, dont le domaine est l'ensemble des noms des classes concrètes, est introduit à cet effet. Son évaluation revient donc à choisir explicitement l'unique classe concrète possible !

- etc.

Cette modélisation a pour conséquence l'introduction de classes et d'attributs supplémentaires dont le but essentiel est de contrôler le déclenchement des inférences d'attributs. Il est intéressant de constater que le système FROME conviendrait mieux à ce genre d'application puisque la distinction explicite entre données d'entrée et de sortie permettrait

d'exprimer en une classe ce qui doit être exprimé en deux (la classe concrète et sa sur-classe) avec le système SHIRKA.

2.1.2. Propriétés cachées de la hiérarchie de classes

La représentation que propose ces systèmes ne permet pas d'exprimer explicitement, et donc complétement, certaines propriétés des objets. Or, il s'avère que certaines de ces propriétés peuvent expliquer le raisonnement sur lequel se fonde la résolution d'un problème d'identification/construction d'instances.

Dans l'exemple de la hiérarchie des équations du second degré (cf. Figure V.3), une propriété importante est occultée : les extensions des classes **Eq-d2-vide**, **Eq-d2-1-racine** et **Eq-d2-2-racines** forment une partition de l'extension de la classe **Equation-degré2**. Si cette propriété pouvait être exprimée par la représentation et exploitée par la classification, les deux problèmes d'instances incomplètes, présentés précédemment, pourraient alors être résolus. En effet, pour chacune d'elles, le raisonnement est le suivant :

1) l'instance appartient à la classe **Equation-degré2** ;
2) seule une sous-classe reste possible pour cette instance (**Eq-d2-1-racine** ou **Eq-d2-2-racines** selon le cas).
3) l'extension de **Equation-degré2** se partitionne en trois sous-ensembles correspondant aux extensions de ses trois sous-classes.
4) **alors** l'instance appartient à la seule sous-classe possible.

A ce titre, on pourrait penser que le mécanisme de classification (mode "avec inférence") proposé par SHIRKA permet de rendre compte de cette propriété de la hiérarchie puisque son application permettrait aussi de résoudre les deux problèmes d'instances incomplètes. Il ne faut pas s'y tromper, ce mécanisme est incapable de traduire tous les aspects que confère cette partition à la logique de représentation.

Pour preuve, il suffit de considérer la donnée d'une instance appartenant à la classe **Equation-degré2** et dont l'appartenance à chacune des sous-classes est prouvée impossible. Il suffit pour cela de considérer une équation dont le déterminant est strictement positif et l'ensemble de solution vide. Cette instance est incohérente d'après la propriété de partition des exetensions. Toutefois, la classification ne pourra traduire cet aspect, acceptant comme valide une assertion qui ne devrait pas l'être.

2.1.3. Conclusion

Les systèmes qui proposent le mécanisme de classification d'instances pour résoudre un problème d'identification/construction d'instances, se reposent sur une construction particulière des hiérarchies de classes. Il est effectivement nécessaire d'ajouter, explicitement ou implicitement, des informations qui n'ont pour autre but que de contrôler la façon dont doit s'appliquer le raisonnement classificatoire. Cette approche mériterait en ce sens le qualificatif de "résolution de problèmes dirigée par la classification" ; qualificatif permettant de rappeler que la *classification heuristique* [Clan85] est certainement la principale source d'inspiration de cette approche.

Du point de vue de la représentation, cette approche prône une "représentation par objets guidée par la notion de problème". Le principe d'indépendance entre la représentation des objets et leur utilisation, énoncé dans l'introduction de ce chapitre (cf. §V.1), n'est donc pas vérifié par cette approche.

2.2. Aspect raisonnement

2.2.1. Limite du raisonnement classificatoire

L'approche du problème d'identification/construction d'instances, proposée dans une application comme MYOSIS ou dans un modèle comme FROME, consiste à préalablement

construire une hiérarchie de classes décrivant la résolution de ce problème. Résoudre un problème consiste alors à choisir la hiérarchie de classes sur laquelle lancer la classification de l'instance à construire. En d'autres termes, le problème est représenté par la hiérarchie de classes.

Schématiquement, une partie de cette hiérarchie est dédiée à l'identification du problème et l'autre partie est dédiée à la construction de la solution. Les classes correspondant à la construction d'une solution forment les classes concrètes (pour le type de problème à résoudre) de la hiérarchie. Dans ce contexte, la construction d'une solution signifie que l'instance est complète.

La hiérarchie de classes joue, dans cette approche, deux rôles : elle permet de définir le problème et elle permet de le résoudre. La définition du problème passe effectivement par les informations que la classification d'une instance va requérir auprès de l'utilisateur. Ces informations sont ensuite exploitées pour déterminer ce qui doit être inféré (les attributs de l'instance) et comment les inférer. La résolution d'un problème d'identification/construction d'instances est gérée de ce fait comme un **raisonnement déductif dirigé par les données**.

Cette approche de la résolution est réductrice car elle n'est pas adaptée au cas d'un problème à solutions multiples. La classification d'une instance ne dispose ni des moyens de distinguer diverses voies de solution, ni des moyens de construire à part les différentes solutions.

Un problème réel auquel doit faire face l'application MYOSIS réside dans la possibilité de construire une instance de signe clinique rassemblant un ensemble d'informations qui permette d'atteindre par des branches différentes deux classes concrètes. C'est-à-dire que les signes cliniques d'un patient évoquent alors deux hypothèses de maladie différentes. Ce problème traduit soit que le patient est effectivement atteint par deux maladies, soit que l'une des hypothèses pourra être réfutée après la série de tests complémentaires utiliser pour la vérifier. Comme un tel cas entraînerait un dysfonctionnement de la classification, la solution adoptée dans les versions suivantes de MYOSIS a consisté à limiter grandement le rôle de la classification pour contrôler de façon externe la construction des entités electromyographiques.

Pour ce qui est du traitement de problèmes à solutions multiples, le système SHOOD fait figure de précurseur. A travers le traitement des conflits qu'il propose, un caractère important de la problématique de l'identification/construction d'instances apparaît : la présence et l'exploitation d'incohérences dans l'identification/construction d'une instance ; c'est-à-dire la présence et la gestion d'hypothèses. On peut reprocher à SHOOD la spécificité de la solution proposée et de ne pas généraliser son principe en introduisant explicitement la notion d'hypothèse. Si le fonctionnement non monotone du mécanisme de classification proposé lui confère une certaine puissance, il favorise en contre-partie une description des objets pour laquelle il devient difficile de garantir un maximum de cohérence.

Il convient donc de séparer clairement classification d'instance et résolution d'un problème d'identification/construction d'instance.

2.2.2. Rôle des inférences d'attributs dans l'identification

Le modèle FROME permet d'éclater la description de la classe de telle façon que les connaissances d'identification soient clairement séparées des connaissances de construction. Cet éclatement constitue un filtre utilisé par le mécanisme de classification d'instance qui établira l'appartenance d'une instance à une classe en se focalisant uniquement sur les connaissances d'identification que cette classe requiert. Si l'appartenance est établie, l'instance peut bénéficier des connaissances de construction de la classe.

Le schéma de description d'une classe est ainsi similaire à une règle du type "*Si* DCI(I) *alors* [*si-besoin*] DCC(I)" dans laquelle DCI(I) représente l'application à l'instance I de la description des connaissances d'identification de la classe et DCC(I) représente l'application à l'instance I de la description des connaissances de construction de la classe. Le terme [*si-besoin*] est introduit dans la règle pour souligner que seules seront utilisées les

connaissances de DCC dont a besoin I. Bien que ce ne soit pas explicité dans leurs hiérarchies de classes, l'application MYOSIS et le modèle SHOOD exploitent la notion de classe mixte de la même façon au travers de leur mécanisme de classification d'instances.

Au delà des problèmes de construction de hiérarchie que cette approche implique, elle soulève aussi certaines interrogations :

- Comment distinguer dans la description d'une classe les éléments qui participent à l'identification d'une instance de ceux qui participent à sa construction ?
- Si l'application des connaissances de construction d'une classe constitue une condition nécessaire de l'appartenance d'une instance à la classe, quelle importance accordée à un résultat qui contredit les informations sur lesquelles l'appartenance a été constatée ?

En pratique, les connaissances de construction sont représentées dans la description d'une classe par au moins l'ensemble des attributs auxquels est associé un ou plusieurs moyens d'évaluation. Les connaissances d'identification seront formées, quant à elle, par l'ensemble ou un sous-ensembles des attributs restants.

Toutefois, certaines nuances peuvent être apportées à la définition des connaissances d'identification : l'attribut est pertinent pour les instances de la classe (FROME), l'attribut doit nécessairement être évalué (FROME, SHOOD). La notion d'attribut pertinent permet d'inclure dans les connaissances d'identification d'une classe un attribut dont la description intègre aussi des moyens d'évaluation. La subtilité dans ce cas réside dans le fait que sa présence dans l'instance est la seule condition qui est requise sur cet attribut pour établir l'appartenance de l'instance à la classe. Aussi, s'il est donné avec une valeur inconnue, celle-ci pourra alors être calculée par les moyens fournis par la classe. En revanche, si sa valeur est fixée avant que l'instance soit confrontée à la classe, les moyens d'évaluation ne sont alors pas déclenchés.

Le problème de cette approche est que les mécanismes d'évaluation qui peuvent être associés à un attribut dans une classe sont systématiquement relégués au cas d'un mécanisme par défaut et ne participent pas à la définition de l'état cohérent des instances de la classe.

Dans cette approche, une modélisation de la classe des équations du second degré consisterait à associer à l'attribut Δ un attachement procédural chargé d'exprimer le calcul du déterminant en fonction des attributs a, b et c. Dans ce cas, la donnée par l'utilisateur d'une instance décrivant l'équation "$2x^2 + 5x + 3 = 0$" et de déterminant fixé à 0, ne provoquerait aucune vérification concernant la cohérence de cette assertion puisque le calcul de Δ ne serait pas déclenché. Pourtant, cette instance ne représente pas une équation du second degré car son déterminant devrait être égal à 1 au vu des coefficients donnés.

L'appartenance d'une instance à une classe ne peut donc pas toujours être prouvée par un appariement basé uniquement sur une confrontation, d'une part, entre la structure de l'instance et celle décrite par la classe et, d'autre part, entre la valeur des attributs de l'instance et le type qui leur est individuellement associé dans la classe. Si la classe décrit, par un moyen ou un autre, des dépendances inter-attributs, celles-ci devront aussi être satisfaites pour toutes ses instances.

Cette remarque peut être généralisée au cas des propriétés particulières que peut posséder une classe dans la hiérarchie de spécialisation. Effectivement, pour prouver l'appartenance d'une instance à une telle classe, il faudrait alors vérifier qu'elle se conforme aux propriétés de la classe. Par exemple, si l'exhaustivité de la classe des équations du second degré est explicitement déclarée, cette propriété constitue un critère de cohérence important pour établir l'appartenance d'une instance à cette classe.

Si l'utilisateur désire savoir si une instance, représentant l'équation "$2x^2 + 5x + 3 = 0$" et pour laquelle l'ensemble de solutions est réduit à l'ensemble vide, est une équation du second degré, l'application de l'exhaustivité est importante pour prouver que cette instance n'en est pas une. En effet, l'exhaustivité ne peut être satisfaite car toutes les sous-classes seront déclarées impossibles. D'une part, le déterminant de cette équation est calculé et égal à 1, donc l'instance ne correspond ni au cas des équations sans solution, ni au cas des équations à racine double.

D'autre part, puisque son ensemble de solution est vide, elle ne correspond pas non plus au cas des équations à deux racines distinctes dont l'ensemble de solutions est de cardinalité 2.

L'un des problèmes des systèmes qui proposent un modèle de classe mixte réside dans le fait qu'il cherche à caractériser à partir de peu d'éléments les conditions nécessaires et suffisantes de l'appartenance à ce type de classe. La description d'une classe garantit que ses instances sont cohérentes dans la limite où seules les connaissances nécessaires à leur identification ont été fournies.

Que ce soient des connaissances concernant l'évaluation d'attributs ou des propriétés de la classe, le second rôle qui est accordée à ces connaissances ne permettra pas au système d'en disposer dans n'importe quelle condition d'exploitation. On sous-estime l'importance de la cohérence dont ces connaissances peuvent être porteuses. Dans le cadre de l'identification/construction d'instance, la détection d'une incohérence lors de la construction permet de réfuter une hypothèse d'identification.

3. Nouvelle approche

En réponse aux problèmes soulevés dans la partie précédente, la nouvelle approche de la problèmatique se décompose en des propositions, d'une part, sur le type de représentation par objets adéquat et, d'autre part, sur le type de raisonnement à mettre en place.

3.1. Aspect représentation

Dans cette partie, nous posons les bases d'une représentation par objets adéquate pour l'identification/construction d'instances. La démarche suivie consiste à séparer la représentation des objets de celle d'un problème. L'objectif est de généraliser la notion de problème afin de se placer dans un cadre où on ne présuppose aucune construction particulière des hiérarchies de classes utilisées. Afin de renforcer l'indépendance de la représentation vis-à-vis des problèmes, l'expression de propriétés sur la hiérarchie de classes est proposée.

Enfin, pour permettre à une classe d'être une unité à la fois d'identification et de construction, l'approche adoptée consiste à considérer une description de classes comme un ensemble de contraintes qui assure à la fois la vérification et la production d'informations. Le comportement des éléments usuels de description d'une classe est alors établi : attributs, facettes de domaine, exploitation de mécanismes d'évaluation d'attributs...

3.1.1. Représentation d'un problème d'identification/construction

Dans une approche où la hiérarchie de classes n'est pas spécialement construite pour résoudre un type de problème particulier, un problème d'identification/construction d'instance doit être formulé explicitement. Un tel problème est caractérisé par la donnée d'une instance sur laquelle l'utilisateur indique l'ensemble des informations qu'il connaît et indique les informations qu'il ne connaît pas et désire connaître.

> Problème d'identification/construction d'instance =
>
> (configuration initiale de l'instance, ensemble d'attributs à évaluer)

Dans cette démarche, le problème d'identification/construction d'instance est complètement défini par la description de deux types d'informations souhaitées par l'utilisateur : l'identité exacte de l'instance (c'est pourquoi il classe l'instance) et les attributs dont il déclare vouloir connaître la valeur. Ce problème est donc précisé en termes de buts : les attributs à évaluer.

L'intérêt de la classification, si elle ne permet pas de résoudre le problème, provient du fait qu'elle permet à l'utilisateur de formuler progressivement son problème. En effet, elle lui offre un support permettant d'apporter sur l'instance les informations qu'il possède au fur et à mesure de l'exploration de la hiérarchie de classes. Il peut donc dans ce contexte définir son problème en stipulant les informations de l'instance qu'il ne possède pas et qu'il souhaiterait déterminer.

3.1.2. Propriétés d'une classe dans la hiérarchie

L'étude des solutions proposées dans FROME et dans MYOSIS montre que la relation de spécialisation et la description des classes s'avèrent insuffisantes à exprimer complètement des relations particulières qu'une classe peut entretenir avec ses sous-classes. Cependant, ces relations sont importantes car elles permettent d'exprimer des connaissances générales sur les objets dont l'exploitation peut permettre l'inférence de nouvelles connaissances.

Les relations, considérées par la suite, peuvent être exprimées comme une propriété particulière de la décomposition d'une classe en sous-classe. Ces propriétés peuvent être :

- L'**exclusivité** de la décomposition d'une classes en sous-classes : les sous-classes directes d'une classe représentent des ensembles disjoints deux à deux. Dans le système TROPES, utilisé par la suite, cette propriété est une hypothèse de base et est donc implicitement imposée à toute hiérarchie. Elle est utilisée pour propager la marque impossible lors de la classification.
- l'**exhaustivité** de la décomposition d'une classes en sous-classes : les ensembles représentés par les sous-classes directes d'une classe forment un recouvrement de l'ensemble représenté par cette dernière. Puisque dans le modèle TROPES l'exclusivité est une propriété générale de la hiérarchie de classes, déclarer qu'une classe connaît une décomposition exhaustive en sous-classes revient à donner une partition de l'ensemble représenté par cette classe en autant de sous-ensembles que de sous-classes. Une classe qui connaît une décomposition exhaustive est caractérisée dans la hiérarchie de classes par son type particulier : **classe-exhaustive**.

L'exemple de la hiérarchie des équations du second degré illustre parfaitement l'intérêt de l'exhaustivité. En effet, si la classe **Equation-degré2** est déclarée exhaustive, cette connaissance est suffisante pour compléter les deux instances qui lui sont rattachées. Cette extension du raisonnement classificatoire permet de résoudre deux problèmes d'identification/construction d'instances sans avoir recours au choix d'attributs d'entrée et de sortie.

Deux autres propriétés seront considérées lors de la mise en place effective d'un mécanisme d'identification/construction d'instances dans le modèle TROPES. Elles seront présentées dans ce cadre (cf. §VIII.3.3.2).

3.1.3. La classe est un ensemble de contraintes

La classe doit répondre aux exigences de la classification, elle doit donc être un modèle pour l'identification. La description d'une classe doit aussi permettre la construction et, plus particulièrement, offrir des mécanismes pour compléter une instance. Cette exigence est cruciale pour la mise en place d'un mécanisme d'identification/construction d'instance.

L'approche proposée se refuse à définir une classe en séparant clairement les connaissances d'identification et les connaissances de construction. Une classe décrit des propriétés que l'ensemble de ses instances vérifient indépendamment du contexte dans lesquelles elles sont utilisées.

En conséquence, la classe peut être vue comme un ensemble de contraintes qui sont satisfaites par toute instance lui appartenant. Ces contraintes sont de deux types : les contraintes décrivant les propriétés qu'une classe possède dans sa hiérarchie et les contraintes sur ses attributs.

CLASSE	=	Contraintes extensionnelles sous-jacentes aux propriétés de la classe dans la hiérarchie
	+	Contraintes sur les attributs

De façon générale, les contraintes concernant les propriétés d'une classe dans la hiérarchie incluent aussi bien les contraintes concernant les relations entretenues avec ces sur-classes (spécialisation) que celles concernant ces sous-classes (exclusivité, exhaustivité, etc.). L'ensemble de ces contraintes réduit l'ensemble des états de classification possibles pour une

instance. Ces états sont définis en fonction du statut que la classe aura pour l'instance (sûre, possible, impossible) et des possibilités de marquage des classes de la hiérarchie que permettent d'atteindre les règles de propagation de marques associées à chaque type de propriétés (une classe sûre implique que toutes ses sur-classes soient aussi sûres, etc.).

En ce qui concerne les attributs, une classe peut comporter trois types de contraintes sur les attributs, décrivant les différents niveaux de raffinements intervenant successivement aussi bien dans l'identification que la construction de ses instances. Ces trois types de contraintes, détaillés par la suite, sont les suivants :

- **Contraintes structurelles** : une classe définit la structure minimale, c'est-à-dire l'ensemble des attributs, qu'une instance lui appartenant doit posséder.
- **Contraintes sur le domaine de valeurs des attributs** : une classe contraint la valeur des attributs de ses instances à appartenir à un domaine précis qui est explicité à travers sa description.
- **Contraintes sur l'évaluation des attributs** : une classe peut imposer une stratégie d'évaluation à certains attributs de ses instances.

L'intention de cette partie n'est pas de proposer ici un langage de contraintes structurées à la façon de CONSTRAINTS [Suss&80], mais de restreindre la notion de classe d'une représentation par objets à se comporter comme un ensemble de contraintes et ainsi d'en souligner les capacités duales d'identificateur et de constructeur d'instances.

3.1.3.1. Appariement et rattachement d'une instance à une classe

Avec une telle interprétation de la notion de classe, l'appariement et le rattachement d'une instance à une classe sont deux mécanismes similaires. En effet, un appariement peut être vu comme l'essai d'un rattachement de l'instance à la classe dont le but est de savoir si cette opération est cohérente ou non (impossible ou possible) et si l'ensemble des contraintes posées par la classe sont satisfaites ou non par l'instance (sûr ou possible).

De façon générale, la distinction entre appariement et rattachement d'instance peut se résumer ainsi :

- *L'appariement a pour objectif la vérification par une instance des contraintes décrites par une classe, alors que le rattachement a pour objectif l'application à une instance des contraintes définies par une classe.*

Le résultat d'un appariement d'une instance s'obtient de la façon suivante :

- Si l'instance satisfait l'ensemble des contraintes de la classe (c'est une solution du système de contraintes), la classe est **sûre**.
- Si l'instance vérifie seulement l'ensemble des contraintes (aucune incohérence n'a été rencontrée, mais des informations manquent à l'instance), la classe est **possible**.
- Si l'instance ne vérifie pas une contrainte (une incohérence est rencontrée), la classe est **impossible** pour l'instance.

Dans le contexte de l'appariement d'une instance à une classe, le statut de la classe explorée n'est pas connu, alors que dans le contexte du rattachement, la classe receveuse obtient le statut **sûr**. Contrairement au rattachement, l'appariement de l'instance à une classe ne peut donc pas exploiter le fait que la classe soit **sûre** pour inférer de nouvelles connaissances. En revanche, le rattachement pourra, grâce à ce statut **sûr**, inférer de nouvelles connaissances : ajout d'attributs à l'instance, réduction de domaine de valeurs de certains attributs, obtention de valeurs d'attributs et, éventuellement, traitement du rattachement à une sous-classe dans le cas où le rattachement en cours s'applique à une classe de type exhaustif.

Par la suite, les contraintes relatives aux propriétés définies pour une classe dans une hiérarchie de classes sont supposées maintenues automatiquement à chaque fois que le statut d'une classe est fixée pour une instance. Elles reposent sur des règles de vérification et de propagation de marques qu'un mécanisme de classification d'instance exploite (cf. §II.3.2.2).

Les parties suivantes sont consacrées, quant à elles, à la présentation des trois types différents de contraintes qu'une classe peut imposer à ses attributs.

3.1.3.2. Contraintes structurelles d'une classe

Classiquement, les contraintes structurelles d'une classe consistent à décrire les attributs que la structure de chacune de ses instances doit inclure. Une possibilité d'extension de la notion de contraintes structurelles est proposée pour permettre aussi d'exprimer les attributs que la structure de l'instance ne doit pas posséder.

- **Contraintes structurelles classiques**

Une classe décrit un ensemble d'attributs que doit posséder nécessairement toute instance lui appartenant. Le rattachement d'une instance à une classe impose à l'instance d'inclure à l'ensemble de ses attributs ceux de la classe qu'elle ne possède pas déjà. L'appariement vérifie que l'instance possède les attributs de la classe ; en fonction du résultat, l'instance sera (potentiellement) complète ou incomplète relativement à la classe.

L'ensemble d'attributs qu'une classe décrit est une contrainte structurelle pour l'instance. Par exemple, un chien doit se plier à la contrainte structurelle : pattes, couleur, tête, queue, maître... La classe **Chien** l'exprime par la déclaration des attributs respectifs. La création, ou simplement l'attachement de l'instance **Médor** à la classe **Chien** lui impose au minimum cette structure. Pour espérer être reconnue comme un chien, **Médor** doit au minimum posséder ces attributs.

Des distinctions dans la nature des attributs peuvent être faites, elles ont pour conséquences d'ajouter des contraintes spécifiques à prendre en compte lors de leur exploitation. Généralement, on distingue les attributs propriétés, les attributs composants et les attributs relations. Les premiers sont des attributs classiques d'une classe qui imposent comme seule contrainte leur existence au sein de la structure des instances de la classe. On associe aux seconds et aux troisièmes types d'attributs, composant et relation, des comportements particuliers dénotant des contraintes spécifiques entre les objets en relation.

- **Extension possible des contraintes structurelles**

Classiquement, une classe permet d'exprimer ce qui doit être dans une instance qui lui appartient, mais il peut être intéressant de spécifier ce qui ne doit pas être dans une instance. Pour illustrer ce propos, l'exemple des barres utilisé pour le modèle SHOOD (cf. §IV.4.2.1) est repris et réadapté afin que la modélisation des classes décrivant les différents types de barre fournisse une hiérarchie de spécialisation correcte (cf. Figure V.4).

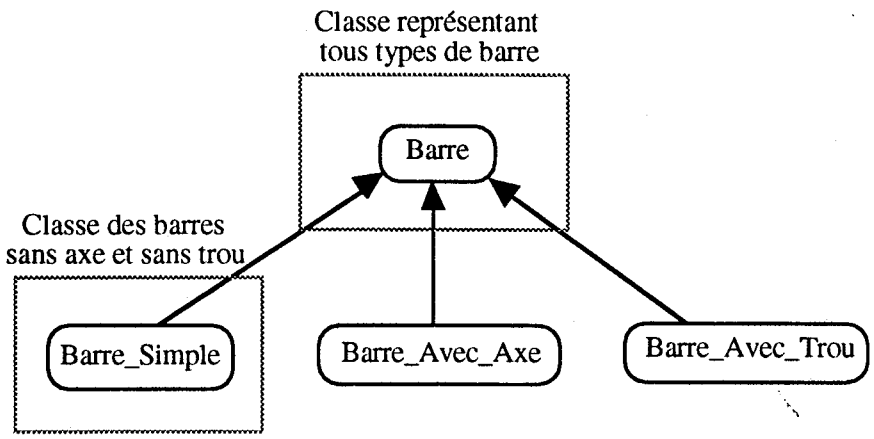

Figure V.4 : Dans cette nouvelle modélisation, la classe **Barre** représente l'ensemble de toutes les barres alors que la classe **Barre_simple** est introduite pour décrire les barres sans axe et sans trou.

Cet exemple constitue le cas type de problème qui aurait tout avantage à bénéficier de la possibilité d'indiquer certaines caractéristiques qui ne doivent pas être dans la description d'une classe. La classe décrivant la barre simple ne comporte que l'attribut **barre**, celle de la barre avec axe ajoute l'attribut **axe**, et enfin celle avec trou ajoute l'attribut **trou**. La classe de barre simple n'exprime pas qu'il n'y a ni axe, ni trou. Ceci pose un problème lors de l'appariement d'une instance de barre ; si celle-ci comporte un trou et/ou un axe, la confrontation avec la classe des barres simples ne la déclarera pas impossible alors que ce devrait être le cas.

Si maintenant on est capable d'exprimer dans une classe que l'objet ne doit pas avoir certains attributs (la classe Barre-Simple de la figure V.5), le problème précédent est résolu : une barre simple n'a ni axe, ni trou ; l'instance ne peut plus lui appartenir (l'instance I1 de la figure V.5).

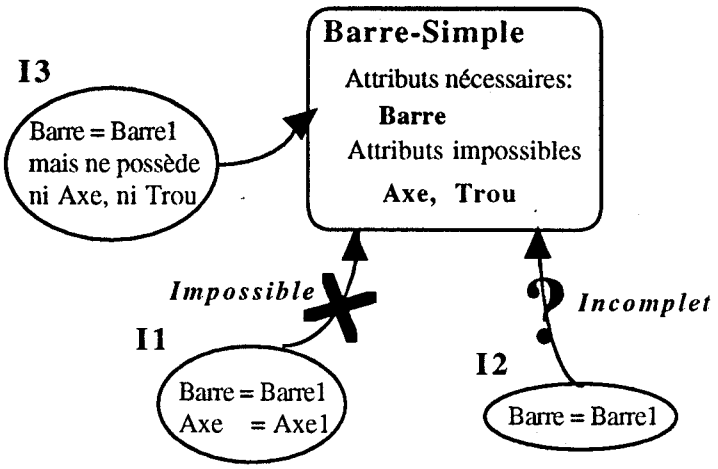

Figure V.5 : Attributs nécessaires et impossibles pour la classe Barre-Simple. Les instances I1, I2 et I3 décrivent les cas respectivement impossible, possible et sûre.

Un nouveau problème se pose alors : si une instance ne comporte qu'une barre (l'instance I2 de la figure V.5), est-ce le cas d'une barre simple ou le cas d'une instance incomplète de barre avec trou et/ou axe ? Cette ambiguïté peut être levée si l'instance, elle-même, possède la connaissance des attributs qu'elle ne doit pas avoir (l'instance I3 de la figure V.5). Si elle ne porte pas cette information, elle est incomplète.

Cette "description négative" est une contrainte structurelle, elle doit être décrite au niveau des classes ; toute instance de la classe doit s'y plier et, par conséquent, l'inclure dans sa propre structure. Aussi, nous proposons de l'intégrer dans une représentation par objets, non pas par l'ajout d'un type particulier d'attribut (qui pourrait s'appeler les "non-attributs"), mais par l'introduction d'une valeur spéciale, **nil**. L'intérêt de cette approche est d'uniformiser le traitement de ces "non-attributs" avec celui des attributs standards d'une classe.

Ainsi, dans l'exemple des barres, la classe des barres simples déclare l'attribut **barre** avec un certain domaine, ainsi que les attributs **trou** et **axe** dont les domaines sont tous deux réduits à la valeur **nil**. Lorsqu'un utilisateur veut spécifier qu'une instance n'a ni axe, ni trou, il affecte dans l'instance la valeur **nil** aux attributs **trou** et **axe**. Dans le doute, l'utilisateur s'abstient ; il n'indique rien à propos des attributs **trou** et **axe**. Ceux-ci n'apparaissent pas dans l'instance traduisant ainsi qu'on ne sait pas si la barre a ou non un axe et/ou un trou. La figure V.6 récapitule la signification de l'information portée dans une instance.

Face à une demande de valeur pour l'attribut A de l'instance I, l'utilisateur peut avoir les réponse suivantes :

- V_A : signifiant que I est connue pour avoir l'attribut A, qui a pour valeur V_A. L'attribut A apparaît dans l'instance I avec la valeur V_A.
- **?** : signifiant que I est connue pour avoir l'attribut A, sa valeur est, par contre, inconnue. L'attribut A apparaît dans l'instance I avec la valeur **?**.
- **nil** : signifiant que I est connue pour ne pas avoir l'attribut A. L'attribut A apparaît quand même dans l'instance I avec la valeur **nil**.
- ***<rien>*** : signifiant qu'on ne sait pas si I possède l'attribut A. L'attribut A n'apparaît pas dans l'instance I.

Figure V.6 : Récapitulatif sur la présence ou non d'un attribut, et sur l'utilisation de **?** et **nil** dans la description d'une instance.

La partie suivante qui traite des contraintes sur le domaine d'un attribut suggère un traitement possible de cette notion dans la description d'une classe.

3.1.3.3. Contraintes sur les domaines de valeurs d'attributs

Généralement, les contraintes sur le domaine de valeurs d'un attribut consiste à ajouter des facettes de typage à la description de l'attribut dans la classe. Vérifier une contrainte de ce genre consiste alors à vérifier le type décrit pas la classe. Le rattachement, quant à lui, ne se distingue pas de ce traitement.

Toutefois, lorsque la description d'une classe permet d'exprimer des contraintes complexes, impliquant plusieurs attributs, le domaine de valeurs d'un attribut d'une instance de la classe dépend d'autres attributs. Dans ce cas, pour suivre les fluctuations du domaine de valeurs d'un attribut en fonction des évolutions de l'instance, le domaine doit être associé à l'attribut dans l'instance pour assurer sa gestion dynamique. Le rattachement d'une instance est donc une opération qui a pour effet la mise à jour de ce domaine.

- **Contraintes de typage sur les domaines d'attributs**

Classiquement, un certain nombre de facettes associées à un attribut contribuent à décrire le domaine de valeurs d'un attribut dans une classe. Ces facettes affinent la description de l'attribut de la (ou les) sur-classe(s) ; elles expriment une réduction du domaine de l'attribut. En général, elles peuvent exprimer des énumérations de valeurs possibles (facette **$domaine**), des énumérations de valeurs impossibles (facette **$sauf**).

Les possibilités d'expressions d'affinement d'un attribut peuvent être plus complexes que de simples énumérations, elles sont essentiellement dépendantes du type de l'attribut. C'est le cas des attributs construits par **ensemble**, ou **liste**, qui peuvent aussi être contraints sur leur cardinalité.

Du point de vue de l'appariement, quelque soit le type, l'appartenance de la valeur de l'attribut au domaine décrit par la classe doit être vérifiée. Du point de vue du rattachement, le nouveau domaine, obtenu après application des facettes de réduction de la classe, est associé à l'attribut dans l'instance. Si ce domaine s'avère vide, l'instance est déclarée incohérente. Le rattachement signale cette incohérence.

- **Proposition d'extension du typage par la valeur *nil***

En ce qui concerne l'introduction de la valeur **nil**, elle constitue une extension de tous les types utilisés dans le modèle. Par exemple, le type entier voit le domaine qu'il décrit, enrichi par cette valeur : $]-\infty, +\infty[\cup \{\textbf{nil}\}$.

Cette valeur que peut prendre un attribut est différente de la notion de domaine vide. En effet, un attribut dont le domaine de valeurs est réduit à l'ensemble vide décrit un cas d'instance incohérente. La valeur **nil**, quant à elle, décrit l'absence affirmée de l'attribut dans l'instance.

Dans la description d'une classe, cette valeur spéciale déclarant l'attribut comme un "non-attribut" est donc introduite par la facette d'énumération du domaine. Une fois que l'attribut est déclaré comme "non-attribut" dans une classe, son domaine est réduit à la valeur **nil**, toutes les sous-classes le considère comme tel ; elles ne peuvent réintroduire l'attribut du fait de la réduction du domaine.

Le traitement de cette valeur spéciale s'uniformise naturellement avec les autres valeurs. Lors de l'appariement d'une instance avec une classe déclarant un "non-attribut", si l'attribut a une valeur différente de **nil** dans l'instance, la classe devient alors impossible. Lors du rattachement, le domaine de l'attribut associé à l'instance est réduit au singleton {**nil**}.

- **Contraintes complexes sur les domaines d'attributs**

Les contraintes complexes sur les domaines d'attributs peuvent apparaître lorsque la description d'une classe autorise l'expression de contraintes inter-attributs. Une proposition d'extension du modèle TROPES par l'intégration d'un module de gestion contraintes va dans ce sens [Gens93].

Cette extension permet, d'une part, d'exploiter au sein d'une classe un langage de contraintes pour exprimer des contraintes inter-attributs et, d'autre part, de bénéficier d'un mécanisme de maintien de contraintes capable de propager les réductions de domaines sur tous les attributs impliqués dans un même réseau de contraintes. Le couplage entre ce module et le système TROPES permet d'assurer la vérification des contraintes lors de l'appariement d'une instance à une classe et l'obtention pour chaque attribut contraint de son domaine réduit lors du rattachement d'une instance à une classe.

L'intégration du module de gestion de contraintes au système TROPES, proposé par Jérôme Gensel, est présentée dans le chapitre VI. Cette intégration fait, toutefois, l'objet d'une révision, dans le chapitre VII, afin de mettre en place un contrôle d'évaluation d'attributs capable de gérer les inférences de valeurs d'attributs lorsque divers types de mécanismes d'inférence coexistent dans le même système.

3.1.3.4. Contraintes sur l'évaluation des attributs

Généralement, les possibilités d'évaluation d'attributs d'une instance sont déterminées par la présence dans sa classe d'appartenance de facettes spéciales comme les facettes **$defaut**, **$si-besoin** ou **$valeur**. L'emploi de ces facettes à travers la hiérarchie de classes permet de décrire pour certains attributs une véritable stratégie d'évaluation.

Cependant, le rôle d'une telle stratégie d'évaluation dans la description des objets est souvent confus : a-t-elle pour rôle de suggérer les valeurs possibles d'un attribut ou d'imposer la valeur que doit nécessairement prendre un attribut ? La nature de la connaissance apportée par ces facettes d'évaluation d'attribut au sein de la description des classes n'est effectivement pas claire : connaissances typiques, hypothétiques ou sûres ?

C'est généralement dans le contexte de leur exploitation effective que cette question connaîtra une réponse : certains attributs ne sont jamais évalués par l'utilisateur et dépendent donc de la stratégie d'évaluation décrite, d'autres sont évalués exceptionnellement par l'utilisateur lorsqu'ils sortent du cadre d'un traitement typique, et d'autres encore ne font appel aux mécanismes d'évaluation qu'exceptionnellement pour remédier à l'absence de valeur fournie par l'utilisateur.

Dans le premier cas, la stratégie d'évaluation d'un attribut vient compléter la description des objets en contraignant l'attribut à prendre la valeur que cette stratégie est en mesure de délivrer. Dans le second cas, la stratégie d'évaluation d'un attribut n'est requise que lorsque l'utilisateur signifie le caractère typique de cet attribut en refusant de donner lui-même une valeur. Enfin, dans le troisième cas, la stratégie d'évaluation d'un attribut constitue pour l'utilisateur un support de production d'hypothèses, en l'occurrence une hypothèse est la proposition d'utilisation d'une méthode d'évaluation ou d'une valeur.

Le dernier cas est similaire au deuxième, seule l'attitude de l'utilisateur les distingue. Dans un cas, il refuse volontairement de donner une valeur pour signifier la typicalité de la situation, la valeur que fournira le système sera donc celle choisie indirectement par l'utilisateur et aura un caractère sûr. Dans l'autre cas, il ne sait pas évaluer l'attribut et le système lui propose une hypothèse.

Dans le cadre du problème d'identification/construction d'instances étudié, seules les situations où les inférences de valeur ont un caractère sûr seront considérées. Ce choix permet de considérer que les connaissances exprimées à travers les mécanismes d'inférence d'attribut participent aussi à la définition cohérente des objets.

Dans ce cas, l'association à un attribut d'un mécanisme d'évaluation d'attribut joue le même rôle qu'une contrainte impliquant cet attribut. Lorsqu'il est déclenché, l'obtention d'un résultat détermine la valeur que l'attribut doit nécessairement posséder. La difficulté de cette approche est d'exprimer les conditions dans lesquelles l'attribut peut être soumis à une telle contrainte ; c'est-à-dire les conditions qui définissent le déclenchement du mécanisme d'inférence.

Le chapitre VII propose un modèle permettant d'exprimer la stratégie d'évaluation d'un attribut à partir d'un ensemble disparate de mécanismes d'inférence. Cette stratégie indique en fonction de l'état de raffinement d'une instance (représentant ce qui est sûr, impossible et possible) et d'un ordre de priorité, quel mécanisme déclencher le cas échéant. Elle a donc pour rôle de définir précisément les conditions nécessaires et suffisantes d'application de tous les moyens d'évaluation associés à un attribut dans la hiérarchie.

L'ensemble des mécanismes considérés constitue une palette représentative des différents modes d'expression adoptés pour exprimer l'évaluation d'un attribut dans la représentation des connaissances par objets. Le contrôle d'évaluation d'un attribut permettant de gérer une stratégie exprimée à partir de tous ces types, il revêt ainsi un caractère général.

Les différents mécanismes sont le détachement procédural qui permet d'exprimer l'évaluation d'un attribut indépendamment de la hiérarchie de classes (cf. §VI.3 et §VII), les inférences par réduction de domaine provenant des contraintes inter-attributs introduites dans les classes (cf. §VI.2 et §VII), la facette de défaut généralement introduite dans une classe pour définir la valeur qu'un attribut doit avoir dans certaines sous-classes.

L'utilisateur est lui-même considéré comme un mécanisme d'évaluation d'attributs et doit se plier au comportement que lui assigne la stratégie d'évaluation de chaque attribut. L'intervention de l'utilisateur est ramenée à la pose de contraintes sur les attributs de l'instance. La prise en compte de l'utilisateur est donc ramenée par le contrôle à celui des évaluations d'attributs par contraintes. Le cas des partages de valeurs d'attributs entre objet composite et objets composants est considéré.

3.2. Aspect raisonnement

L'objectif de la partie précédente étaient de définir un modèle à objets permettant de décrire une hiérarchie de classes dans lesquelles les connaissances ne sont pas décrites pour résoudre un type particulier d'identification/construction d'instances. A cet effet, les notions de connaissances d'identification et de construction sont unifiées par la notion de classe qui est interprétée comme un ensemble de contraintes. Cette perception de la classe permet de définir l'appariement comme une vérification de contraintes et le rattachement comme l'application des contraintes.

L'objectif de la présente partie consiste à définir une nouvelle approche du problème d'identification/construction d'instances. La compréhension de cette problématique permet de définir le principe général de résolution.

3.2.1. Classes potentielles concrètes induites par un problème

La nouvelle définition du problème d'identification/construction d'instances (cf. §V.3.1.1) permet de redéfinir la notion de classe concrète :

Une classe est **concrète** pour un problème d'identification/construction d'instance si le rattachement de l'instance à cette classe est cohérent et permet d'atteindre tous les buts définissant ce problème ; c'est-à-dire d'évaluer les attributs requis.

Une classe possible pour une instance sur laquelle des buts ont été définis par l'utilisateur est une **classe concrète potentielle** si sa description recouvre l'ensemble des attributs désignés comme buts (cf. figure V.7).

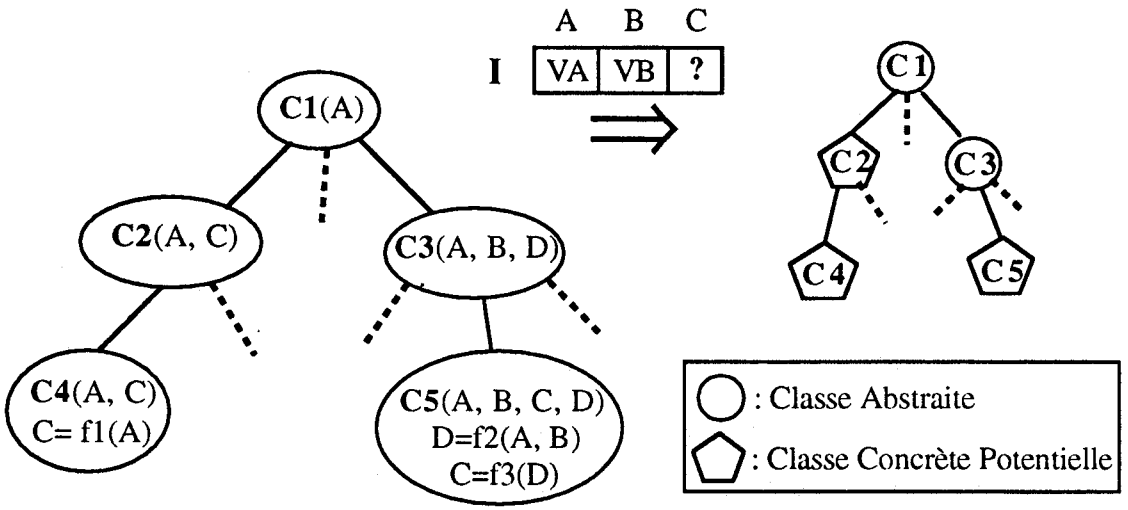

Figure V.7 : Le statut des classes est donné en fonction des attributs désignés comme buts. Ici, l'attribut C est désigné comme but. Les classes C2, C4 et C5 sont des classes concrètes potentielles pour I. Les classes C1 et C3 sont abstraites. Si I appartient à C4 ou C5, elle pourra être complétée grâce aux moyens d'obtention d'attribut f1 pour C, ou f2 et f3 pour D et C respectivement.

Plusieurs remarques peuvent être faites en fonction de la figure précédente. Si dans C5, le calcul de D n'était pas fourni, cette classe n'aurait pas la possibilité de devenir concrète pour l'instance I. Une classe concrète peut donc introduire de nouveaux attributs dans l'instance. Ces nouveaux attributs peuvent être eux aussi sans valeur ; c'est le cas, d'un passage éventuel de l'instance I par la classe C3, l'attribut D doit être pris en compte dans I. Cet attribut n'étant pas impliqué à ce niveau dans le calcul de C, l'introduire comme nouveau but est une décision qui appartient à l'utilisateur. En revanche, dans C5, D est un but car pour atteindre le but C, il faut atteindre le sous-but D.

Une classe concrète n'est pas obligée de couvrir tous les attributs donnés dans l'instance. Par exemple, C4 ne spécifie pas l'attributs B, et la fonction f1 n'a besoin que de A comme argument. Enfin, si l'attribut A n'avait pas de valeur pour l'instance I, il n'y aurait pas de classe concrète pour I.

Dans l'exemple, parmi les classes concrètes potentielles de I, les deux classes C4 et C5 sont de sérieuses candidates à être concrètes puisqu'elles offrent des moyens de calcul des attributs manquants.

La résolution d'un problème d'identification/construction consiste donc à explorer la hiérarchie de classes afin de trouver les classes concrètes pour le problème posé. La partie suivante interprète ce principe de recherche en terme d'exploration de la hiérarchie de classes.

3.2.2. Recherche des classes concrètes par exploration hypothétique

L'utilisation de la classification pour mettre en place la recherche des classes concrètes d'un problème particulier d'identification/construction d'instance se heurte au problème de la confrontation d'une instance incomplète aux classes de la hiérarchie. Lors de la définition d'un problème d'identification/construction d'instance, deux situations peuvent apparaître :

- les informations portées par l'instance sont suffisantes pour atteindre une classe concrète de ce problème d'identification/construction. Dans ce cas, les classes

d'appartenance connues permettent l'évaluation des attributs désignés comme buts soit directement car l'une d'elles est concrète, soit indirectement par exploitation des propriétés particulières de ces classes qui permettent de déduire de nouvelles classes d'appartenance de l'instance et d'accéder ainsi à une classe concrète.

- les informations portées par l'instance ne sont pas suffisantes pour atteindre une classe concrète. L'état incomplet de l'instance ne permet pas de la classer plus bas dans la hiérarchie. S'il existe une ou plusieurs classes concrètes pour ce problème, elles font partie des classes possibles de l'instance.

La première situation correspond au schéma classique de la déduction : l'utilisateur fournit des connaissances qui, confrontées à celle de la hiérarchie de classes, s'avèrent suffisantes. Ce type de problème rentre dans le cadre d'un raisonnement classificatoire normal. Toutefois, il faut noter dans ce cas que la classe concrète peut être atteinte parce que soit l'utilisateur l'a désignée comme classe d'appartenance (le problème est alors uniquement un problème de construction), soit une propriété particulière comme l'exhaustivité de classes peut être exploitée par la classification.

La seconde situation traduit un cas de connaissances incomplètes : ni l'utilisateur, ni la description des objets ne disposent des connaissances suffisantes pour résoudre le problème posé. C'est cette situation qui nous intéresse par la suite.

Prenons le cas simple de la hiérarchie des équations du second degré, si la propriété d'exhaustivité de la sur-classe **Equation-degré2** est omise dans la hiérarchie, la pose du problème d'identification/construction de l'instance **I-(2x²+5x+3=0)** ayant pour but l'évaluation de l'attribut *S* conduira à une situation de connaissances incomplètes. L'état de classification et l'état incomplet de l'instance traduiront alors cette situation (cf. Figure V.8).

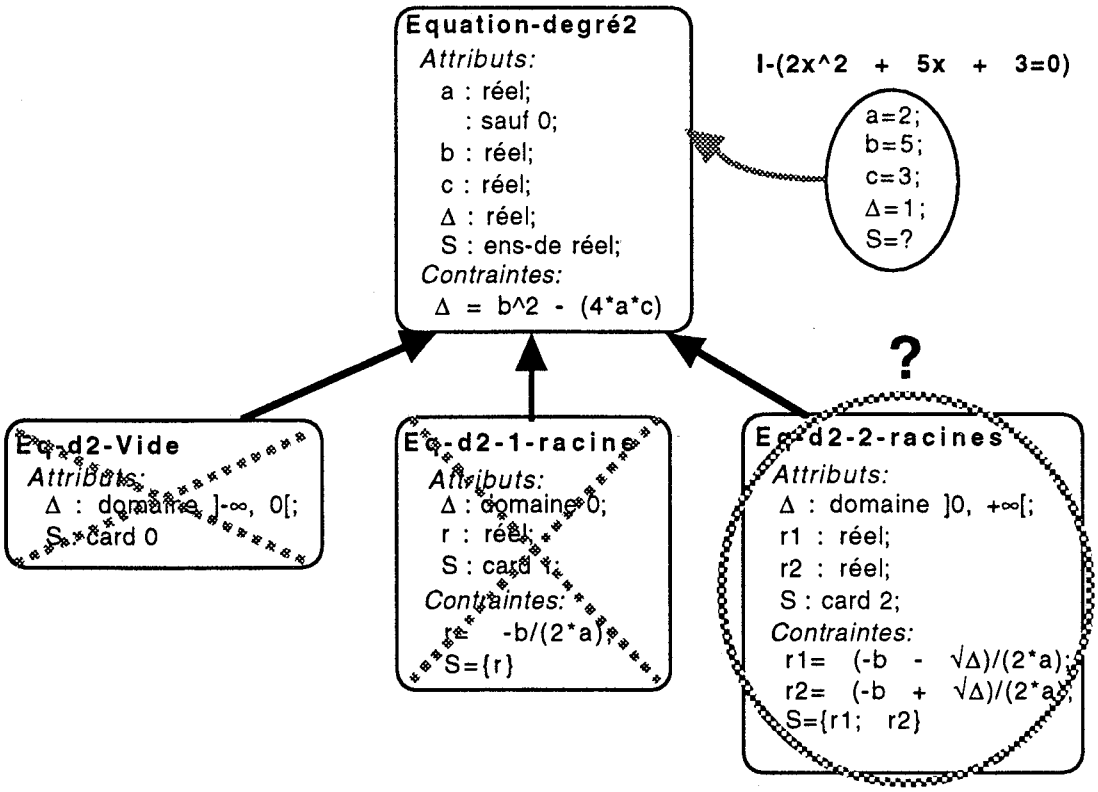

Figure V.8 : Dans cet exemple, l'exhaustivité de la classe **Equation-degré2** a été omise. Dans ce cas, la classification de l'instance **I-(2x²+5x+3=0)** ne permet pas de descendre l'instance car celle-ci est incomplète vis-à-vis des sous-classes. La détermination par la contrainte de Δ permet quand même de déclarer deux sous-classes impossibles. La dernière est possible, elle correspond à la classe solution mais l'instance ne peut y être attachée par la classification.

Dans ce cas, le système ne peut délivrer à l'aide du schéma classificatoire la solution car les connaissances décrites par la hiérarchie sont incomplètes. Toutefois, puisque l'état de classification indique que la classe **Eq-d2-2-racines** reste possible, il permet de souligner qu'une solution est *peut-être* encore possible. La distance qui sépare le problème de la solution est alors d'une hypothèse : la classe **Eq-d2-2-racines** est sûre pour l'instance.

Si une telle hypothèse est posée, le système sera en mesure de donner une solution au problème d'identification/construction d'instance d'équation. En effet, le rattachement à la classe **Eq-d2-2-racines** permettra le calcul, jusqu'alors inaccessible, des deux racines *r1* et *r2*, puis de l'ensemble de solution *S*.

Il faut toutefois noter une nuance fondamentale entre le résultat fourni par un tel principe et celui qui serait fourni par l'exploitation de l'exhaustivité de la classe **Equation-degré2**. L'exhaustivité permet de délivrer **la** solution (la seule), tandis que le recours à l'hypothèse d'appartenance permet de délivrer **une** solution (d'autres existent peut-être mais ne sont pas décrites par la hiérarchie).

Dans l'exemple des équations, il est préférable de déclarer l'exhaustivité puisque c'est une propriété fondée des équations du second degré. Par contre, dans la modélisation de domaines plus complexes, comme celui des neuropathies de MYOSIS, l'exhaustivité des classes ne peut pas toujours être affirmée avec certitude, ni même être suffisante pour déterminer exactement la maladie au vu des informations données initialement. En effet, certaines maladies ne sont pas prises en compte dans la hiérarchie (car peut-être inconnues ou insuffisamment connues), et d'autres peuvent être possibles à partir du même type de signes cliniques, seul le résultat de tests propres aux maladies les différencieront. Lors de l'identification/construction d'une instance de maladie plusieurs classes possibles peuvent apparaître. Chacune d'elles représentent alors une hypothèse de maladie qu'il faut vérifier. Dans ce cas, plusieurs solutions peuvent être construites : le patient est atteint de plusieurs maladies.

La nouvelle approche de résolution du problème d'identification/construction d'instances qui est proposée consiste donc à permettre d'explorer la hiérarchie de classes sur un mode hypothétique. La partie de la hiérarchie explorée sur ce principe est délimitée par l'état de classification de l'instance : l'ensemble des classes possibles de la hiérarchie.

Pour garder la généralité du modèle de classification il est important de distinguer les connaissances hypothétiques des connaissances sûres. Le principe adopté par la suite consiste à gérer la notion d'instance hypothétique. Si un problème d'identification/construction d'instances admet plusieurs solutions, ces dernières sont représentées par des instances hypothétiques différentes. Chacune d'elles se caractérise par les hypothèses dont elle a fait l'objet.

Le mécanisme de résolution de problème d'identification/construction d'instances relève donc d'un raisonnement de type abductif. En effet, il explore la hiérarchie de classes afin de produire et valider différentes hypothèses de rattachement. La validation de ces hypothèses consiste à déterminer si elles permettent à l'instance d'atteindre les buts posés par l'utilisateur.

4. Conclusion

Le premier objectif recherché par ce chapitre était de montrer que la prise en compte d'un type particulier de problèmes détermine et influe sur la façon de construire une hiérarchie de classes. A ce titre, l'étude des modèles proposant la classe comme support de construction ou de gestion d'instances, comme les langages de programmation par objets, est formatrice.

Dans un tel modèle, une hiérarchie de classes est directement conçue pour répondre aux besoins d'une application particulière. Ces besoins se traduisent au niveau de la hiérarchie par la notion de *classe concrète*. Une classe est concrète parce qu'elle décrit un ensemble d'objets dont l'état et le comportement représentent les informations nécessaires et suffisantes aux traitements d'une étape particulière de l'application. Les autres classes, appelées *classes abstraites*, n'ont pas pour vocation de décrire un ensemble d'objets mais permettent d'organiser en hiérarchie l'ensemble des classes concrètes afin d'en factoriser les descriptions.

Dans un modèle dans lequel la classe sert aussi bien à l'identification qu'à la construction d'une instance, une démarche similaire de conception de hiérarchies de classes peut être

observée. Bien que la distinction entre classes abstraites et classes concrètes ne soit pas explicite, une hiérarchie est cependant construite de telle façon que le processus de classification puisse atteindre certaines classes particulières. Ces classes dénotent les situations dans laquelle une instance peut être effectivement construite. Elles délimitent donc l'espace de recherche du processus d'identification/construction et jouent le rôle de classes concrètes pour ce problème.

Les autres classes de la hiérarchie, reléguées à un rôle de classes abstraites, sont décrites pour représenter un choix exhaustif de classes concrètes. C'est donc sur ces classes que le raisonnement classificatoire s'appuie pour effectuer la recherche des classes concrètes pouvant correspondre à un problème particulier.

Pour que la classification d'instance permette la résolution d'un problème d'identification/construction, il faut donc concevoir la hiérarchie de classes de façon ascendante : définir les classes concrètes (pour ce type de problème), puis définir les classes abstraites de telle façon que le cheminement de la classification jusqu'à une classe concrète soit garanti. Cette méthode de conception d'une hiérarchie suppose que la hiérarchie de classes est conçue pour résoudre un type particulier de problème d'identification/construction.

Cette approche du problème d'identification/construction d'instances est réductrice puisqu'elle contraint la description des objets à répondre aux besoins d'un raisonnement classificatoire particulier ; celui correspondant au processus de résolution de problème. En effet, cette approche favorise une description des objets orientée par la notion de problème et, par là-même, ne fournit qu'une interprétation incomplète des propriétés générales que possèdent ces objets. La spécificité d'une telle description d'objets restreint l'exploitation qui peut en être faite.

Le second objectif de ce chapitre était de proposer une nouvelle approche du problème d'identification/construction d'instances visant à respecter le principe d'indépendance entre la représentation et l'exploitation des connaissances. L'intérêt de ce principe est de favoriser l'exploitation multiple des connaissances décrites.

Cette nouvelle approche s'oppose donc à une solution prônant une description des objets orientée par la notion de problème. Pour ce faire, trois thèmes sont abordées :

- la révision de la notion de problème d'identification/construction d'instances ;
- la représentation de connaissances pour l'identification et pour la construction d'objets au sein d'un même modèle ;
- le type de raisonnement permettant la résolution d'un tel problème.

En ce qui concerne le modèle à objets, nous proposons d'interpréter la notion de classe comme un ensemble de contraintes. Cette approche de la description des objets unifie les deux rôles d'une classe, unité d'identification ou de construction, sans préjuger de la façon dont elle sera exploitée. En effet, l'ensemble de contraintes représenté par une classe permet à la fois de vérifier l'adéquation d'un ensemble de données par rapport à cette classe et, parfois, de compléter cet ensemble de données lorsqu'il s'avère incomplet.

Un problème d'identification/construction d'instances n'est pas qu'un problème de classification. En effet, un tel problème se caractérise par la donnée d'une instance pour laquelle certaines informations sont requises par l'utilisateur. Dans ce contexte, le problème posé est de savoir quelles classes offrent à cette instance à la fois un contexte de rattachement cohérent et un contexte de construction suffisant pour répondre à la requête initiale.

La résolution d'un problème d'identification/construction d'instances repose sur un raisonnement capable de remédier aux insuffisances du raisonnement classificatoire. Le principe de ce raisonnement consiste à étendre, grâce à la gestion d'hypothèses, l'exploration de la hiérarchie de classes au delà des limites atteintes par la classification de l'instance.

Cette solution permet, d'une part, de considérer à partir d'une même hiérarchie de classes divers types de problèmes d'identification/constructions d'instances et, d'autre part, de traiter le cas d'un problème à solutions multiples.

Par la suite, l'ensemble de ces propositions est mis en œuvre dans le modèle TROPES. La première étape de cette mise en œuvre consiste à étendre ce modèle en intégrant des mécanismes d'évaluation d'attributs. Cette intégration est complétée par la mise en place d'un contrôle global permettant d'intégrer l'évaluation des attributs au processus de raffinement d'une instance dans une hiérarchie de classes. La seconde étape consiste à étendre le fonctionnement du modèle TROPES vers la mise en place d'un raisonnement hypothétique. Dans ce contexte, cette extension se concrétise par la proposition d'un mécanisme général de configuration hypothétique d'instance, à partir duquel peut être exprimé et résolu un problème d'identification/construction d'instance.

Partie 3

IDENTIFICATION/CONSTRUCTION D'INSTANCES DANS TROPES

Chapitre VI
EXTENSIONS DE TROPES

1. Introduction

La première version du modèle TROPES (cf. §II.4) pose les bases nécessaires à un raisonnement classificatoire. Cependant, pour envisager un raisonnement qui allie à la fois l'identification et la construction d'une instance, le modèle doit, de plus, permettre les inférences de valeurs d'attributs et la possibilité d'introduire la notion d'hypothèse.

Cette partie présente l'intégration au modèle de deux types distincts d'inférences :

- **Les inférences locales à une classe** : ces inférences sont fournies par la possibilité d'exprimer des contraintes au sein de la description d'une classe. Ce type d'inférence est donc local dans le sens où son déclenchement dépend du rattachement de l'instance à des classes. Un module de gestion de contraintes se charge de la maintenance et des inférences de valeurs d'attributs qui découlent de la pose de contraintes sur des objets de la base.
- **Les inférences globales à un concept** : ces inférences sont le résultat d'un mécanisme d'évaluation global, nommé "détachement" procédural, qui fournit dans la déclaration d'un attribut, au niveau du concept, des connaissances opératoires permettant le calcul de sa valeur. L'utilisation de ce mécanisme est indépendante de tout rattachement de l'instance à une classe, c'est un moyen global d'obtention de valeurs d'attributs qui ne dépend que de l'état des attributs de l'instance concernée.

L'intégration des contraintes présentée ici (cf. §VI.2) est celle proposée par Jérôme Gensel dans [Gens93] et [Gens95]. La spécification du détachement procédural, dont le principe a été décrit dans [Orsi90] et [Rech92], reste à préciser. Pour ce faire, ses modalités de déclenchement, son interfaçage avec les objets et la description de ses états d'exécution sont établies (cf. §VI.3).

Dans ce chapitre, chacune de ces extensions est présentée indépendamment l'une de l'autre. Les problèmes de contrôle, liés à la coexistence de plusieurs mécanismes d'évaluation d'attributs au sein du même modèle et à leur exploitation lors du raffinement d'une instance dans une hiérarchie de classes, sont traités dans le chapitre suivant (cf.§VII).

2. Rappel du principe d'intégration des contraintes

Afin d'augmenter le pouvoir d'expression et d'inférence du système TROPES, [Gens93] propose d'y introduire la notion de contrainte pour exprimer les liens qui peuvent exister entre des attributs d'un même objet (liens internes) ou d'objets distincts (liens externes). L'intérêt des contraintes dans la représentation des objets est double.

D'une part, les contraintes permettent d'exprimer au sein de la description d'une classe des conditions complexes auxquelles doit se plier toute instance lui appartenant. Ces conditions dépassent le cadre du typage des attributs des objets puisqu'elles doivent être vérifiées dynamiquement au niveau de l'instance dès que l'un de ses attributs figurant dans une contrainte reçoit une nouvelle valeur.

D'autre part, les contraintes s'avèrent être un nouveau moyen d'inférence de valeurs d'attributs. En effet, dans le cas d'une instance, la donnée des valeurs d'attributs impliqués dans un réseau de contraintes permet d'appliquer de nouvelles restrictions sur les domaines des attributs contraints qui restent encore sans valeur. Aussi, lorsqu'un tel domaine est réduit à une seule valeur, l'attribut concerné se voit affecter cette valeur.

Pour ce faire, le système de raisonnement, ou moteur d'inférence, est augmenté d'un module de gestion de contraintes (cf. Figure VI.1).

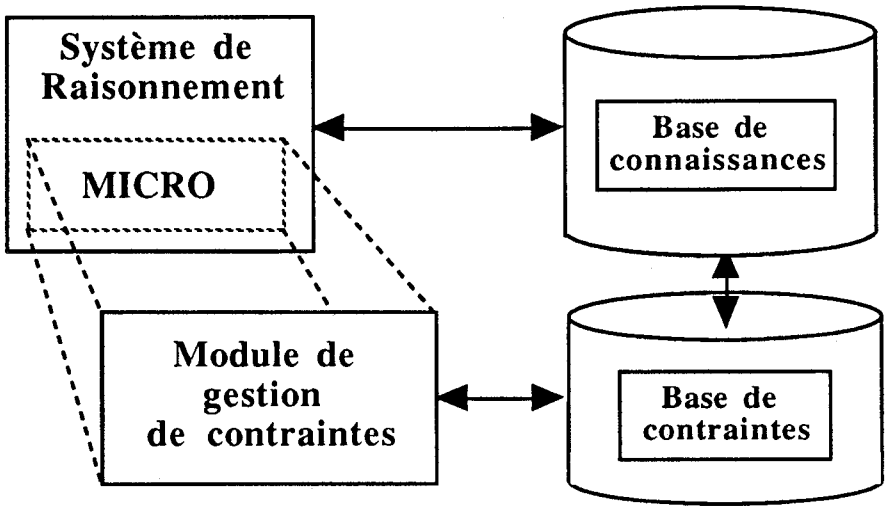

Figure VI.1 : Intégration du module de gestion de contraintes MICRO dans le système de représentation par objets TROPES.

Le module MICRO, acronyme pour Module pour l'Intégration de Contraintes et de Relations aux Objets, gère une **base de contraintes**. Cette base contient l'ensemble des contraintes activées par le système TROPES. En effet, des réseaux de contraintes gérés par MICRO sont construits et introduits dans la base de contraintes à chaque fois que de nouvelles contraintes sont posées sur une instance de la base de connaissances TROPES.

Dans la base de MICRO, une variable, appelée **variable contrainte**, est associée à chaque attribut d'une instance qui est impliqué dans une contrainte. Cette variable rend compte du domaine de valeurs permises pour l'attribut. Ce domaine est calculé par MICRO en fonction des contraintes dans lesquelles figure l'attribut.

L'interface entre objets et contraintes est assurée par l'introduction de primitives de déclaration de contraintes au niveau de la définition des objets, et par la gestion des liens qui assurent les échanges d'informations entre les attributs de TROPES et les variables de MICRO (cf. §VI.2.1). Le module MICRO dispose d'un ensemble de primitives permettant de construire différents types de contrainte (cf. §VI.2.2), d'un ensemble de règles spécifiques à chaque type de contraintes permettant d'appliquer les réduction de domaine des variables participant à un réseau, et d'un mécanisme de propagation permettant de répercuter les effets d'une réduction de domaine d'une variable à toutes les variables d'un même réseau de contraintes.

2.1. Interface entre TROPES et MICRO

2.1.1. Déclaration de contraintes dans une classe

De façon analogue au système THINGLAB (cf. §II.3.4.2), une description de classe TROPES est enrichie d'un espace réservé pour la déclaration de contraintes (cf. Figure VI.2). Afin de respecter la sémantique de la relation de spécialisation, les contraintes d'une classe sont héritées par toutes ses sous-classes.

Contrairement à THINGLAB qui propose l'expression de *contraintes fonctionnelles*, la déclaration de contraintes inter-attributs dans une classe de TROPES correspond à l'expression de problèmes de satisfaction de contraintes (CSP). Le service de base que propose le module MICRO au système TROPES est de maintenir la consistance (locale) des réseaux de contraintes. Les techniques de satisfaction et de maintien de contraintes mises en place dans MICRO sont largement détaillées dans [Gens95]. Seule l'exploitation de MICRO par TROPES sera abordée par la suite.

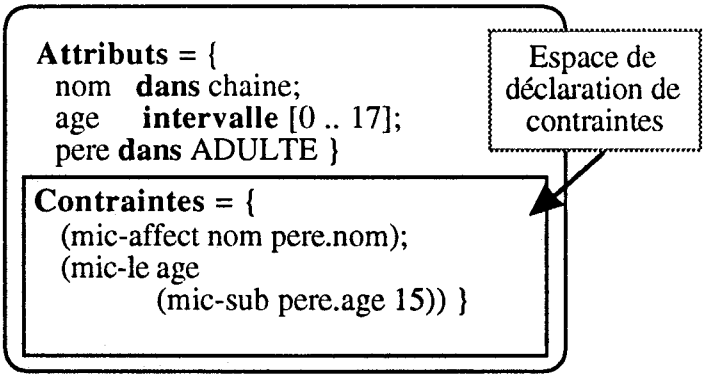

Figure VI.2 : Exemple de déclaration de contraintes dans la classe ENFANT. La première contrainte, (*mic-affect nom pere.nom*), exprime que le nom d'un enfant doit être le même que celui de son père. Dans cette contrainte d'affectation, l'accès à l'attribut *nom* du père est donné par le chemin *pere.nom*. La deuxième contrainte exprime que l'âge d'un enfant doit être inférieur ou égal à l'âge du père moins quinze.

Dans une description de classe, chaque contrainte est exprimée sous la forme fonctionnelle suivante :

- (<*nom-contrainte*> <arg_1> ... <arg_n>);

où *nom-contrainte* est le nom de la contrainte MICRO, et chaque arg_i peut être :

- une constante.
- un **chemin** d'accès à un attribut d'une instance. La donnée d'un chemin est nécessaire pour atteindre un attribut particulier à partir d'un attribut complexe, c'est-à-dire un attribut qui prend ses valeurs dans un ensemble d'objets. Un chemin est donc défini par la liste des attributs par lesquels il faut nécessairement et successivement passer pour atteindre l'attribut impliqué dans la contrainte.
- une contrainte.

Une fois qu'une description de classe est donnée, elle est interprétée de la façon suivante :

1) phase de typage des attributs [Capp94] de la classe. Le domaine de définition de chaque attribut est calculé à partir de son sur-type (provenant de la sur-classe ou du concept) et de la description de l'attribut dans la classe courante. Ce nouveau domaine est associé à l'attribut dans la classe.
2) Constitution dans MICRO des réseaux de contraintes exprimés à travers la déclaration de la classe. Les domaines de la phase de typage sont associées aux variables contraintes correspondantes.
3) réduction des domaines par application des règles de réduction de chaque contrainte et par la propagation des réductions aux autres variables du réseau.
4) donnée des domaines réduits par MICRO au module de types. Ce dernier met alors à jour les types de chaque attribut de la classe avec ces nouveaux domaines.

Ce couplage entre le module de contraintes et le module de types permet d'effectuer une première phase de filtrage afin de ne retenir que les valeurs admissibles de chaque domaine d'attribut.

Pour le module de contraintes, ce premier calcul de domaines évite un recalcul systématique à chaque rattachement d'une instance à une classe. En effet, commun à toute

utilisation ultérieure des contraintes de la classe, le calcul est ainsi effectué une seule fois. Les domaines d'attributs résultants sont alors enregistrés au niveau des types d'attributs.

En ce qui concerne le module de types, la prise en compte partielle des contraintes au niveau du calcul des domaines d'attributs est importante pour améliorer la vérification des types, et par extension des sous-types, des attributs contraints dans des classes. En effet, cette procédure permet de signaler les cas de contraintes trop restrictives, c'est-à-dire celles dont l'application réduit un domaine d'attribut à l'ensemble vide. Toutefois, les types ne pouvant pas prendre en compte l'aspect dynamique des contraintes, l'appariement d'une instance à une classe comportant des contraintes fait nécessairement appel au module MICRO pour la vérification des contraintes.

2.1.2. Etablissement des liens entre variables et attributs contraints

Lors du rattachement d'une instance à une classe comportant des contraintes, le module de contraintes MICRO construit les réseaux de contraintes correspondants. Les variables de ces réseaux qui correspondent à des attributs de l'instance (ou de plusieurs instances, lorsque l'objet est complexe) sont initialisées avec les domaines fournis par les types des attributs de la classe.

De plus, l'application d'un ensemble de contraintes sur une instance nécessite la mise en place de liens pour, d'une part, assurer la prise en compte des informations associées à l'instance dans le réseau de contraintes et, d'autre part, propager au niveau de l'instance toute information inférée par les contraintes (cf. Figure VI.3).

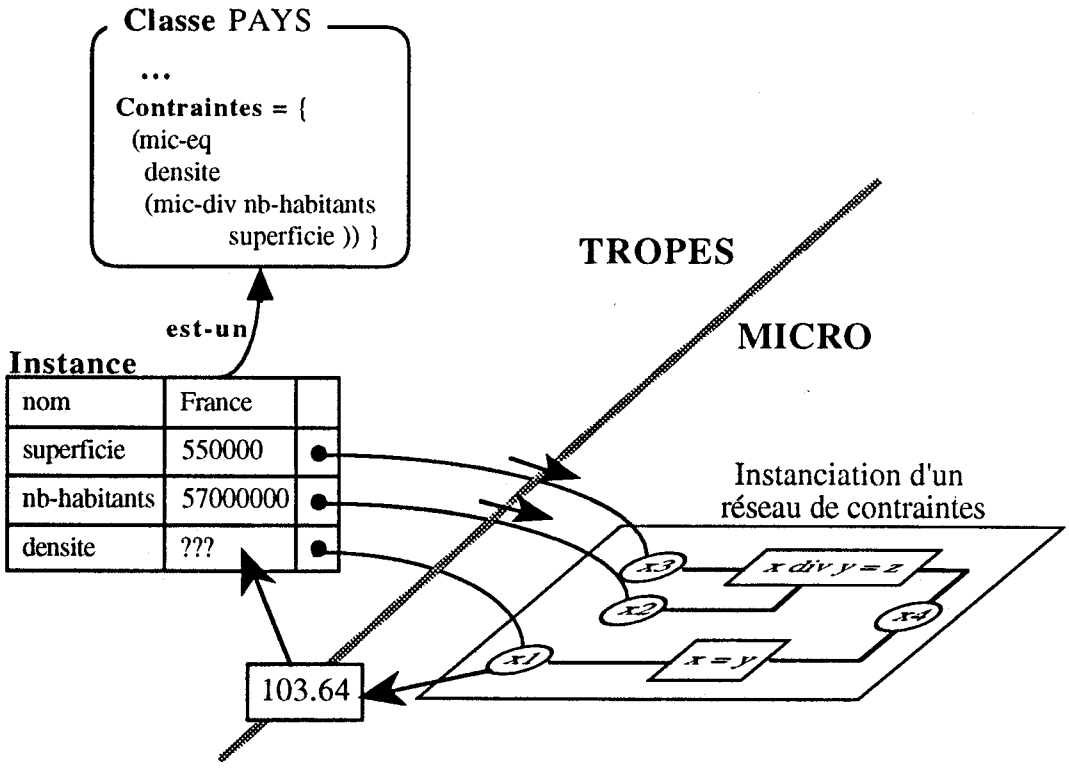

Figure VI.3 : Exemple de liaisons entre une instance de TROPES et un réseau de contraintes géré par MICRO. Chaque attribut contraint de l'instance est lié à la variable MICRO qui lui correspond dans le réseau de contraintes : l'attribut *superficie* est lié à la variable contrainte *x3*, etc. Dans cet exemple, l'attribut *densite* est déterminé grâce à la contrainte (*densite* = *nb-habitants*/*superficie*).

A travers les liaisons entre une instance de TROPES et des réseaux de contraintes de MICRO, le mécanisme de maintenance proposé permet :

- de statuer sur la consistance d'un ensemble de contraintes appliqué à une instance au vu de ses valeurs d'attributs déjà connues. Si aucun des domaines de variables contraintes n'est réduit à l'ensemble vide, MICRO informe TROPES que les contraintes posées sur l'instance sont respectées.
- d'inférer des valeurs d'attributs lors de la réduction de domaines de valeurs à des singletons.
- de prendre systématiquement en compte les modifications de l'instance. Dans ce cas, le système de maintenance de raisonnement permet de défaire toutes les inférences de valeurs dépendantes de cette modification et les réseaux de contraintes concernées sont réinitialisés en conséquences.

La partie suivante donne un aperçu rapide des types de contraintes proposés ainsi que du principe de la maintenance d'un réseau de contraintes.

2.2. Module MICRO

Bien que l'intégration des contraintes aient été proposée initialement selon un couplage entre le système TROPES et la bibliothèque de programmation par contraintes PECOS [ILOG92], il s'est avéré par la suite que cette bibliothèque était peu adaptée aux expressions complexes de domaine d'attributs présents dans le modèle TROPES [Gens95]. Motivé, de plus, par les possibilités offertes par le module de gestion des types, le module MICRO a été conçu par Jérôme Gensel pour remplacer la bibliothèque PECOS.

Adapté aux possibilités d'expressions d'attributs, le module MICRO offre :

- un ensemble de contraintes de comparaison : $(x \mathcal{R} y)$, où x et y sont des variables contraintes de même type, et $\mathcal{R}$ une des relations $=, <, \leq, >, \geq, \neq$ ou $\leftarrow$. Cette dernière relation, $\leftarrow$, représente une contrainte d'affectation qui, contrairement aux précédentes, n'a pas de caractère bi-directionnel.
- un ensemble d'expressions arithmétiques (sur les entiers ou les réels) applicables sur des variables contraintes à valeurs monovaluées :
  - sous la forme binaire $f(x, y) : x + y, x - y, x \times y, x \div y, x^y$. Ces expressions permettent l'écriture des contraintes arithmétiques ternaires de la forme $(z \mathcal{R} f(x, y))$, où $\mathcal{R}$ est une contrainte de comparaison.
  - sous la forme unaire $g(x) : -x, |x|, e^x$, log x, sin x, cos x permettant l'expression de contraintes binaires $(y \mathcal{R} g(x))$, où $\mathcal{R}$ est une contrainte de comparaison.
- un ensemble d'expressions sur les variables à caractère multivalué, permettant d'exprimer des contraintes sur :
  - un élément particulier d'une liste.
  - tout un ensemble ou une liste de variables contraintes.
  - l'intersection, l'union, la différence de deux ensembles.
  - la concaténation de deux listes.
  - la nécessité pour une valeur de variable monovaluée d'être présente dans un ensemble ou une liste.
  - l'ensemble ou la liste résultant de l'application d'une fonction ou d'une contrainte à chaque élément d'un ensemble ou d'une liste.
  - la cardinalité d'un ensemble ou la longueur d'une liste.
  - etc.
- un ensemble de contraintes conditionnelles et d'évolution qui permettent respectivement de poser des contraintes sous certaines conditions, et de vérifier que

l'évolution d'une variable respecte bien certaines conditions. Ces deux types de contraintes permettent de contrôler la pose des contraintes.

Cette bibliothèque permet d'exprimer des contraintes complexes comme le montre l'expression suivante :

```
(mic-eq    prix-total
           (mic-apply    mic-add
                         (mic-map-list get-slot prix meubles)))
```

signifiant que le prix total (d'un bureau, par exemple) doit être égal à la somme des prix de tous les meubles (du bureau) compris dans la liste. La modification d'un prix, l'ajout ou le retrait d'un meuble dans la liste seront automatiquement pris en compte pour recalculer le nouveau prix total.

A chaque type de contrainte correspond un ensemble de règles de restriction des domaines des variables contraintes impliquées. Pour former un réseau, les contraintes sont reliées par le biais des variables qu'elles partagent et à travers lesquelles sont propagées les réductions de domaine d'une contrainte à une autre.

3. Principe d'intégration des évaluations globales d'attributs

Les moyens d'évaluation globaux, aussi appelés "détachements" procéduraux par la suite, ont fait l'objet d'une première étude [Orsi90] dont les principales propositions sont reprises et étendues dans le travail présenté par la suite. Le principe général consiste à sortir des classes les calculs de valeurs d'attributs qui y sont introduits par des attachements procéduraux (cf. §II.3.4.3). Bénéficiant du contexte de déclaration globale des attributs fourni par la notion de concept de TROPES, les connaissances de calcul d'un attribut peuvent être raisonnablement réunies au niveau du concept et donc utilisables par une instance indépendamment de tout rattachement à une classe.

Cependant, le regroupement de cette connaissance au niveau d'une déclaration globale d'attribut pose un nouveau problème : comment sélectionner les méthodes de calculs adaptées à la configuration particulière d'une instance ? Cette problématique est propre à l'approche puisqu'avec les attachements procéduraux ce choix était statiquement établi par la répartition des méthodes de calcul dans les classes de la hiérarchie d'objets : la méthode de calcul à exécuter pour évaluer l'attribut est alors celle disponible dans la classe de rattachement de l'instance.

Pour remédier à ce problème, la solution qui a été proposée est générale et dépasse le cadre de cette simple problématique. Elle consiste à lier dynamiquement les objets aux méthodes de calculs d'attributs par l'ajout d'un niveau stratégique exprimé sous forme de tâches.

Une **tâche** dans sa généralité représente la donnée d'une connaissance de résolution d'un problème particulier définie par le type des données d'entrée et des données de sortie, ainsi que par la stratégie à mettre en œuvre pour résoudre le problème. Cette stratégie consiste en une décomposition en choix ou séquence de sous-tâches. Le modèle de tâches adopté par B. Orsier s'inspire des idées développées dans [Rous88] sur l'intégration de la notion de tâches dans un système par objets, reprises aussi dans les systèmes SCAI [Ponc91][Ponc&91], SCARP [Will94]. Un tel modèle a été proposé pour le modèle TROPES dans [Gens&92] [Gens&94]. La représentation d'une tâche sous forme d'objets de la base de connaissances est un point commun à ces modèles.

Aussi, dans le contexte de l'évaluation d'un attribut d'une instance, l'utilisation de tâches est particulièrement intéressante. En effet, elle permet d'exprimer le choix d'une méthode de calcul d'un attribut en fonction des informations portées par l'instance demandeuse, mais aussi de composer entre elles des procédures afin de proposer de nouvelles méthodes de calcul. Cette intégration étend de façon déclarative les possibilités de calculs d'attributs habituellement représentées par les attachements procéduraux.

Le travail présenté dans la suite vient compléter cette première proposition d'intégration de calculs d'attributs par appel de tâche. En effet, l'esquisse initiale avait essentiellement porté sur

la mise en place d'un moteur de tâches, laissant imprécisé le processus de déclenchement du calcul d'un attribut. C'est pourquoi, cette partie spécifie le comportement attendu du "détachement" procédural par rapport aux besoins des instances : modalités de déclenchement, modalités de passage d'informations entre tâche et objet, et les états d'exécutions possibles.

Une des extensions majeures du comportement réside dans le maintien de l'état d'exécution d'une tâche de calcul d'un attribut. Ce maintien, analogue à celle d'une contrainte posée sur une instance, permet de gérer les cas où les informations contenues dans une instance sont incomplètes et insuffisantes pour mener à bien les calculs d'attributs entrepris. Au lieu d'être déclarée en échec et abandonnée, l'exécution de tels calculs est suspendue, sa reprise est assurée dès que l'instance est suffisamment complète. Cette extension a donc pour but général de rendre l'exécution du calcul d'attribut indépendant de son instant de déclenchement, et trouve pleinement sa justification lors de la mise en place de raisonnement hypothétique qui constitue l'extension suivante du modèle.

L'intégration des évaluations globales d'attributs nécessite la mise en place d'un module de gestion des déclenchements et de maintien de l'exécution des tâches de calcul correspondantes. Aussi, ce nouveau module vient prendre place dans le système de raisonnement parallèlement au module MICRO (cf. Figure VI.4).

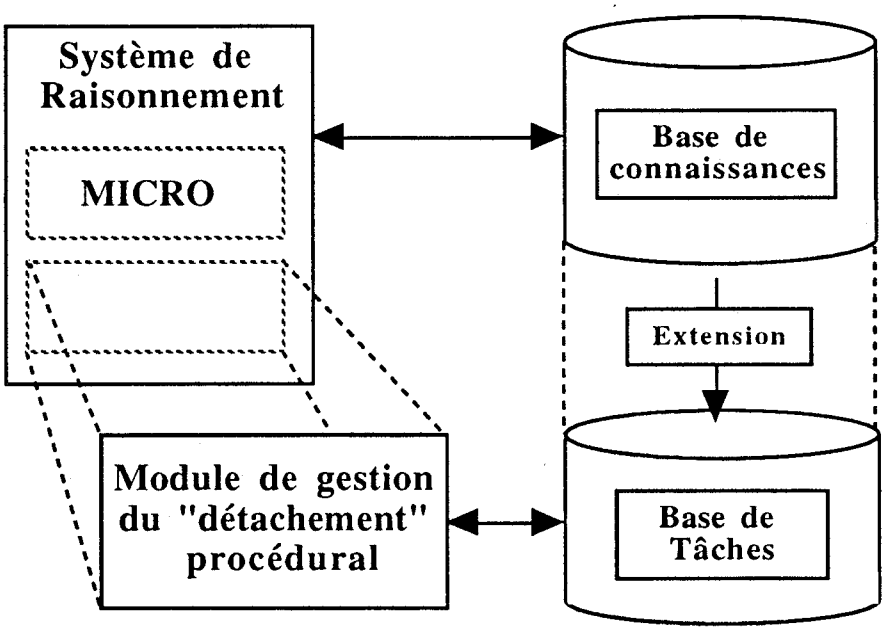

Figure VI.4 : Intégration des moyens d'évaluations d'attributs, "détachements" procéduraux, dans le modèle TROPES. Ce module est chargé de gérer les déclenchements d'exécution de tâches en fonction des besoins du système de raisonnement en valeurs d'attributs et d'assurer la mise en place des échanges de données entre instances et tâches.

Par la suite, la mise en place de l'interface entre une instance et l'exécution d'une tâche de calcul d'un de ses attributs est décrite au sein du modèle TROPES : la déclaration, le déclenchement, les différents états d'exécution, et le maintien des échanges de données.

Pour des raisons de lisibilité évidente, un moyen global d'évaluation d'attribut est appelé "détachement" procédural ou encore plus simplement DP.

3.1. Déclaration d'un détachement procédural

Un DP est décrit pour un attribut au niveau du concept, c'est une connaissance globale du concept. La donnée d'un détachement procédural d'un attribut s'effectue au niveau de la déclaration générale de l'attribut dans le concept (cf. Figure VI.5). Elle consiste en :

- La désignation d'un type de tâche particulier permettant l'évaluation de l'attribut considéré. Les tâches étant modélisées sous forme d'objets, cette information correspond à la donnée du nom d'une classe du concept prédéfini "Tâche" [Gens&94].
- La donnée d'une table de correspondances entre les attributs de l'instance et les attributs d'entrées/sorties (E/S) de la tâche du DP.

Ces informations associées à l'attribut permettent à un raisonnement nécessitant la valeur de l'attribut de fournir au module de gestion des DP les données nécessaires à la création de l'instance de tâche et les moyens de récupérer dans l'instance appelante les valeurs d'attributs que la tâche nécessite en entrée.

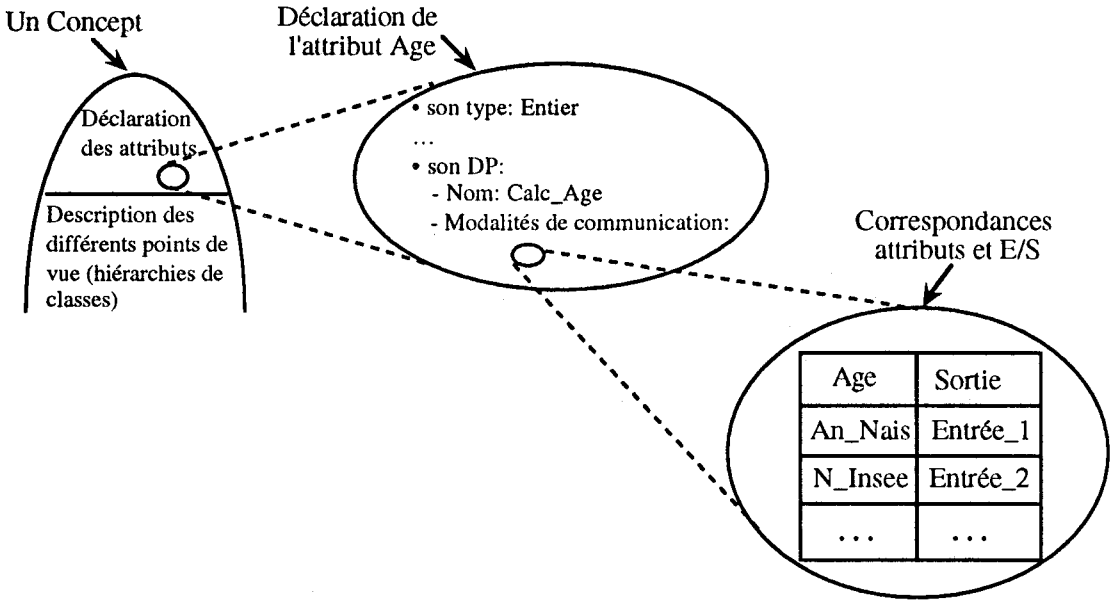

Figure VI.5 : Déclaration du DP de l'attribut Age au niveau de sa déclaration dans le concept. Cette déclaration comporte la désignation de la tâche de calcul Calc_Age, et les correspondances possibles entre les attributs d'une instance du concept et les E/S d'une tâche de type Calc_age.

La description de la correspondance entre les attributs de l'instance et la tâche est une donnée statique utilisée et partagée par toute instance du concept ayant déclenché ce DP. Elle permet à une tâche active de traduire la demande d'une valeur d'entrée en une demande de valeur d'attribut de l'instance.

3.2. Déclenchement d'un détachement procédural

Le DP, contrairement à la notion courante d'**attachement** procédural [Masi&89], est indépendant de la notion de classe. Toute instance appartenant à ce concept et nécessitant la valeur d'un attribut fait appel au DP correspondant.

La seule condition de déclenchement d'un DP par une instance provient de la mise en évidence d'un attribut à valeur inconnue. Aussi, tout mécanisme qui adresse une demande de valeur d'attribut à une instance est susceptible de déclencher l'évaluation de cet attribut par son DP. Ce peut être lors de la classification, d'un rattachement, de l'évaluation d'un autre attribut de l'instance, d'une interrogation de l'utilisateur sur le contenu d'une instance, etc.

Le déclenchement d'un DP correspond à la création d'une instance de tâche qui est initialement configurée avec l'attribut attendu en sortie, à valeur inconnue, et sa classe de rattachement indiquée dans la déclaration de l'attribut à calculer. Cette configuration déclenche l'exécution de la tâche. Sa stratégie se développe en s'adaptant aux informations portées par l'instance qui sont récupérées via la table de correspondances (cf. Figure VI.6).

La récupération des valeurs d'attributs ne se fait pas d'un bloc mais au contraire progressivement en fonction du développement de la stratégie vers la sélection d'une méthode de calcul précise et effective. Cette dernière n'a pas nécessairement besoin de toutes les entrées définies dans la table de correspondance. La détermination de l'ensemble des entrées nécessaire à une exécution de DP est dynamique, elle s'adapte en fonction de la configuration de l'instance. Ainsi, dans l'exemple suivant (cf. Figure VI.6), pour effectuer le calcul de l'attribut *Age*, si la stratégie explore la possibilité de le calculer en fonction de l'attribut *An_Nais*, elle n'a pas besoin de disposer aussi de l'attribut *N_Insee*. Dans ce cas, l'instance de tâche n'a besoin que de l'attribut *Entrée_1*.

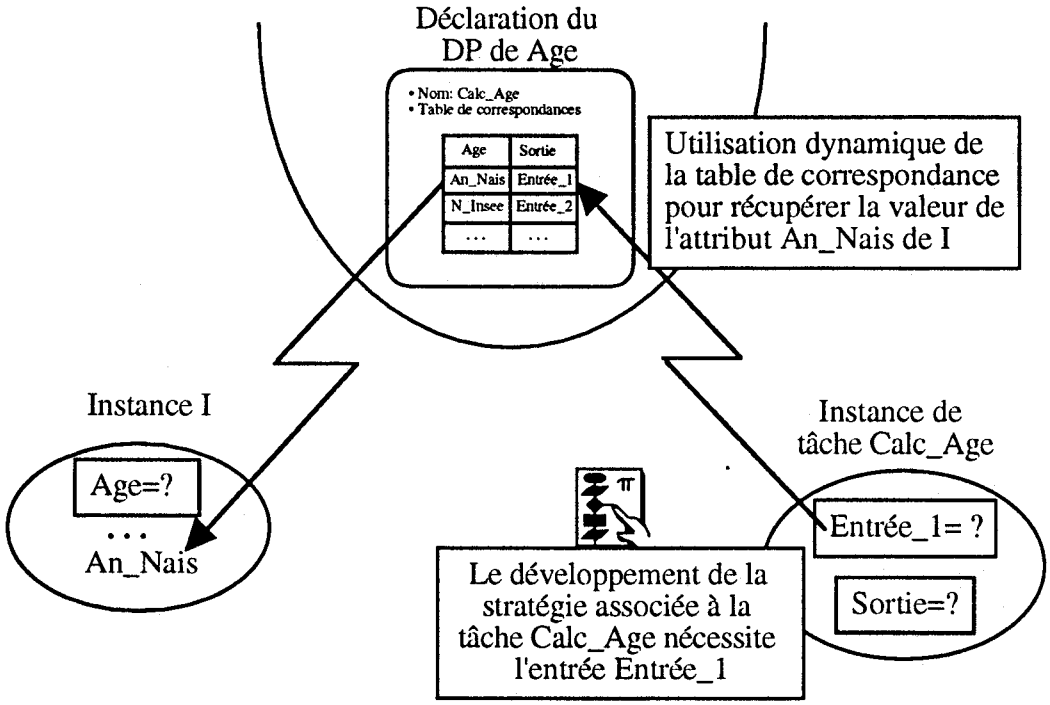

Figure VI.6 : Utilisation de la table de correspondance d'un DP. L'attribut *Sortie* a été initialement introduit lors de la configuration de l'instance de tâche *Calc_Age* car il est associé dans la table de correspondance à l'attribut *Age* à calculer dans l'instance *I*. L'attribut *Entrée_1* est, quant à lui, introduit lors du développement de la stratégie de la tâche, il déclenche le processus de récupération de la valeur de l'attribut *An_Nais* via la table de correspondances.

Enfin, comme toute demande d'attribut, celles formulées à une instance à travers la table de correspondance d'un DP sont susceptibles de déclencher les DP des attributs demandés.

3.3. Prise en compte de l'évolution d'une instance

La principale originalité du fonctionnement de DP proposé ici réside dans la gestion d'états d'attente durant l'exécution d'une tâche de calcul. Un tel état permet de prendre en compte l'aspect évolutif d'une instance et de rendre indépendant le calcul d'un attribut de l'instant de son déclenchement. Pour ce faire, le module de gestion du DP doit mettre en place un mécanisme capable de gérer les attentes et les reprises des tâches de calcul déclenchées (cf. §VI.3.5).

Le calcul d'un attribut dans une instance peut être déclenché à n'importe quel moment, notamment lorsque l'instance s'avère incomplète. Si la tâche se base sur les informations de l'instance à cet instant précis, il est peu probable qu'elle puisse fournir un résultat par faute de données d'entrées à consommer.

Afin de ne pas statuer par un échec sans avoir tenté avant de récupérer les valeurs d'attributs manquantes, le déclenchement du DP est suspendu, l'exécution de la tâche est alors

en attente. Dès que les informations nécessaires sont acquises, l'exécution peut reprendre pour aboutir soit sur un nouvel état d'attente, soit sur un état abouti (échec ou réussite). Le schéma suivant (cf. Figure VI.7) décrit la phase d'attente d'un DP.

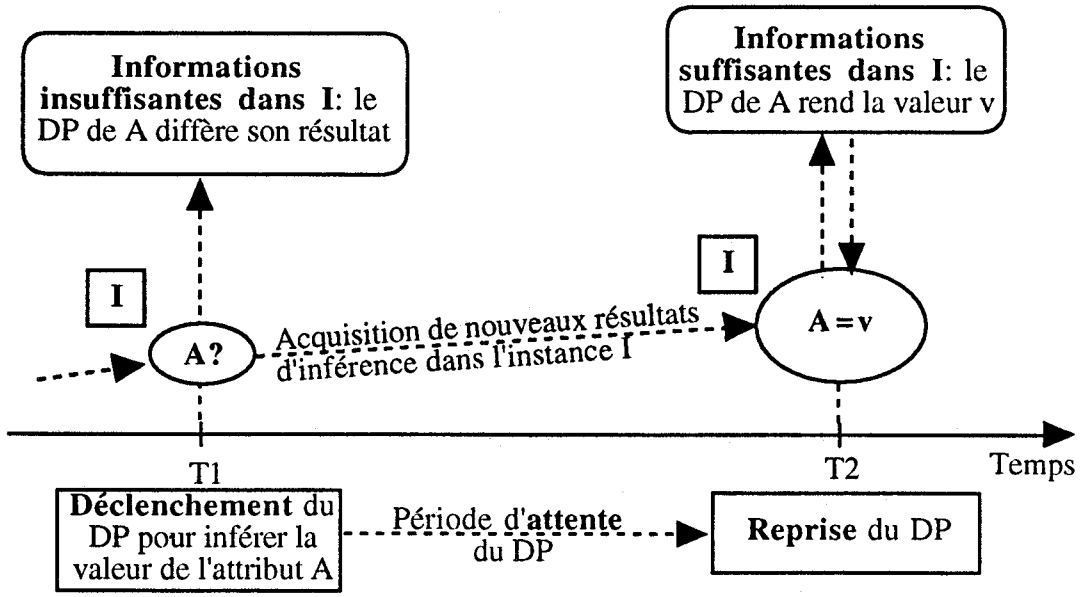

Figure VI.7 : Pour rendre le calcul global d'un attribut indépendant de l'état temporel d'une instance, le module de gestion du DP gère des états d'attente dans leur exécution. L'évolution de l'instance I de l'instant T1 à l'instant T2 a permis au DP de l'attribut d'aboutir.

Contrairement à la notion de calcul instantané que proposait l'attachement procédural, l'exécution d'un DP s'abstrait du temps pour ne dépendre que de l'état de l'instance. Cette solution d'attente permet alors un ordonnancement dynamique des inférences de connaissances sur l'instance en fonction de sa propre évolution. Cette possibilité est importante car l'acquisition de connaissances sur une instance est un processus irrégulier qui peut dépendre de plusieurs sources : utilisateur(s), déclenchements de DP, inférences par contraintes, etc.

3.4. Etats d'exécution d'un détachement procédural

De façon analogue à une contrainte posée sur un attribut, le déclenchement d'un DP reste maintenu au niveau de l'attribut demandé. Etant données les possibilités d'attente d'une exécution de DP, il est important que lors de la consultation d'un attribut le système soit à même de disposer des informations concernant son DP (notamment pour éviter de le redéclencher lorsqu'il est déjà en attente).

Quand un DP est déclenché, une association entre l'instance de tâche créée et l'attribut à évaluer est établie et enregistrée au niveau de cet attribut. Cette procédure rentre dans le cadre du travail du système de maintien du raisonnement (cf. §VI.3.5).

A travers cette association, le système possède le moyen de s'informer de l'état d'exécution de la tâche de calcul. Cette perception que le système a du comportement de la tâche se résume en trois états possibles :

- état "**attente**" : cet état correspond à la phase d'exécution de la tâche. Lorsque le système constate cet état, la tâche est en attente d'une valeur d'entrée.
- état "**aboutie**" : l'exécution de la tâche vient d'aboutir, c'est-à-dire :
  - **réussite** : proposition d'une ou de plusieurs valeurs pour l'attribut.

- **échec** : la stratégie d'évaluation mise en place n'est pas adaptée à la configuration de l'instance demandeuse, toutes les méthodes de calcul connues sont rejetées.

- état "**modifiée**" : cet état apparaît quand l'instance de tâche, étant dans un état "aboutie", se retrouve modifiée au niveau de ses entrées. La reprise de l'exécution avec la remise à jour des entrées modifiées est différée à la prochaine demande de la valeur de l'attribut qui a déclenché cette tâche.

A partir de ces états, les différentes opérations d'échanges de données entre l'instance appelante et l'instance de tâche peuvent être données sous forme de transitions (cf. Figure VI.8).

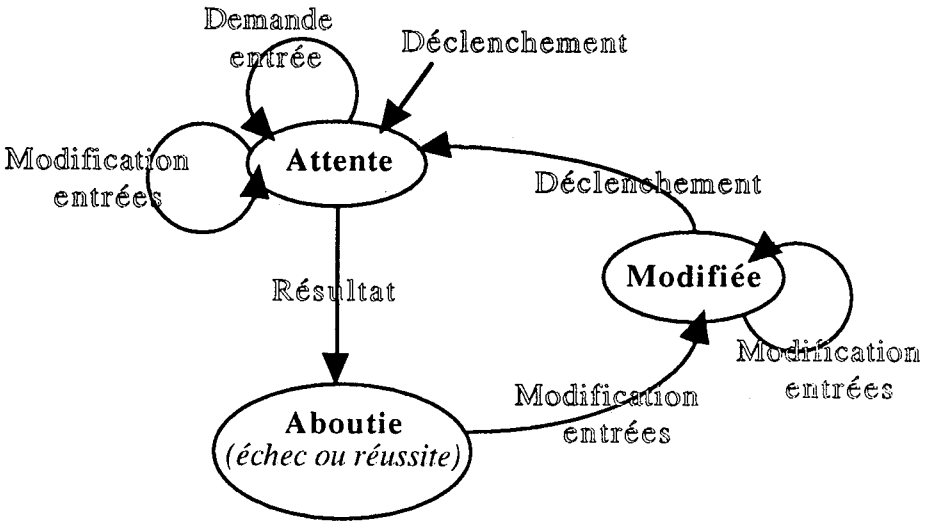

Figure VI.8 : Comportement d'une tâche schématisé par un ensemble d'états et les transitions possibles entre ces états.

A l'exception du déclenchement qui a déjà été présenté (cf. §VI.3.2), les différentes transitions sont précisées ci-dessous :

- **Demande d'entrée** : Lorsque l'attribut demandé est inconnu, la tâche reste en état d'attente. Par contre, lorsque l'attribut demandé est déjà évalué, l'entrée prend la valeur et la tâche peut reprendre sa stratégie.

- **Modification d'une entrée** : la tâche est en veille (état d'attente d'une valeur ou état modifié). la modification peut intervenir de deux façons possibles :

  - La modification s'opère sur une entrée en attente d'évaluation. La valeur de l'attribut demandé est récupérée et l'exécution reprend son cours.

  - La modification atteint une des entrées déjà évaluées. L'exécution doit être remise en cause. Pour ce faire, lorsque la stratégie sera **reprise**, la mise à jour de l'entrée modifiée sera prise en compte et confrontée à l'étape de stratégie courante. Si cette donnée ne concorde plus, l'exécution reprend au point de la stratégie où la valeur de cette entrée avait été initialement demandée, sinon elle reste en veille.

- **Résultat** : Cette phase correspond bien sûr à la donnée d'un résultat par la tâche. La sortie est soit évaluée, soit inconnue.

Lorsqu'un DP est dans un état de réussite, l'attribut de l'instance qui est concerné par ce DP reçoit pour valeur le résultat fourni.

Enfin, le cas particulier d'un attribut qui ne possède pas de DP dans sa déclaration, i.e. pas de moyen d'évaluation global connu, est traité comme un cas où le DP associé échoue systématiquement.

3.5. Réalisation effective de la gestion des attentes/reprises

Il s'agit ici de mettre en place le mécanisme permettant à une tâche déclenchée par le DP d'un attribut d'enchaîner les phases d'attente et de reprise d'exécution. La solution adoptée, et présentée par la suite, consiste à réutiliser le principe du système de maintien du raisonnement en mettant en place des dépendances entre l'instance et la tâche. L'ordonnancement des divers calculs d'attributs est inscrit dans cette structure de dépendances, il est assuré moyennant quelques aménagements de la gestion des dépendances.

3.5.1. Contrôle réparti vs contrôle centralisé

Gérer la dynamique des DP demandent un contrôle particulier afin d'assurer la reprise des exécutions de tâches en attente lors de modifications de la base. Ce contrôle revient à gérer un agenda des tâches activées. La gestion de cet agenda consiste à ordonner les tâches en fonction des demandes d'attributs effectuées et de permettre la reprise d'exécutions de tâches à chaque fois qu'une demande d'attribut en attente est satisfaite.

Mettre en place cet agenda de façon centralisé au niveau du moteur d'inférence serait lourd car il devrait prendre en compte toutes les exécutions de tâches liées à toutes les instances, et gérer l'ensemble de tous les attributs attendus par toutes ces tâches. Chaque modification de la base deviendrait alors coûteuse en temps étant données les vérifications qu'elle entraînerait au niveau de l'agenda.

Une solution plus réaliste consiste à répartir les informations de l'agenda au niveau des instances : chaque instance possède les informations concernant les DP qu'elle a déclenchés et qui restent en attente d'une valeur d'un de ses attributs. Pour ce faire, il suffit d'indiquer au niveau de chaque attribut les liens qu'il entretient avec les tâches en attente. Le contrôle peut alors être réparti puisque la reprise d'une tâche peut être effectuée lors de l'accès en écriture à un attribut : lorsqu'une valeur est enregistrée au niveau d'un attribut, cette information peut être diffusée à toutes les tâches qui l'attendent, et les réactiver.

Un tel contrôle réparti existe déjà pour gérer les répercussions d'une modification de la base de connaissances, c'est le rôle du système de maintien du raisonnement (cf. §III). Pour ce faire, ce système gère un ensemble de liens de dépendances entre les attributs et les instances de la base. Chaque inférence est ainsi enregistrée, liant les informations qui ont permis son aboutissement aux informations qu'elle a produites.

Aussi, le traitement des dépendances entre les attributs d'une instance n'étant pas un problème spécifique aux DP, le module de gestion de DP peut se décharger de l'ordonnancement des déclenchements. Toutefois, comme tout autre mécanisme d'inférence sur une instance, ce module doit inscrire les informations nécessaires à la gestion de ces dépendances, et notamment celles qui permettront la reprise d'une tâche en attente.

3.5.2. Mise en place de dépendances lors des échanges entre instance et tâche

L'un des problèmes principaux de la gestion des attentes réside dans la conservation des informations concernant l'échange des données : une demande de valeur d'attribut doit être enregistrée au niveau de l'attribut. Le principe adopté consiste à matérialiser les échanges de données entre l'instance et la tâche de calcul par la mise en place de liens de dépendances entre leurs attributs.

Une dépendance entre deux attributs possède un sens qui indique que la source est nécessaire à l'évaluation du destinataire. Dans le système, pour représenter une dépendance, la source est annotée par la référence au destinataire (la **descendance** de la source), et le destinataire enregistre, quant à lui, l'information concernant le moyen utilisé, ou à utiliser, pour l'évaluer à partir de la source (la **justification** du destinataire). Ainsi, un attribut dispose de la liste de tous les attributs qui dépendent de lui (sa descendance), ainsi que du moyen qui a permis, ou permettra, d'inférer sa propre valeur (sa justification). Si la justification d'un attribut

est une référence à un autre attribut, alors l'inférence correspond à l'affectation de la valeur du second au premier attribut.

Grâce aux dépendances, on dispose d'un moyen pour réaliser un échange de données différé (il existe un délai entre la demande et l'obtention de la valeur). Un tel échange est une opération entre deux objets qui se décompose en deux procédures :

- Enregistrement dans une instance de la demande d'une valeur d'attribut en indiquant dans la liste de **descendances** de cet attribut d'où provient exactement cette demande (cf. Figure VI.9).

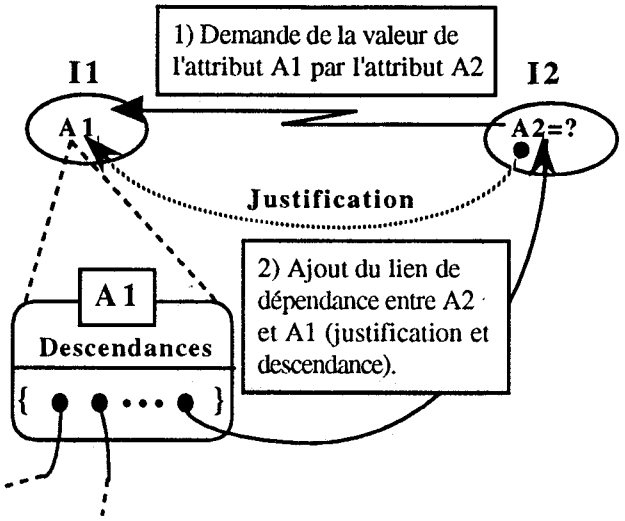

Figure VI.9 : Enregistrement d'une demande de valeur d'attribut par ajout d'un lien de dépendance.

- Diffusion, à partir de l'instance, d'une valeur d'attribut qui vient d'être acquise à tous les demandeurs (ceux de la liste de descendance) (cf. Figure VI.10).

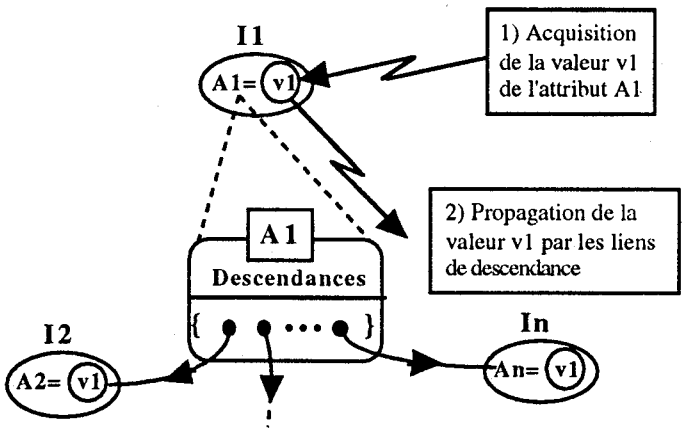

Figure VI.10 : Exploitation des liens de descendance pour propager la valeur d'un attribut.

Entre l'instance et la tâche de calcul d'un attribut, ce principe peut-être vu comme la mise en place de contraintes d'affectation entre les attributs de l'instance et ceux de l'instance de tâche (cf. Figure VI.11). Dès qu'une valeur d'un attribut demandé est connue, elle est affectée à l'attribut demandeur via le lien de descendance.

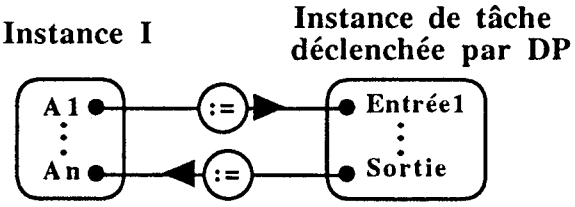

Figure VI.11 : La liaison entre une instance I et une instance de tâche se matérialise par la mise en place de dépendances entre les attributs de part et d'autre. Ces dépendances jouent le rôle de contraintes d'affectation.

Le principe d'utilisation du système de maintien du raisonnement doit permettre de reprendre une exécution de tâche d'un DP qui est en attente. C'est-à-dire que, dans un cas d'attente, la récupération d'une valeur d'entrée doit pouvoir déclencher la reprise de l'exécution de la tâche. Pour ce faire, la gestion des propagations de modification du SMR de TROPES proposé dans [Gens90] doit être aménagée car elle n'offre pas cette dynamique.

3.5.3. Problème de la propagation paresseuse du SMR

Prévu pour propager les effets d'une modification, le SMR effectue uniquement la propagation d'une marque (un drapeau est associé à chaque attribut d'une instance) de modification à travers les dépendances et ne relance aucune inférence. De ce fait, une inférence n'est reprise qu'à partir du moment où il y a à nouveau une tentative d'accès à un élément de connaissances dont le drapeau de modification est valide. Cette pratique permet de réaliser progressivement les modifications en fonction des besoins d'un utilisateur ou d'un raisonnement.

Dans le cas du déclenchement d'un DP, cette propagation paresseuse pose un problème dans la dynamique des déclenchements en chaîne des évaluations d'attributs d'une instance. Par exemple, lorsque le calcul d'un attribut A d'une instance nécessite en premier lieu l'évaluation d'un ou plusieurs autres attributs de cette instance, la tâche de calcul de A reste en attente. Dès que l'évaluation des autres attributs est assurée, leurs résultats doivent normalement être répercutés au niveau de la tâche de calcul de A via les dépendances qui ont été posées entre ces attributs et les entrées de la tâche. Etant donné le fonctionnement paresseux de la propagation proposée, la répercussion n'est pas effective : seuls les drapeaux des attributs d'entrée de la tâche seront mis à jour.

Aussi, bien que le calcul de l'attribut A soit une opération dont on attend le résultat et que sa tâche de calcul soit en mesure de poursuivre son exécution (les informations manquantes sont maintenant disponibles), le processus reste suspendu. L'attente persiste donc tant qu'il n'y a pas de nouvel accès à A.

Deux modifications doivent être apportées à ce fonctionnement : prise en compte des calculs immédiats lors de la propagation, et traitement du cas spécial d'une tâche lors de la propagation d'informations à une instance de tâche. Le premier point est traité en distinguant dans une instance les attributs demandeurs (dont l'évaluation est un problème en cours) des autres attributs. Le second point nécessite la mise à jour et la reprise de l'exécution d'une tâche lorsqu'une modification est propagée sur ses entrées. Cette modification peut aussi bien être une valeur attendue provenant d'un attribut demandé qu'une altération des entrées déjà évaluées due à la répercussion de modification à travers le réseau de dépendances établi.

3.5.4. Attribut demandeur

Afin de faire la différence entre le cas d'un attribut qui subit indirectement les effets d'une modification et le cas d'un attribut dont le raisonnement est en attente de son évaluation, il convient de distinguer deux types d'attributs dans une instance : les attributs stables et les attributs demandeurs. Soit I une instance, l'ensemble d'attributs de I, $\mathcal{A}(I)$, se divise en deux ensembles disjoints :

- $\mathcal{A}_D(I)$: l'ensemble des attributs **demandeurs** de I, c'est-à-dire les attributs qui ne sont pas évalués et qui ont leur DP en attente.
- $\mathcal{A}_S(I)$: l'ensemble des attributs **stables** de I, c'est-à-dire les attributs qui appartiennent aux catégories (partition de $\mathcal{A}_S(I)$) :
  - $\mathcal{A}_I(I)$: ensemble des attributs **inconnus** avec un DP inactif (non déclenché ou en échec).
  - $\mathcal{A}_E(I)$: ensemble des attributs déjà **évalués.**
  - $\mathcal{A}_M(I)$: ensemble des attributs à modifier au prochain accès en lecture (leur valeur est obsolète car dépend d'attributs qui ont été modifiés). C'est un cas d'attribut qui a été traitée par la propagation paresseuse des modifications.

Techniquement, la différence entre les attributs stables et demandeurs se fait simplement sur la donnée des informations concernant la valeur d'un attribut (inconnue ou non) et sur l'état de son drapeau de modification comme l'indique le tableau suivant :

Valeur	Drapeau "à-modifier"	Type d'attribut
• **Inconnue**	**faux**	**Stable** Etat d'attribut initial ou en échec
	vrai	**Demandeur** DP en attente
• **Connue**	**faux**	**Stable** Etat d'un attribut évalué
	vrai	**Stable** Attribut modifié, à recalculer

L'algorithme de propagation initialement proposé traitait tous les attributs comme des attributs stables. Les attributs demandeurs sont les attributs qui doivent être traités immédiatement par la propagation car ils participent à l'une des inférences dont le résultat est attendu par le raisonnement courant. Ainsi, la phase de propagation connaît deux modes : traitement de la propagation des résultats (non paresseuse), ou traitement de la propagation d'une modification (paresseuse).

3.5.5. Etats des attributs d'entrée/sorties et comportement d'une tâche

Lorsqu'une tâche est déclenchée par un DP, elle passe par un certain nombre d'états. Ces derniers dépendent des échanges de données entre les attributs de l'instance demandeuse et les attributs d'entrées/sorties de la tâche. Le mécanisme de propagation est donc chargé d'assurer les transitions d'états de la tâche en fonction des états de ces attributs.

A tout moment une instance de tâche doit se trouver dans l'un de ces états, il convient donc de caractériser précisément ces états en fonction des attributs mis en cause (cf. Figure VI.12).

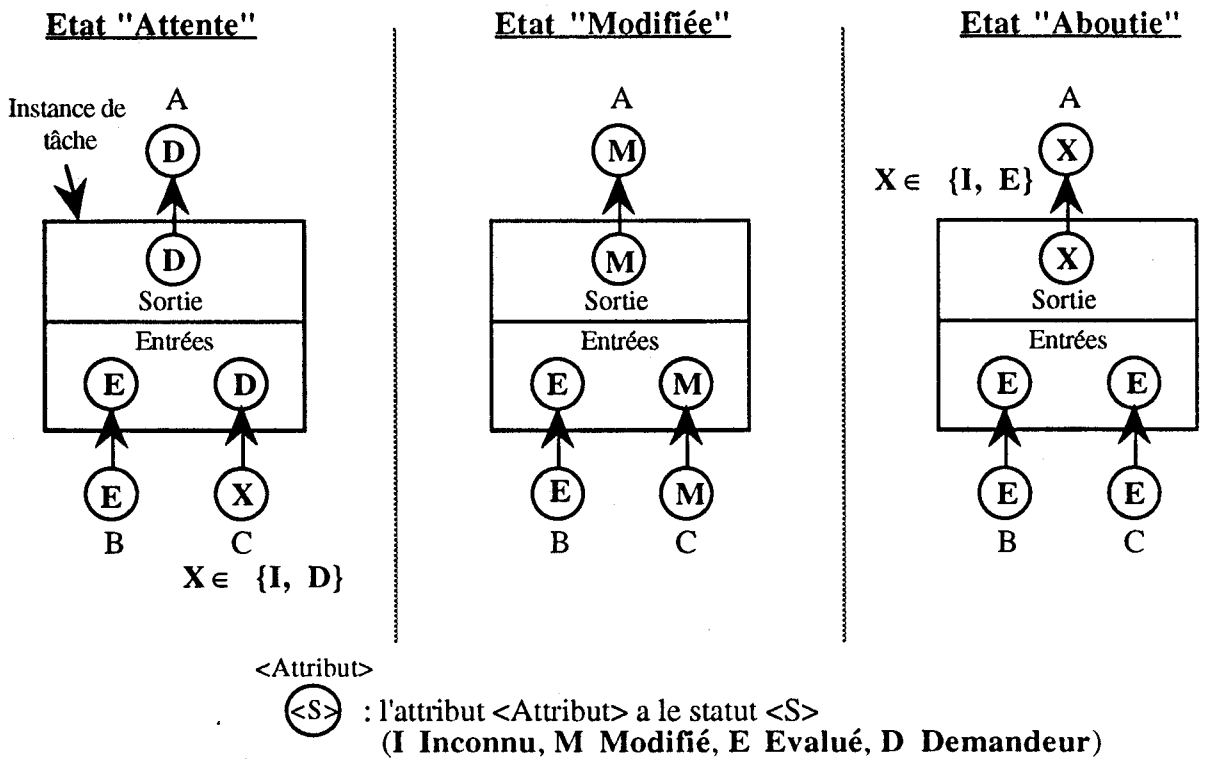

Figure VI.12 : Description des états d'une instance de tâche en fonction des états possibles des attributs mis en jeu. La tâche considérée ici permet le calcul de l'attribut A en fonction des attributs B et C. Durant l'exécution d'une tâche, les dépendances sont mises en place entre ces attributs et les attributs d'entrées/sorties correspondants de la tâche. Ces dépendances sont données ici par les liens de descendance pour montrer le sens de circulation des données.

- **Etat "attente"** : l'état d'attente correspond à la phase d'exécution de la tâche. Au niveau de la base, cet état d'une instance de tâche répond aux exigences suivantes :
  - l'attribut de sortie de la tâche et l'attribut de l'instance que la tâche doit calculer sont tous deux dans l'état **demandeur**.
  - il existe un attribut d'entrée de la tâche qui est dans l'état **demandeur**, l'attribut qui lui correspond dans l'instance est dans l'état soit **demandeur**, soit **inconnu**.
  - S'il existe d'autres attributs d'entrées, ils sont tous **évalués**.

- **Etat "aboutie"** : l'exécution de la tâche vient d'aboutir. Etant donné la description de l'état d'attente précédente, l'état "aboutie" correspond aux états d'attributs suivants :
  - l'attribut de sortie et l'attribut de l'instance que la tâche doit calculer sont tous deux dans l'état soit **inconnu** en cas d'échec de la tâche, soit **évalué** en cas de réussite de la tâche.
  - Tous les attributs d'entrées (s'il en existe) et les attributs correspondant dans l'instance sont dans l'état **évalué**.
- **Etat "modifiée"** : cet état apparaît quand l'instance de tâche, étant dans un état "aboutie", se retrouve modifiée au niveau de ses entrées. Il correspond à un traitement paresseux de la propagation des modifications. L'état des attributs de l'instance de tâche correspond alors à :

- l'une des entrées au moins est dans l'état **modifié**.
- l'attribut de sortie de la tâche et l'attribut de l'instance que la tâche doit calculer sont dans l'état **modifié**.

En dehors du déclenchement, toutes les transitions entre les états d'une tâche correspondent au traitement d'une mise à jour d'un des attributs de la tâche : soit l'une des entrées est modifiée, auquel cas la tâche reprend son exécution ; soit la sortie est mise à jour par l'exécution de la tâche et le résultat est propagée à l'attribut demandeur. Pour mettre en œuvre cette dynamique, il faut mettre en place la propagation des mises à jour qui est primordiale lorsqu'il y a des enchaînements de tâches et des états d'attente. Le résultat d'une telle propagation est de stabiliser un graphe de dépendances qui est altéré par une modification d'attribut.

3.5.6. Propagation d'une mise à jour

La dynamique de l'exécution des tâches de calcul réside dans la propagation des mises à jour. Lorsqu'un attribut participant dans les calculs est modifié, la propagation doit assurer la reprise des tâches en état d'attente qui l'utilisent. La propagation est donc déclenchée par l'opération de mise à jour d'un attribut.

Le résultat de cette opération est l'obtention d'un nouveau graphe de dépendances dans lequel les états des tâches de calcul sont mis à jour, la modification d'attribut ayant occasionné une altération de ces états.

3.5.6.1. Données initiales

Les données de la propagation sont un attribut modifié et un graphe de dépendances entre attributs (données par les descendances et les justifications des attributs). Le graphe est orienté par les liens de descendance (liste des descendances) qu'un attribut entretient avec les attributs dont le calcul dépend de lui. Chaque nœud du graphe correspond à un attribut qui est étiqueté par son statut (**inconnu**, **modifié**, **demandeur**, **évalué**). Parmi ces nœuds, il y a les entrées et sorties des instances de tâches. La propagation à travers une instance de tâche est assurée de façon à ce que la modification d'une entrée puisse être répercutée sur la sortie (si besoin est). Une instance de tâche peut être vue comme la donnée d'un ensemble de dépendances entre la sortie et les entrées.

L'hypothèse raisonnable posée initialement est que le graphe des dépendances est **sans circuit**. Un circuit dans un tel graphe impliquerait que le calcul d'un attribut dépend de lui-même, ce qui est une situation bloquante et sans autre solution que l'intervention de l'utilisateur afin de casser le cycle (par la donnée d'une valeur à un attribut du cycle). On suppose donc que tous les problèmes de cycle ont été résolus lors d'une phase de vérification mise en place dès la pose des dépendances, c'est-à-dire des déclenchements de calcul.

La propagation commence au niveau de l'attribut mis à jour qui constitue la racine du graphe de descendances qui va être traité.

3.5.6.2. Propagation en deux phases

Lorsqu'un attribut déjà évalué est modifié, il est important de savoir, avant de relancer des calculs, quels sont les attributs perturbés par cette modification pour ne pas effectuer des calculs déclenchés en suivant un chemin qui seraient déclarés obsolètes en suivant un autre chemin (cf. Figure VI.13).

La solution d'un ordonnancement des demandes d'attributs via la liste des descendances d'attribut est hélas à proscrire. Dans l'exemple suivant, il n'est pas certain que ce soit le calcul de A qui ait déclenché les calculs de B et C. En effet, B avait pu être demandé par l'utilisateur avant que C soit lui-même demandé par A. On ne peut donc être sûr de l'ordre au niveau de la liste des descendances de D.

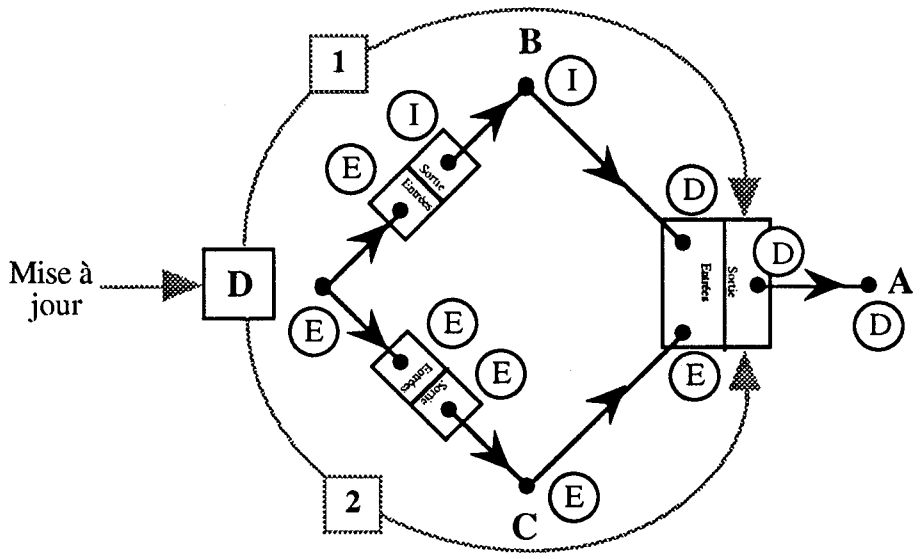

Figure VI.13 : Illustration du problème de l'ordre du parcours. Dans cet exemple, A est calculé en fonction des attributs B et C qui sont eux-même calculés en fonction de D. L'étiquetage des nœuds indiquent que A est demandeur de B, C est évalué grâce à D, B est inconnu bien que D soit évalué. La modification de D entraîne une propagation. Si cette propagation suit en premier le chemin 1, B pourrait être recalculé, entraînant par la même le recalcul de A. Or, le chemin 2 indique que le calcul de A doit prendre en compte la modification de C. La propagation par le chemin 2 doit être effectuée en première pour éviter un calcul obsolète de A.

Pour résoudre ce problème, la propagation s'effectue en deux phases : recalcul des statuts d'attributs à partir de l'attribut modifié, puis reprise des calculs nécessaires en partant de l'attribut modifié.

3.5.6.3. Recalcul des statuts

Cette première phase correspond à une propagation paresseuse de marques. Les branches dont les recalculs doivent être relancés seront marquées du statut **demandeur**, celles dont les recalculs peuvent être différés seront marqués du statut **modifié**.

Cette phase peut être perçue comme une restitution du marquage d'origine du graphe, c'est-à-dire lorsque les calculs des déclenchements étaient dans l'état d'attente de la valeur de l'attribut modifié. Le problème revient donc à déterminer pour chaque nœud du graphe s'il est demandeur ou modifié. Dans le premier cas, il s'agit d'un attribut qui possède dans ses descendances un attribut demandeur, et dans le deuxième cas, c'est un attribut qui était évalué et dont tous les attributs descendants étaient eux-même évalués.

Aussi, pour effectuer ce recalcul de statuts, il faut faire un parcours en profondeur, c'est-à-dire **déterminer le statut de tous les nœuds descendants avant de déterminer le statut du nœud courant**. Ensuite il suffit d'appliquer les règles suivantes :

- S'il existe un des descendants déclaré **demandeur** ou **inconnu**, le nœud courant passe à **demandeur** aussi.
- Sinon il est déclaré **modifié**.

L'exploration en profondeur s'arrête sur une branche dès qu'un attribut rencontré est soit une feuille du graphe, soit demandeur, soit modifié.

La figure suivante (cf. Figure VI.14) illustre une phase de recalcul de statuts d'un graphe de descendance. Les instances de tâches ne sont pas reprises pour alléger le dessin, elles sont implicitement présentes entre les nœuds de chaque niveau.

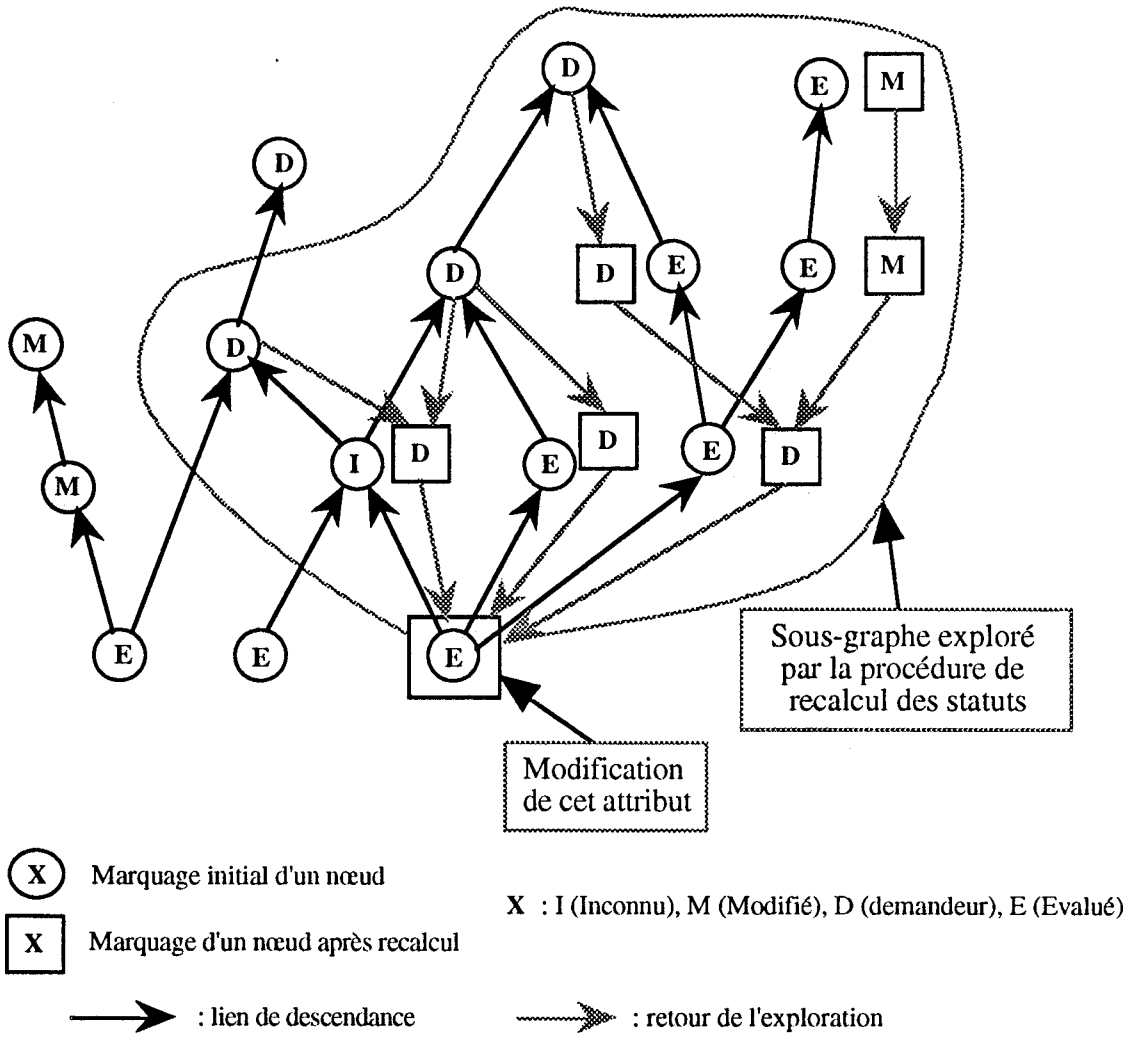

Figure VI.14 : Illustration du parcours d'un graphe de descendances pour le recalcul du statut de chaque nœud impliqué par une modification. Le parcours est effectué en profondeur, le statut des nœuds les plus profonds sont calculés, puis les nœuds dont ils dépendent, et ainsi de suite jusqu'à revenir à la racine, c'est-à-dire le nœud modifié.

Il est à noter qu'avec ce traitement deux cas particuliers sont naturellement traités :

- aucun des calculs n'étaient en attente, le marquage sera celui d'une propagation paresseuse (statut **modifié**).
- les calculs étaient en attente de cette valeur, les descendants directs de l'attribut modifié étaient déjà tous **demandeurs**, le recalcul des statuts s'arrêtent donc tout de suite, on passe à la phase des reprises de calcul immédiatement.

Après cette phase de remise à jour des statuts, la phase de reprise des calculs peut avoir lieu.

3.5.6.4. Reprise des calculs

La phase de recalcul des statuts effectuée, il reste à propager la modification de l'attribut pour éventuellement relancer les tâches de calcul qui attendent sa valeur en entrée. Seule la partie du graphe en partant de l'attribut modifié et comportant des attributs dans l'état **demandeur** est concernée par cette propagation.

La propagation proposée consiste simplement à mettre en place à partir de l'attribut modifié la **reprise de l'inférence** de chaque attribut de sa descendance directe qui est **demandeur**. Chacun de ces attributs étant lui-même mis à jour propagera à son tour sa propre modification, la phase de recalcul des statuts étant alors évitée puisque tous ses descendants sont déjà demandeurs.

La **reprise d'une inférence** d'un attribut sera interprétée en fonction du type de l'instance auquel appartient l'attribut (il suffit de consulter le concept d'appartenance de l'instance) :

- Si l'attribut n'est pas un attribut (d'entrée) d'une tâche, sa justification comporte la référence de l'attribut dont il dépend, sa valeur lui est alors affectée.
- Si l'attribut est un attribut d'entrée d'une tâche, sa valeur est aussi récupérée via l'attribut dont il est le demandeur mais l'exécution de la tâche est, de plus, reprise en tenant compte de toutes les modifications d'entrées que la phase de recalcul de statut a pu signaler.

Grâce à la première phase, l'ordre de cheminement de la propagation des mises à jour ne risque plus d'entraîner une reprise d'exécution avec des données périmées. Cependant, telle qu'elle est proposée, cette façon de faire n'assure pas qu'au moment de la reprise de l'exécution la tâche disposera des nouvelles valeurs d'attributs d'entrée, certains d'entre eux étant encore demandeurs car ils n'ont pas encore été traités par la propagation des mises à jour. Dans ce cas, la reprise consiste à retrouver un état d'attente stable pour la tâche étant données les modifications d'entrée, puis d'arrêter la propagation des mises à jour selon ce cheminement.

3.5.7. Exemple

L'exemple suivant illustre l'enchaînement des étapes (cf. Figure VI.15). Dans cet exemple, une instance possède trois attributs dont les attributs *Age* et *An_Nais* sont inconnus tandis que l'attribut *N_Insee* est évalué. Le scénario proposé montre le résultat de la demande de l'attribut *Age*. La première étape correspond au déclenchement du DP avec la création d'une instance de tâche *Calc_Age*. L'exécution de cette tâche demande dans un premier temps la valeur de l'attribut *An_Nais*. Etant lui-même inconnu, son DP est déclenché. Cette fois-ci la tâche d'exécution correspond à une procédure de demande à l'utilisateur. Répondant par la valeur inconnue, le DP interprète cette réponse comme un échec.

L'échec de l'évaluation de l'attribut *An_Nais* se traduit par une mise à jour de l'attribut de sortie de sa tâche de calcul *Demander_An_Nais* avec la valeur inconnue. Cette mise à jour déclenche une propagation qui fait l'économie d'une phase de recalcul des statuts d'attributs puisque la seule descendance, l'attribut *An_Nais*, possède un statut de demandeur. Le résultat de la propagation est donc le passage de *An_Nais* et de l'entrée de la tâche de *Calc_Age* à inconnu.

La mise à jour de l'attribut d'entrée de la tâche déclenche sa reprise. La stratégie étant en phase de demande prospective d'attribut abandonne la demande de *An_Nais* au profit de l'attribut *N_Insee* qui est, quant à lui, évalué. De ce fait, le nouvel attribut d'entrée est mis à jour avec sa valeur, le calcul peut se poursuivre et l'attribut de sortie de *Calc_Age* se trouve évalué. Sa mise à jour déclenche une propagation à l'attribut *Age* qui fournit le résultat escompté.

Il faut noter que dans un scénario où l'attribut *An_Nais* posséderait un DP permettant son calcul en fonction de l'attribut *N_Insee*, l'évaluation de *An_Nais* n'échouerait pas. En effet, la demande de *An_Nais* déclencherait automatiquement son DP. La valeur de *N_Insee* étant connue, la tâche de calcul de *An_Nais* rendrait alors sa valeur. Dans ce cas, l'attribut *Age* serait ensuite évalué en fonction de l'attribut *An_Nais*.

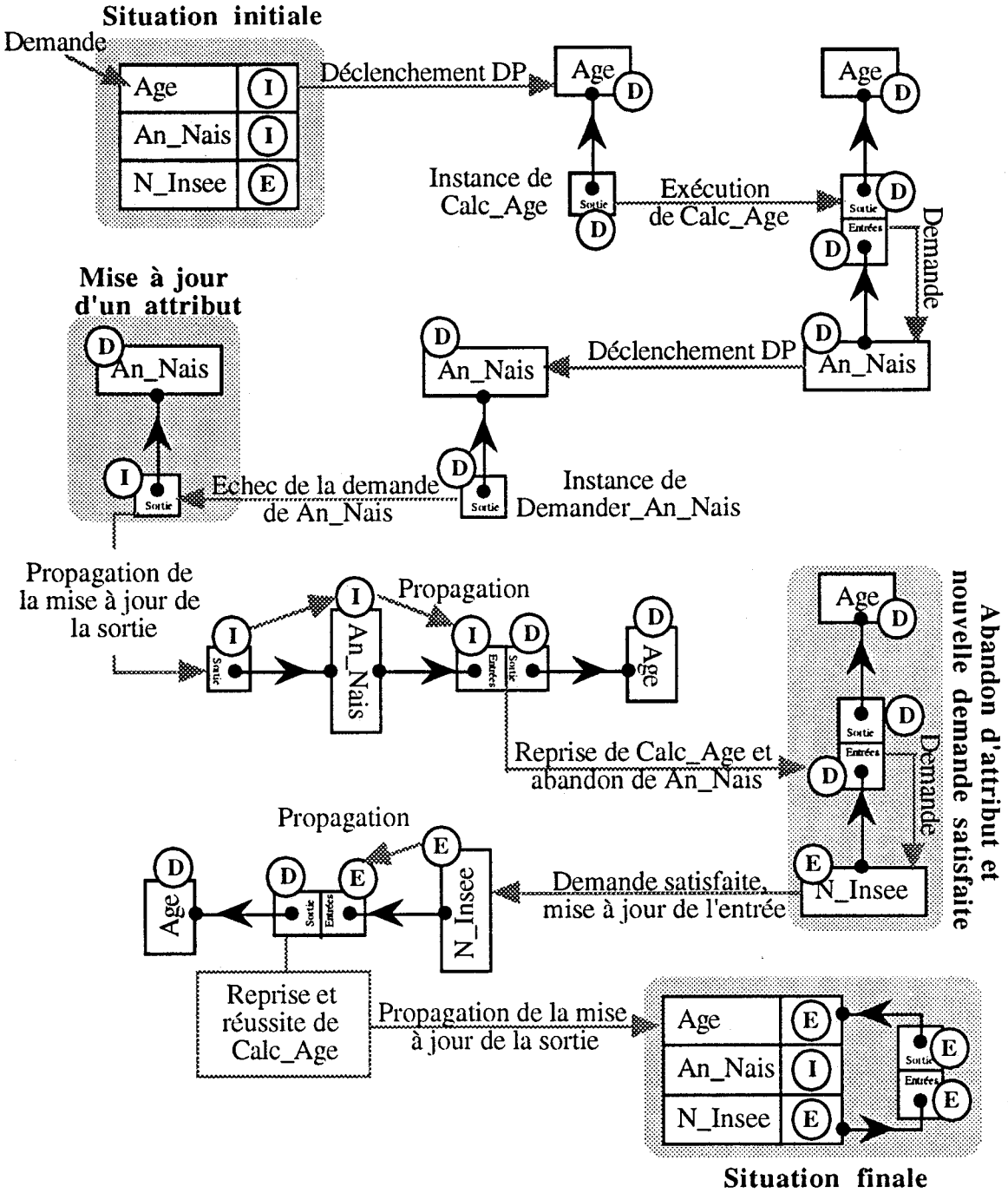

Figure VI.15 : Exemple de l'enchaînement des évaluations de l'attribut *Age* et *An_Nais*, puis abandon de *An_Nais* au profit de *N_insee* qui permet de mener le calcul d'*Age* à bon terme. Deux grandes phases se dessinent dans ce mécanisme : la première phase établit les différents liens exprimant les dépendances entre les attributs de l'instance appelante et les tâches déclenchées par les DP, la seconde phase prend naissance à chaque mise à jour et consiste à répercuter l'obtention des résultats attendus et à reprendre les calculs suspendus.

Maintenant, si l'utilisateur décide de changer la valeur de l'attribut *N_Insee*, sa mise à jour déclenchera une phase de recalcul des statuts des attributs en partant de *N_Insee*, passant par les attributs d'entrée et de sortie de l'instance de tâche *Calc_Age* et aboutira sur *Age*. Tous, excepté *N_Insee* qui vient d'être mis à jour, passeront au statut modifié puisqu'il n'y a aucun demandeur de l'attribut *Age*. Il ne sera donc pas recalculé tout de suite. Par contre, lors de la

prochaine lecture de cet attribut, le calcul sera de nouveau effectué en prenant en compte la nouvelle valeur d'attribut *N_Insee*.

4. Conclusion

La version initiale de TROPES est consacrée essentiellement au raisonnement classificatoire. Elle n'intègre donc pas de mécanisme d'inférence de valeurs d'attributs.

Les deux extensions présentées dans ce chapitre ont donc pour objectif d'augmenter les capacités d'inférences de connaissances sur les objets afin d'offrir au modèle des possibilités plus riches de construction d'objets.

La première prend place directement au niveau de la description des classes par l'introduction de la notion de contraintes, tandis que la deuxième propose d'associer à chaque attribut d'un objet une stratégie (une tâche) d'évaluation globale, affranchie de toute contrainte de rattachement de l'objet à une classe, et indépendante de l'instant de son déclenchement.

L'intégration de contraintes dans TROPES constitue une extension majeure du langage de description des objets. La bibliothèque de contraintes que le module MICRO met à la disposition de TROPES permet d'introduire des expressions d'interdépendances complexes entre les attributs d'une classe. Notamment, par le biais de la notion de chemin, les contraintes peuvent décrire des dépendances entre un objet et les objets que décrivent ses attributs complexes.

Le couplage de MICRO avec le module de types est avantageux dans les deux sens. En premier lieu, il permet aux modules de gestion des contraintes de bénéficier de la normalisation et des pré-calculs de domaines d'attributs pour initialiser les variables contraintes correspondant aux contraintes d'une classe. En second lieu, il fournit un moyen au module de types de tenir compte des contraintes posées dans le calcul des types d'attributs d'une classe.

La maintenance d'un réseau de contraintes posé sur une instance assure la vérification dynamique de la consistance du tout. Pour prendre en compte les contraintes d'une classe (et de ses sur-classes) lors d'un rattachement, il suffit donc de poser l'ensemble des contraintes sur l'instance. Une fois posées, les contraintes sont maintenues. Aussi, les modifications des attributs contraints sont automatiquement répercutées par le mécanisme de maintenance des contraintes.

Par ailleurs, la réduction dynamique des domaines de variables contraintes permet d'inférer des valeurs d'attributs d'une instance. De ce fait, le rattachement d'une instance à une classe peut déclencher l'évaluation d'attributs inconnus.

Les primitives permettant d'exprimer les contraintes sont à disposition d'un utilisateur. Elles peuvent donc être utilisées par lui pour poser dynamiquement des contraintes sur une instance qui les cumulera avec les éventuelles contraintes de ses classes de rattachement.

Outre l'extension de la bibliothèque de contraintes, deux perspectives principales sont en cours d'étude : la gestion des modifications dynamiques de contraintes et la satisfaction des contraintes. La première perspective soulève essentiellement le problème de la suppression d'une contrainte qui a été posée sur une instance (l'ajout est déjà une fonctionnalité omniprésente du couplage entre TROPES et MICRO). La seconde perspective a pour objectif l'exploration des différentes solutions d'un réseau de contraintes. Appliquée aux instances, cette possibilité permettrait de proposer pour certains attributs à valeur inconnue un ensemble de valeurs possibles, des valeurs hypothétiques d'attribut. Il est à noter que cette perspective rejoint la problématique de la construction de versions hypothétiques d'instance proposée dans la suite de ce document (cf. §VIII).

Le mécanisme de détachement procédural présenté dans cette partie rompt avec le principe classique de l'attachement procédural en permettant d'exprimer hors des classes l'association entre un attribut et ses méthodes de calcul. Considéré comme une connaissance globale, le moyen d'évaluation d'un attribut est fourni dans TROPES au niveau de la déclaration des attributs d'un concept. Ce moyen est donc utilisable par une instance indépendamment de tout rattachement à une classe et spécialement lors d'une phase de classification.

Ce mécanisme d'évaluation globale d'un attribut est déclenché dès que la valeur de cet attribut fait défaut à une instance. Pour mener à bien son calcul, ce mécanisme dispose des autres attributs présents dans l'instance. Le problème de la sélection d'une méthode de calcul particulière est résolu en exprimant le moyen d'évaluation sous forme d'une tâche qui adapte sa stratégie de calcul en fonction des données mises à disposition par l'instance appelante.

L'originalité de ce mécanisme réside dans la mise en place d'une phase de coopération entre la tâche déclenchée et l'instance qui requiert une valeur d'attribut. En effet, cette coopération consiste à constituer dynamiquement et progressivement l'ensemble des entrées suffisant pour mener à bien l'évaluation de l'attribut. Notamment, le mécanisme est capable de maintenir en attente l'exécution d'une tâche lorsque l'instance ne dispose pas de certaines données nécessaires en entrée de la tâche. Ce fonctionnement permet de tenir compte des évolutions futures de l'instance.

Techniquement, le déclenchement de l'évaluation d'un attribut correspond à la création d'un objet représentant la tâche à exécuter. La mise en place de liens de dépendance entre les attributs de cette instance de tâche et l'instance appelante permet d'assurer les échanges différés de données qui caractérisent un cas d'attente. La transition entre l'attente et la reprise d'exécution de la tâche est alors réalisée en étendant le mécanisme proposé par le système de maintien du raisonnement de TROPES pour propager les mises à jour d'attributs à travers les liens de dépendance.

Cette double extension du modèle TROPES permet d'exprimer des situations complexes et diverses d'inférences de valeurs d'attributs sur une instance. Cependant, il convient de définir clairement comment les connaissances ainsi exprimées peuvent coexister ensemble et dans quelle mesure elles peuvent influer sur la définition des objets et, donc, sur le raffinement d'une instance. Ces problèmes s'inscrivent dans une problématique plus vaste : le contrôle d'évaluation des attributs dans le modèle TROPES. Cette problématique constitue le thème central du chapitre suivant (cf. §VII).

Chapitre VII
CONTROLE D'EVALUATION DES ATTRIBUTS

1. Introduction

L'intégration de différents mécanismes d'inférence d'attribut au sein du système à objets TROPES constitue une extension nécessaire à la mise en place d'un mécanisme combinant phases d'identification et de construction d'instance. En cela, le chapitre précédent apporte un premier élément de réponse en montrant comment deux mécanismes d'obtention de valeurs d'attributs, les contraintes et le détachement procédural, peuvent être intégrés dans un modèle à objets, tout en respectant ses propriétés classificatoires. Cependant, ces deux intégrations étant réalisées indépendamment l'une de l'autre, le problème de leur coexistence au sein du modèle TROPES, c'est-à-dire le problème de concurrence qui peut se poser lors de l'évaluation d'un attribut, a été négligé. C'est donc afin d'assurer une intégration complète et cohérente que le présent chapitre est consacré à la mise en place du contrôle d'évaluation des attributs d'une instance.

Ce problème de contrôle est abordé en considérant tous les moyens par lesquels peut être fixée la valeur d'un attribut dans le système TROPES. L'ensemble de ces moyens ne se réduit donc pas seulement aux mécanismes d'inférences par contraintes et par détachement procédural, mais comprend aussi les opérations de configuration de l'utilisateur, le partage de valeurs d'attributs entre différentes instances et l'exploitation des facettes défaut.

L'étude préliminaire d'une solution de type **masquage**, principe mis en place dans les langages de *frames* pour résoudre un problème similaire, permet de souligner les problèmes particuliers auxquels doit répondre le contrôle dans le modèle TROPES (cf. §VII.2). Les principales critiques adressées à ce type de solution sont, en général, son caractère trop pragmatique et trop technique et, en particulier, son insuffisance à répondre aux problèmes particuliers posés par un modèle classificatoire. La première critique peut se résumer ainsi : l'exploitation des facettes d'inférences d'attributs relève d'une démarche plus proche de la programmation que de la description de connaissances et, par là-même, peut avoir des incidences notables, éventuellement regrettables, sur la façon de construire les hiérarchies de classes d'une base de connaissances. La seconde critique tend à montrer la difficulté à exprimer par masquage un contrôle capable de gérer correctement l'évaluation des attributs dans un contexte où le rattachement d'une instance fait l'objet de raffinements successifs.

Ces critiques du principe de masquage permettent d'envisager une nouvelle approche du contrôle visant à établir et à se reposer sur les dépendances pouvant exister entre la façon dont est identifié un objet et la façon dont il est construit. La stratégie de construction d'un objet étant décrite par l'ensemble des stratégies d'évaluation de ses attributs, l'effort de modélisation est donc focalisé sur l'expression et la mise en place du contrôle d'évaluation des attributs d'une instance.

L'évaluation d'un attribut repose sur un modèle de connaissances, dites *de production*, chargé d'identifier les différentes associations, établies par la description des objets, entre un contexte d'identification particulier (une classe) et l'emploi d'un moyen d'obtention de valeur (cf. §VII.3). Aussi, de la même façon que la phase de typage permet d'extraire de la description des objets les différents domaines de valeurs d'un attribut (cf. §II.3.1.1), une phase de recensement est mise en place pour identifier, rassembler et organiser les connaissances de production associées à chaque attribut.

Pour ce qui est de l'organisation des connaissances de production associées à un attribut, le modèle reprend et étend la notion de priorité sur lequel se base le principe de masquage des langages de *frames*. Le concepteur de la base peut ainsi exprimer pour chaque attribut une priorité visant à obtenir un ordonnancement global des connaissances de production qui lui sont

associées (cf. §VII.3.3). L'ensemble ainsi ordonné représente la stratégie d'évaluation d'un attribut définie par la description des objets.

Le modèle de connaissances de production est exploité pour mettre en place le contrôle d'évaluation des attribut d'une instance (cf. §VII.4). La consultation de l'ensemble de connaissances de production d'un attribut permet au contrôle d'établir quand et comment déclencher une inférence. Sur la base de cette consultation, le contrôle reprend et généralise la notion d'état d'attente d'évaluation d'un attribut présentée dans le cadre du détachement procédural.

Enfin, le principe du contrôle d'évaluation des attributs est appliqué dans le cas des attributs composants (cf. §VII.5). Le problème abordé est celui de la construction des composants d'un objet composite dans un contexte de raffinement.

2. Insuffisance d'une solution de type masquage

Le problème du contrôle des connaissances de production n'est pas sans analogie avec celui de l'héritage des facettes **$valeur**, **$defaut**, et **$si-besoin**, des langages de frames (cf. §IV.3.2.1). Dans ces langages, la solution adoptée consiste généralement à fixer une politique de choix globale au modèle de connaissances. Cette politique se traduit par la mise en place de règles de priorité qui sont à prendre en compte lors de la consultation et de l'héritage des différentes facettes. Ce principe, dit de *masquage*, s'inspire directement du mécanisme de surcharge de méthodes adopté dans les langages de programmation par objets.

Bien qu'une telle adaptation du mécanisme d'héritage au cas des connaissances de production procure un moyen modulaire et concis d'exprimer leur contrôle, cette solution s'avère insuffisante pour spécifier le comportement des connaissances de production dans un modèle permettant le raffinement d'une instance dans une hiérarchie de classes. En effet, le principe de masquage étant uniquement basé sur l'héritage de classes, il est difficile, voire impossible, d'assurer sa validité dans un modèle qui permet à une instance de changer de contexte d'héritage.

L'exemple qui suit a pour objectif de montrer les problèmes que pose le principe de masquage dans un système à objets de type classificatoire comme TROPES et de montrer que les modalités d'évaluation d'un attribut ont une incidence considérable sur la description des objets dans laquelle apparaît cet attribut.

2.1. Exemple d'emploi de connaissances de production

Cette partie introduit la description d'un concept représentant un ensemble de voitures et décrit un scénario possible d'exploitation de ce concept qui servira de support aux propos des parties suivantes.

2.1.1. Description du concept VOITURE

Dans la description du concept VOITURE présentée ci-dessous (cf. Figure VII.1), les connaissances de production, introduites à l'aide de détachement procédural, de contrainte et de facette **défaut**, permettent de définir les différentes politiques de facturation d'une voiture qui peuvent être pratiquées par les diverses marques considérées. Une politique de facturation est décrite en plusieurs étapes (prix TTC, prix HT, prix du moteur,...) et à différents niveaux d'abstraction (le concept général de voiture, la marque, le modèle,...).

La spécialisation choisie dans cet exemple correspond à celle des marques et de leurs différents modèles. L'exemple montre la politique complète de facturation de la marque Marque-N pour le modèle Modèle-M. Cette facturation s'organise autour du calcul du prix TTC qui est global à toutes les voitures et se base sur le prix HT. Ce dernier, spécifique à la marque Marque-N, est défini comme la somme du prix du moteur (Moteur.prix) et du prix de base du modèle (PrixBase). Le défaut associé à PrixBase dans la classe Modèle-M est chargé d'exprimer qu'à l'exception du cas de la série spéciale "jeune", le prix de base de ce modèle est fixé à 50000.

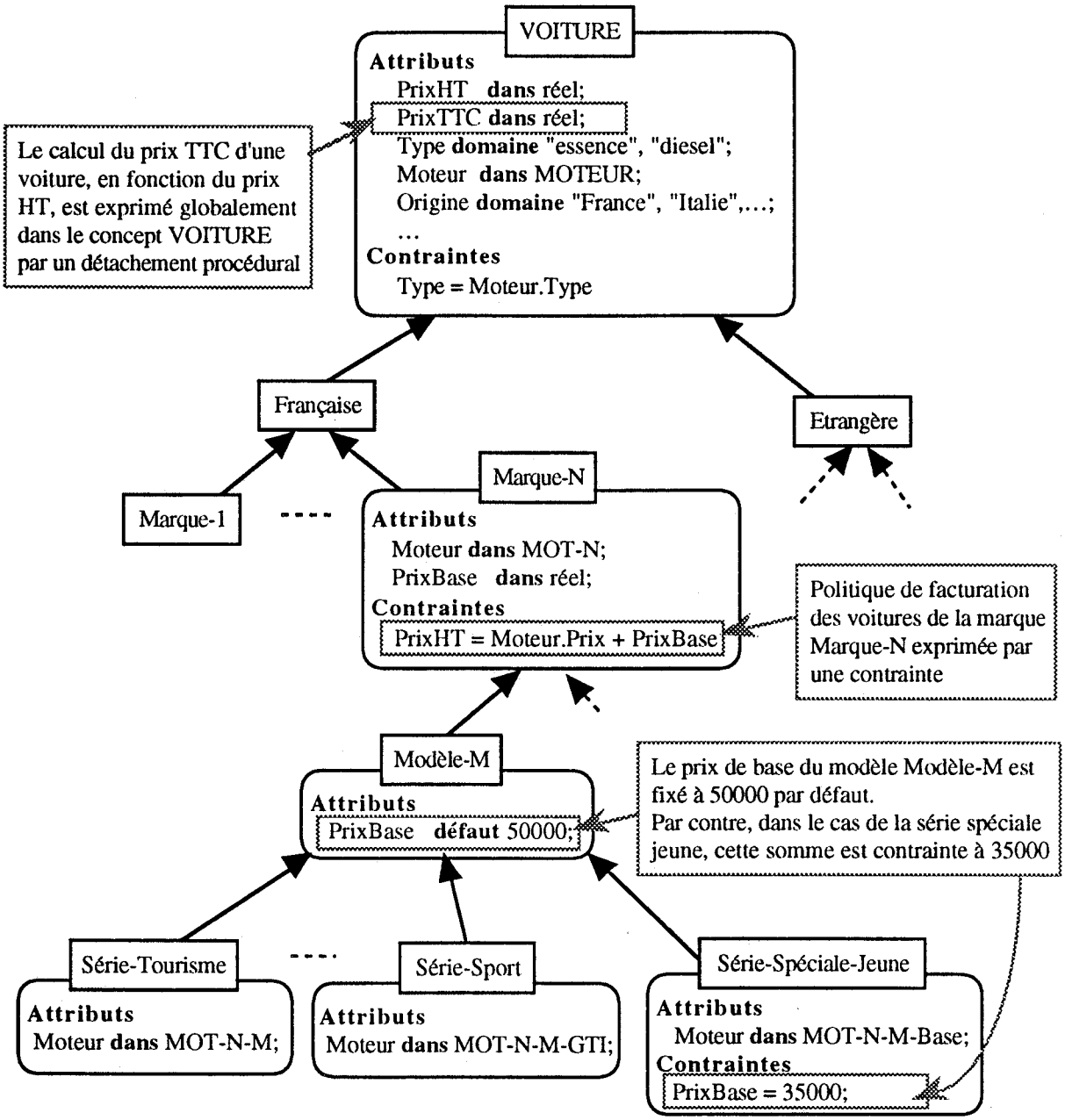

Figure VII.1 : Exemple de la hiérarchie de classes du concept VOITURE.

Le concept VOITURE est supposé prendre place dans la base de connaissances d'un système d'aide à un vendeur de voitures neuves. A moins que ce vendeur ne soit peu scrupuleux et définisse sa marge bénéficiaire à la tête du client, il préfèrera déléguer au système le soin de gérer lui-même la facturation des voitures en fonction des marques et des modèles. Le scénario suivant décrit rapidement un cas d'exploitation de ce concept dans lequel le vendeur cherche à la fois à identifier et à construire une instance de voiture.

2.1.2. Scénario d'exploitation du concept VOITURE

Outre les fonctionnalités standards comme la gestion de son stock et de sa clientèle, le vendeur attend de ce système qu'il lui permette d'utiliser la hiérarchie des voitures comme un catalogue interactif présentant les différents modèles qu'il peut proposer à ses clients. Le fonctionnement de ce catalogue doit permettre au vendeur de définir progressivement la voiture que son client désire. Cette définition doit tenir compte aussi bien des goûts que des contraintes financières du client.

Pour mettre en œuvre une telle démarche, le vendeur souhaite simplement créer et configurer une instance chargée de représenter la voiture de son client. Ce dernier n'étant pas nécessairement fixé sur un modèle précis, ni même une marque, la configuration entreprise par le vendeur devra être complétée en cherchant à raffiner l'instance dans la hiérarchie de classes. Lors de cette exploration, le vendeur entend bénéficier des capacités de calcul de prix du concept VOITURE afin de sélectionner uniquement les modèles respectant les contraintes financières du client.

Si ce scénario semble a priori réalisable, il reste néanmoins qu'il s'appuie sur une utilisation particulière des connaissances de production. En effet, leur exploitation a ici autant pour objectif l'évaluation du prix d'une voiture que celui de critère d'appartenance à un modèle particulier. Ce rôle actif que l'on compte faire jouer à ces connaissances dans un raisonnement d'identification soulève deux problèmes fondamentaux que les deux parties suivantes se proposent de souligner.

2.2. Problème relatif au caractère des connaissances de production

Le premier problème important auquel se heurte le scénario précédent réside dans l'exploitation des résultats délivrés par les connaissances de production. En effet, l'ensemble des moyens mis en place pour obtenir le prix de la voiture est déterminant pour établir la cohérence d'une instance. Notamment, si l'un des résultats obtenus est inconsistant avec les contraintes imposées à l'instance, le traitement de l'incohérence ne consiste pas seulement à rejeter le moyen de production utilisé mais aussi à rejeter les connaissances de l'instance qui sont à l'origine de l'inférence. Ainsi, l'échec d'une politique de facturation doit se traduire par l'impossibilité pour l'instance d'appartenir au modèle de voiture qui met en place cette politique de facturation.

Or, excepté dans le cas des contraintes, il n'est pas usuel de donner aux connaissances de production un rôle actif dans l'identification des objets. En effet, dans la plupart des langages de *frames*, ce type de connaissances est implicitement doté d'un caractère par défaut. C'est-à-dire que la moindre contradiction rencontrée lors de leur application ne remet en cause que leur utilisation.

L'attribution aux connaissances de production d'un caractère différent de celui de *défaut* pourrait pourtant avoir des conséquences intéressantes sur la description des objets. Notamment, si un caractère *sûr* est envisagé pour une telle connaissance, celle-ci est alors directement impliquée dans la cohérence du contexte qui définit son emploi. Dès lors, une instance ne peut appartenir à ce contexte que si le résultat fourni par la connaissance de production est validée par l'instance. La connaissance de production opère alors comme une contrainte et ce, indépendamment du mécanisme de production employé. La partie suivante montre d'ailleurs que ce comportement est tout à fait envisageable pour une connaissance de production exprimée à l'aide d'une facette **défaut**.

Appliquée dans le cadre du concept VOITURE, la possibilité pour le concepteur de la base de fixer lui-même le caractère des connaissances de production va lui permettre d'envisager différentes interprétations de l'ensemble des voitures modélisé. Si ces connaissances possèdent un caractère *sûr*, la politique de facturation tient lieu de loi à laquelle toute instance du concept VOITURE doit se soumettre. L'ensemble des voitures modélisé correspond alors exactement à celui des voitures neuves.

Dans le cas contraire, les connaissances de production possédant un caractère de défaut, l'utilisateur a la possibilité de fixer lui-même le prix de l'instance de voiture qu'il configure. Cette possibilité permet, par exemple, de prendre en compte des instances de voiture d'occasion pour lesquelles le prix est généralement fixé au cas par cas. La gamme des prix n'étant alors plus limitée à ceux d'une voiture neuve, les connaissances de production ne jouent plus le rôle d'une contrainte dans la description des voitures et ne participent donc plus à l'établissement de l'appartenance d'une voiture à un modèle particulier.

Par la suite, seul le cas des connaissances de production possédant un caractère *sûr* sera considéré. Cette restriction permet de se focaliser essentiellement sur les problèmes liés aux interactions entre l'identification et la construction d'instance, tout en évitant les problèmes

secondaires comme l'expression et la gestion des caractères différents qui pourraient être associés aux connaissances de production.

2.3. Problème relatif aux conditions d'emploi d'une connaissance de production

Le second problème qu'il convient de souligner dans l'exploitation des connaissances de production concerne les conditions de leur emploi. Puisque le caractère qui leur est associé est *sûr*, ces connaissances participent pleinement à la cohérence des objets. Plus précisément, elles contraignent les attributs d'une instance à se conformer aux résultats qu'elles peuvent délivrer. Il est donc particulièrement important de définir clairement le contexte dans lequel doit être appliquée une connaissance de production. Cette partie se propose de montrer que le principe de masquage tel qu'il est réalisé dans les langages de *frames* s'avère dans certains cas insuffisant à définir le contexte exact d'application d'une connaissance de production.

Dans l'exemple des voitures, le cas de l'évaluation de l'attribut PrixBase constitue une illustration intéressante du principe de masquage. Du point de vue de l'héritage, la description des voitures traduit effectivement la stratégie d'évaluation souhaitée par le concepteur : "si l'instance de voiture appartient à la classe Série-Spéciale-Jeune alors la valeur de l'attribut PrixBase est 35000, sinon (sous-entendu, pour les autres séries de ce modèle) cette valeur est par héritage 50000". Toutefois, la mise en place de ce contrôle se base sur l'artifice de modélisation qu'introduit la facette **défaut** associée à l'attribut PrixBase dans la classe Modèle-M. Le recours à cette facette ne constitue ici qu'un moyen astucieux d'utiliser les priorités pour obtenir au niveau des sous-classes de Modèle-M le contrôle souhaité lors du masquage.

Le concepteur entend ainsi rendre l'utilisation de cette facette obligatoire dans les classes où elle peut être héritée et en ignorer l'existence au niveau de sa classe de déclaration. C'est-à-dire qu'il se place dans un contexte précis d'utilisation de la hiérarchie où il n'envisage même pas le rattachement, ne serait-ce que transitoire, de l'instance à la classe Modèle-M. Un tel rattachement entraînerait l'utilisation prématurée de la facette défaut et exclurait anormalement la classe Série-Spéciale-Jeune des classes de rattachement envisageables lors de la prochaine étape de raffinement.

En négligeant ainsi la description de certaines classes pour mettre en place les connaissances de production, le concepteur compromet l'emploi de la hiérarchie de classes comme support de raffinement d'instances. Pour éviter ce type de problème, il est nécessaire que les priorités établies via le masquage prennent en compte aussi bien les mécanismes d'évaluation qui sont déjà accessibles par une instance (via les descriptions des classes sûres) que ceux qui sont potentiellement accessibles par l'instance (via les descriptions des classes possibles). Dans le cas précis de la classe Modèle-M, le contrôle devrait indiquer que l'emploi de la facette de défaut est dépendante du résultat de l'appartenance de l'instance à la classe Série-Spéciale-Jeune. Le moyen d'évaluation de l'attribut PrixBase (la contrainte "PrixBase=35000"), proposé dans le contexte de rattachement de la classe Série-Spéciale-Jeune, est prioritaire sur la facette de défaut de la classe Modèle-M.

Ce constat suggère une révision du contrôle de l'évaluation d'un attribut afin de l'adapter aux évolutions possibles d'une instance dans la hiérarchie de classes. Dans cette optique, les trois parties suivantes sont consacrées à la définition d'un modèle des connaissances de production induites par une description d'objets. Ce modèle est proposé comme support de contrôle des évaluations d'attributs du modèle TROPES.

3. Modélisation des connaissances de production

La mise en place du contrôle repose sur une modélisation des connaissances de production présentes dans la description des objets. Un élément de description est reconnu comme pouvant être une connaissance de production dès qu'il peut être à l'origine d'une inférence de valeur d'un attribut.

La notion de connaissance de production résulte donc d'une interprétation particulière de l'ensemble des connaissances contenues dans la description d'un concept. Cette interprétation repose sur le fait que toute situation de production de valeurs d'attributs est prédéterminée par la description des objets. Chacune de ces situations dénote la présence d'une des connaissances de production associée à un attribut.

Dans le modèle proposé ici, la notion de connaissance de production se concrétise par l'introduction dans la description des objets du concept de *déclencheur d'inférence*. Un attribut se voit associer autant de *déclencheurs* qu'il existe de situations différentes de production de valeur dans la description d'objets. Un *déclencheur d'inférence* regroupe les informations précisant le type de mécanisme d'inférence à utiliser et les conditions que doit réunir l'instance pour le déclencher.

Dans un premier temps, cette partie décrit quels sont les différents types de mécanismes d'inférence auxquels peut prétendre un attribut. Ces types, communs à tous les attributs, sont appelés les *sources de connaissances* des attributs. Dans un second temps, la notion de *déclencheur d'inférence* est introduite pour représenter les différentes connaissances de production associées à un attribut dans la description des objets. La modélisation globale des connaissances de production d'un attribut est enfin complétée en prenant en compte la notion d'ordre de priorité.

3.1. Sources de connaissances possibles pour un attribut

Il s'agit ici de répondre à une question simple : "par quels types de mécanisme un attribut d'une instance peut-il être évalué ?". L'ensemble de ces mécanismes, appelés *sources de connaissances*, peut être organisé en deux catégories :

- **Sources internes à la description de l'objet à configurer** :
  - le détachement procédural,
  - le domaine de valeurs de l'attribut qui subit des réductions dues au typage et/ou aux contraintes,
  - toute facette de défaut associée à l'attribut dans la description des objets.
- **Sources externes à la description de l'objet à configurer** :
  - le ou les utilisateurs,
  - toute instance en relation avec l'instance concernée qui partage avec elle des valeurs d'attributs (Ex. : des attributs d'un objet composite peuvent permettre l'évaluation d'un attribut d'un objet composant).

Toutefois, par la suite, **toute source externe est ramenée à un cas de source interne**. En effet, par souci d'uniformisation et de simplification, tout ajout à une instance de connaissance d'origine externe sera réalisé par la pose de nouvelles contraintes.

Les partages de valeurs d'attributs entre objets sont déjà ainsi traités à travers les descriptions des classes comportant des attributs de type *complexe* (leur domaine de valeurs est un ensemble d'instances). En effet, le partage de valeurs entre attributs d'objets distincts est mis en place grâce aux contraintes de classe et à la notion de chemin. On rappelle qu'un chemin permet l'accès aux attributs des objets référencés par les attributs complexes (cf. §VI.2.1.1).

L'exemple des voitures fournit deux situations où il peut y avoir partage de valeurs d'attributs entre un objet du concept VOITURE et un objet du concept MOTEUR :

- dans la classe VOITURE : (*Voiture.*)type = **Moteur**.type
- dans la classe Marque-N : (*Voiture.*)PrixHT = **Moteur**.Prix + (*Voiture.*)PrixBase

Il est important de noter cependant que :

- si ces contraintes permettent l'évaluation d'attributs de l'instance voiture, elles constituent alors une source de connaissances **interne** à la description de la voiture,

- tandis que si elles permettent l'évaluation d'attributs de l'instance de moteur, elles constituent une source de connaissances **externe** à la description du moteur.

En effet, ces contraintes sont propres à la description du concept VOITURE et sont complètement indépendantes du concept MOTEUR. Pour l'instance de moteur, ces contraintes proviennent donc du contexte de son utilisation et non de sa définition.

En ce qui concerne l'utilisateur, le principe est de restreindre l'utilisateur à l'emploi des contraintes pour exprimer les connaissances qu'il possède sur les attributs d'une instance. C'est donc par l'intermédiaire du mécanisme de pose de contraintes dynamiques sur une instance, offert par le module MICRO, que l'utilisateur communique les informations qu'il possède sur les attributs (cf. §VI.2).

Ainsi, l'utilisateur n'*évalue* pas un attribut, il le *contraint*. Lorsqu'il veut évaluer un attribut, il lui suffit de poser une contrainte d'égalité entre cet attribut et la valeur choisie. L'intérêt de ce principe est de ne pas accorder à l'utilisateur un rôle particulier qui le privilégierait vis-à-vis de l'évaluation des attributs et qui lui permettrait donc d'échapper aux contraintes stratégiques imposées par le contrôle d'évaluation d'attributs.

Dans le cadre général de ce contrôle, il convient effectivement de préciser que l'affectation d'une valeur à un attribut passe nécessairement par la désignation préliminaire de la source de connaissances à consulter. Par exemple, la réduction d'un domaine de valeurs à un singleton ne constitue pas une condition suffisante pour déclencher une inférence par contrainte, encore faut-il que le domaine soit la source de connaissances de l'attribut désignée par le contrôle. Cette mesure donne donc lieu à une révision du mécanisme d'inférence par contraintes proposé initialement par le module MICRO (cf. §VI.2). Les modalités du nouveau couplage entre le contrôle d'évaluation d'un attribut et le module de contraintes MICRO seront détaillées par la suite (cf. §VII.4.3.1.3).

De façon générale, quelque soit la source de connaissances impliquée, le déclenchement d'une inférence est une décision qui dépend des relations établies par la description des objets entre l'état d'une instance et les sources de connaissances possibles d'un attribut. Ces relations sont matérialisées et représentées par la notion de *déclencheur d'inférence*.

3.2. Déclencheur d'inférence d'attribut

Trois sources de connaissances sont donc envisageables pour l'évaluation d'un attribut : le **détachement procédural**, la facette **défaut** et le **domaine**.

Les différentes situations dans lesquelles une source de connaissances **doit** être utilisée sont fixées par la description du concept. Leur identification est réalisée par le biais de la notion de *déclencheur d'inférence*. Cette notion, relative à un attribut, a pour but d'indiquer d'une part les conditions nécessaires et suffisantes au déclenchement d'une inférence et d'autre part la source de connaissances qui a en charge cette inférence. Un déclencheur se définit donc par :

- **Les conditions nécessaires et suffisantes de déclenchement**, qui sont représentées par deux informations :
  - **l'accès** : spécifié par une référence à une classe du concept qui indique la classe la plus générale à laquelle doit appartenir une instance pour avoir accès à la connaissance de production représentée par ce déclencheur.
  - **les connaissances de production prioritaires** : dénotées par un ensemble de déclencheurs d'inférence. Ces connaissances de production doivent avoir été essayées avant celle représentée par ce déclencheur. Cette condition se base sur l'existence de priorités sur les connaissances de production. La partie suivante lui est consacrée.
- **La source de connaissances à consulter**, qui est signifiée par l'un des mots clefs suivants : **Dom** pour Domaine, **DP** pour Détachement Procédural et **Def** pour Défaut.

L'interprétation d'un *déclencheur d'inférence* est simple : si les conditions sont réunies, l'instance appartient à la classe de référence et toute connaissance de production prioritaire a déjà été écartée, le système peut déclencher l'évaluation de l'attribut en se référant à la source de connaissances indiquée. Les principes de ce fonctionnement sont détaillés par la suite (cf. §VII.4).

En faisant abstraction des informations qui ont trait aux priorités, les déclencheurs associés à un attribut dans un concept sont obtenues à partir des règles suivantes :

- un déclencheur est créé spécialement pour prendre en compte les propositions de l'**utilisateur** :
  - La condition d'accès minimale est définie par la **classe racine** du concept puisque l'utilisateur est en mesure de faire ses propositions dès la création de l'instance dans le concept (le rattachement à la classe racine est alors nécessairement réalisé).
  - La source de connaissances est fixée à **Dom** (domaine) puisque l'utilisateur configure les attributs d'une instance par le biais des contraintes (cf. §VII.3.1).
- un déclencheur est créé pour représenter le détachement procédural quand il est défini dans la description de l'attribut au niveau du concept. La source de connaissance est alors fixée à **DP** et sa condition d'accès est, à l'instar de l'utilisateur, dénotée par la **classe racine** du concept.
- un déclencheur est créé pour chaque classe dans laquelle le domaine de valeur de l'attribut est raffiné, que ce soit par les facettes de typage ou par son implication dans une contrainte de la classe. La condition d'accès correspond à la classe concernée et la source de connaissances est fixée à **Dom**. Cette règle n'est pas appliquée si la classe est la racine puisqu'un tel déclencheur à déjà été créé pour prendre en compte l'utilisateur.
- un déclencheur est créé pour chaque classe où apparaît une facette **défaut** associée à l'attribut. La classe constitue alors la condition d'accès du déclencheur tandis que sa source de connaissances est fixée à **Def** (Défaut).

L'ensemble des déclencheurs ainsi obtenu pour chaque attribut est associé à sa description au niveau du concept. Par exemple, l'attribut PrixBase du concept VOITURE se verra associer l'ensemble suivant :

Dℓ(PrixBase)= {	(VOITURE, Dom);	=>	dû à l'utilisateur
	(Marque-N, Dom);	=>	dû au raffinement de la classe
	(Modèle-M, Def);	=>	dû à la facette défaut de la classe
	(Série-Spéciale-Jeune, Dom) }	=>	dû au raffinement de la classe

où chaque couple (Nom de classe, Source de connaissances) est un déclencheur qui ne tient pas compte des priorités. Dans cet exemple, l'attribut PrixBase ne possède pas de détachement procédural.

Pour compléter la description d'un tel ensemble, il faut de plus prendre en compte l'ordonnancement des déclenchements qui apparaît lorsque des priorités peuvent être établies entre les connaissances de production. Ce cas est étudié dans la partie suivante.

3.3. Ordonnancement des déclenchements d'inférence

Dans les langages de *frames*, l'accès aux facettes d'obtention de valeur obéit à un ordre de priorité fixé. Cet ordre est appliqué pour ordonnancer dynamiquement les tentatives d'inférences auxquelles peut prétendre un attribut à travers le graphe d'héritage. Il résulte de la combinaison de deux ordres : l'un est établi en fonction du type des facettes et l'autre est relatif aux niveaux de spécialisation auxquels apparaissent les facettes.

La priorité couramment appliquée sur les types de facettes consiste à favoriser l'emploi de la facette **$valeur**, puis de la facette **$si-besoin** et enfin de la facette **$defaut** [Napo92]. Quant à la priorité relative à la spécialisation, elle indique dans quel ordre appliquer deux

facettes apparaissant dans des classes comparables par la relation de spécialisation. Cet ordre favorise la facette de la classe la plus spécialisée si les deux facettes sont de même type. Si elles sont de types différents, l'établissement d'une priorité entre elles dépendra de la façon de combiner cet ordre avec celui sur les types de facettes.

Les priorités constituent une information que le concepteur de la base de connaissances doit nécessairement connaître s'il veut définir correctement la stratégie d'évaluation de chaque attribut. A cet égard, la solution permettant d'indiquer explicitement le type de priorité à appliquer selon chaque cas apparaissant dans la description des objets, offre l'avantage d'impliquer directement le concepteur dans l'expression de la stratégie d'évaluation des attributs. Par ailleurs, elle permet une description plus souple des objets.

Toutefois, dans les langages de *frames*, la validité des priorités est directement dépendante de la classe de rattachement de l'instance. En effet, les priorités constituant un paramètre du mécanisme d'héritage, seules les connaissances de production définies entre la classe de rattachement courant d'une instance et l'ensemble de ses sur-classes sont prises en compte. Comme le souligne l'exemple de la classe *Modèle-N* (cf. §VII.2.3), cette solution n'est acceptable que dans la mesure où la classe de rattachement permet d'hériter un ensemble de connaissances de production représentatif de la stratégie d'évaluation qu'il faut appliquer à l'attribut dans le contexte courant.

Dans le modèle TROPES, la solution adoptée consiste à dégager la notion de priorité du cadre trop restreint de l'héritage pour en faire un réel élément de description des objets permettant d'exprimer la stratégie d'évaluation en fonction de l'ensemble des évolutions possibles d'une instance dans la hiérarchie de classes. De ce fait, le développement de la stratégie d'évaluation d'un attribut est dépendante du processus de raffinement de l'instance dans la hiérarchie de spécialisation.

Techniquement, le principe consiste à permettre au concepteur de définir l'ordre de priorité qu'il souhaite pour chaque attribut. Cet ordre est ensuite appliqué par le système à l'ensemble des déclencheurs d'inférence de l'attribut qui ont été recensés. Cet ensemble étant ainsi ordonné statiquement et défini au niveau du concept, toute instance peut s'y référer pour contrôler l'évaluation de l'attribut. Sur cette base, le contrôle peut établir s'il y a lieu ou non de déclencher une inférence en fonction de la classe de rattachement de l'instance.

A titre indicatif, deux grandes tendances de définition de priorités sont préalablement commentées afin de montrer comment le concepteur peut utiliser les priorités pour accorder à deux attributs distincts des comportement fortement différents dans la même hiérarchie de classes. Ces deux tendances diffèrent selon que le concepteur souhaite favoriser l'évaluation de l'attribut :

a) par les connaissances de production les plus générales.

   Il s'agit dans ce cas de donner plus d'importance aux sources de connaissances qui sont accessibles globalement comme le détachement procédural, l'utilisateur ou une instance en relation avec l'instance courante.

   Dans l'exemple du concept VOITURE, le détachement procédural doit être prioritaire sur les autres sources de connaissances pour l'attribut PrixTTC. Le déclencheur le représentant sera donc prioritaire et appliqué dès que cet attribut apparaît dans une instance. Dans ce cas, l'appartenance inconditionnelle de toute instance à la classe racine VOITURE entraînant nécessairement l'introduction de l'attribut PrixTTC, le déclenchement de son détachement procédural sera réalisé dès la création d'une instance.

   Par ailleurs, si l'on suppose l'existence d'un attribut Couleur dans la description d'une voiture, il paraît normal que son évaluation soit plutôt du ressort de l'utilisateur que fixé par le modèle précis de la voiture. Etant donné que l'utilisateur est modélisé par un déclencheur dont la source de connaissance est le domaine et dont l'accès est défini par la classe racine, il suffit de donner la priorité aux déclencheurs de type domaine dont l'accès est le moins spécialisé. Ainsi, si l'utilisateur contraint l'attribut à une valeur, cette dernière lui est affectée immédiatement.

Il est à noter que si l'utilisateur ne pose pas une telle contrainte, l'application du déclencheur qui lui correspond aboutit sur un échec. L'utilisateur est alors écarté des sources de connaissances possibles pour l'attribut. Les déclencheurs de moindre priorité peuvent alors être employés pour évaluer l'attribut. Par exemple, si le concept des voitures possède une classe représentant un modèle de Ferrari, dans laquelle une facette de défaut associe la valeur "rouge" à l'attribut Couleur, le rattachement d'une instance à cette classe permettra d'inférer que la valeur de l'attribut Couleur est rouge si l'utilisateur n'a pas daigné la spécifier lui-même.

b) par les connaissances de production les plus spécialisées.

Il s'agit dans ce cas d'un attribut dont l'évaluation est fortement dépendante des possibilités de raffinement de l'instance. La connaissance de production la plus prioritaire est celle dont l'accès est défini par la classe la plus spécialisée. Cette priorité favorise donc surtout les sources de connaissances de type domaine et défaut. L'évaluation de l'attribut sera possible quand l'instance sera rattachée à une classe suffisamment spécialisée.

L'attribut PrixBase du concept VOITURE illustre ce genre de priorité. La contrainte d'égalité de la classe Série-Spéciale-Jeune est prioritaire sur les autres moyens d'évaluation. Ensuite, c'est la facette défaut de la classe Modèle-M qui prime sur la contrainte impliquant PrixHT et Moteur.Prix de la classe Marque-N. La priorité est établie ainsi de suite en remontant dans la hiérarchie de classes jusqu'aux déclencheurs correspondant à l'utilisateur et au détachement procédural.

L'importance et l'originalité des priorités relevant de cette catégorie proviennent de la possibilité de différer l'évaluation de l'attribut tant que le raffinement de l'instance n'a pas permis d'atteindre ou d'écarter la connaissance de production la plus spécialisée.

Cette partie présente, d'une part, comment peut être définie la priorité associée à un attribut (cf. §VII.3.3.1) et, d'autre part, comment elle est appliquée pour ordonner les déclencheurs d'inférences de l'attribut (cf. §VII.3.3.2).

3.3.1. Définir un ordre de priorité

Pour définir l'ordre de priorité d'un attribut, le concepteur a la possibilité de fixer un ensemble de **règles de priorité**. Chacune d'elles est chargée d'indiquer les cas dans lesquels les connaissances de production appartenant à une certaine source sont prioritaires sur les connaissance de production appartenant à une autre source. Une règle de priorité obéit à la syntaxe suivante :

<Règle-Précéd> ::= <Type-Source> (<Ens-Rel-Spéc>) <Type-Source> ;
<Type-Source> ::= **Dom** | **DP** | **Def** ;
<Ens-Rel-Spéc> ::= < | = | >
| <Ens-Rel-Spéc>, <Ens-Rel-Spéc> ;

L'expression S_1 **(r)** S_2 a pour signification :

> "Une connaissance de production de source S_1, dont la classe d'accès est C_1, est prioritaire sur une connaissance de production de source S_2, dont la classe d'accès est C_2, si la relation (C_1 **r** C_2) est vérifiée."

La relation **r** entre les deux classes C_1 et C_2 est exprimée en fonction de la relation de spécialisation "<", de son inverse ">" (la généralisation) et/ou de l'égalité de classe "=". Les différentes combinaisons pouvant apparaître dans l'expression **(r)**, ainsi que leur interprétation respective, sont données dans le tableau suivant :

(r)	C_1 r C_2
(<)	"C_1 est sous-classe de C_2" (spécialisation)
(=)	"C_1 est la même classe que C_2" (égalité de classe)
(>)	"C_1 est sur-classe de C_2" (généralisation)
(<, =)	équivalent au OU logique des deux premiers cas
(=, >)	équivalent au OU logique du deuxième et troisième cas
(<,>)	équivalent au OU logique du premier et troisième cas
(<, =, >)	équivalent au OU logique des trois premiers cas

Il est à noter qu'une règle de priorité de la forme S_1 (r) S_2, où (r) est l'une des formes combinées (<, =), (=, >), (<,>) ou (<, =, >), est équivalente à la donnée d'un ensemble de règles de la forme S_1 (x) S_2 où chaque x correspond à l'un des symboles <, > ou =, apparaissant dans (r). On dira dans ce cas que la règle S_1 (r) S_2 est une forme complexe et que les règles S_1 (x) S_2 sont des formes simples.

L'exemple suivant décrit l'équivalence entre une règle sous forme complexe et un ensemble de règles sous forme simple :

- S_1 (<, =) $S_2 \Leftrightarrow \{S_1 (<) S_2; S_1 (=) S_2\}$

Le tableau ci-dessous propose un ensemble non-exhaustif de règles de priorité, et donne pour chacune d'elles sa signification.

Dom (<, =) Def	toute connaissance de production de type **domaine** est prioritaire sur toute connaissance de production de type **défaut** si elle apparaît dans la même classe ou dans une classe plus spécialisée.
Dom (>) Dom	toute connaissance de production de type **domaine** est prioritaire sur une connaissance du même type si elle apparaît dans une classe moins spécialisée. Cette priorité permet de rendre prioritaires les contraintes globales posées par l'**utilisateur** ou par une **instance en relation avec l'instance courante**.
Dom (<, =, >) Def	toute connaissance de production de type **domaine** est prioritaire sur toute connaissance de production de type **défaut** quelque soit leur niveau de spécialisation respectif.
DP (=, >) Dom	Le détachement procédural est prioritaire sur toute connaissance de production de type domaine.

Grâce à ces règles, le concepteur peut indiquer pour chaque attribut les priorités à employer pour entreprendre son évaluation. Pour ce faire, au même titre qu'il spécifie dans la déclaration du concept le **type**, le **constructeur** et la **nature** d'un attribut (cf. §II.4.1.1), le concepteur indique la **priorité** à appliquer à cet attribut en donnant explicitement un ensemble de règles de priorité. Par exemple, il pourrait déclarer la priorité suivante :

priorité={ DP (=) Dom;
Dom (<) DP;
Dom (<, =) Def;
Dom (<) Dom;
Def (<) DP;
Def (<) Dom;
Def (<) Def }

Dans cet exemple, la première règle de priorité permet de rendre prioritaire le détachement procédural sur les inférences d'origine externe comme l'utilisateur (contraintes posées par l'utilisateur et prenant effet dès la classe racine). Dans les autres cas, les inférences provenant des niveaux les plus raffinés (facettes défaut et réductions de domaine) sont prioritaires. Les réductions de domaine priment sur les facettes de défaut si elles apparaissent dans la même classe ou à des niveaux plus spécialisés. Sinon, les facettes défaut priment sur les inférences de type domaine.

Cet exemple tend à confiner l'évaluation de l'attribut aux possibilités d'inférence qu'offre la description des objets. Comme cette priorité favorise les inférences provenant des classes les plus spécialisées, elle établit une dépendance très forte entre le raffinement de l'instance et l'évaluation de l'attribut auquel cette priorité s'applique. Dans ce sens, une telle priorité conviendrait parfaitement au cas des attributs PrixBase et PrixHT du concept VOITURE.

3.3.2. Priorité par défaut

En l'absence d'une priorité définie par le concepteur, un attribut a la priorité par défaut suivante :

priorité_par_défaut={
Dom (=) DP;
DP (<) Dom;
DP (=, >) Def;
Dom (>) Dom;
Dom (<, =, >) Def;
Def (<) Def }

Cette priorité favorise les inférences possibles au plus haut niveau de raffinement. Les inférences d'origine externe (notamment, l'utilisateur) sont prioritaires sur toutes les autres. Le détachement procédural prime ensuite, puis les réductions de domaine et enfin les facettes défaut. La facette défaut la plus spécialisée est prioritaire sur les autres facettes défaut ; la situation contraire, bien qu'elle puisse être définie, n'ayant conceptuellement que peu d'intérêt.

Cette priorité est typiquement celle qui serait associée à l'attribut Couleur du concept VOITURE, et à tous les autres attributs du même acabit dont la valeur est normalement fixée en fonction des desiderata de l'utilisateur.

3.3.3. Application d'une priorité

Une fois que la priorité d'un attribut est définie, par le concepteur ou par défaut, elle est appliquée à l'ensemble de déclencheurs d'inférences associés à cet attribut. Cette application consiste à ordonner l'ensemble des déclencheurs des plus prioritaires aux moins prioritaires. La relation d'ordre strict, notée $\rightarrow$, sur les déclencheurs est obtenue à partir d'une priorité de la façon suivante :

Soient $d_1=(C_1, S_1)$ et $d_2=(C_2, S_2)$ deux éléments de $\mathfrak{D}\mathfrak{l}(A)$, l'ensemble des déclencheurs associés à l'attribut A. Soit $P(A)= \{R_i\}_{1\leq i\leq n}$ la priorité associée à A où R_i représente une règle de priorité de P(A). La relation $(d_1 \rightarrow d_2)$ est vérifiée si et seulement si l'une des conditions (1) ou (2) l'est :

(1) $\exists\ R \in P(A)$, $R=(S_1\ (r)\ S_2)$ et $(C_1\ r\ C_2)$

(2) $\exists\ d_3 \in \mathfrak{D}\mathfrak{l}(A)$, $(d_1 \rightarrow d_3)$ et $(d_3 \rightarrow d_2)$

L'ordre obtenu sur l'ensemble des déclencheurs est généralement partiel. En effet, s'il existe deux déclencheurs d'un même attribut dont les classes d'accès respectives n'entretiennent aucune relation de spécialisation, ces déclencheurs ne sont pas comparables par la relation →.

Afin d'illustrer l'impact d'une priorité sur la stratégie d'évaluation d'un attribut, l'exemple de l'attribut PrixBase est repris. Dans cette reprise, la description des objets est étendue par l'ajout de la classe *Série-Spéciale-Luxe*, contraignant l'attribut PrixBase à 100000. Cette mesure permet d'introduire dans l'exemple le cas de deux déclencheurs incomparables par →, ceux provenant des deux classes sœurs *Série-Spéciale-Jeune* et *Série-Spéciale-Luxe*.

La figure VII.2 présente la phase d'obtention des déclencheurs d'inférence induits par la nouvelle description du concept VOITURE.

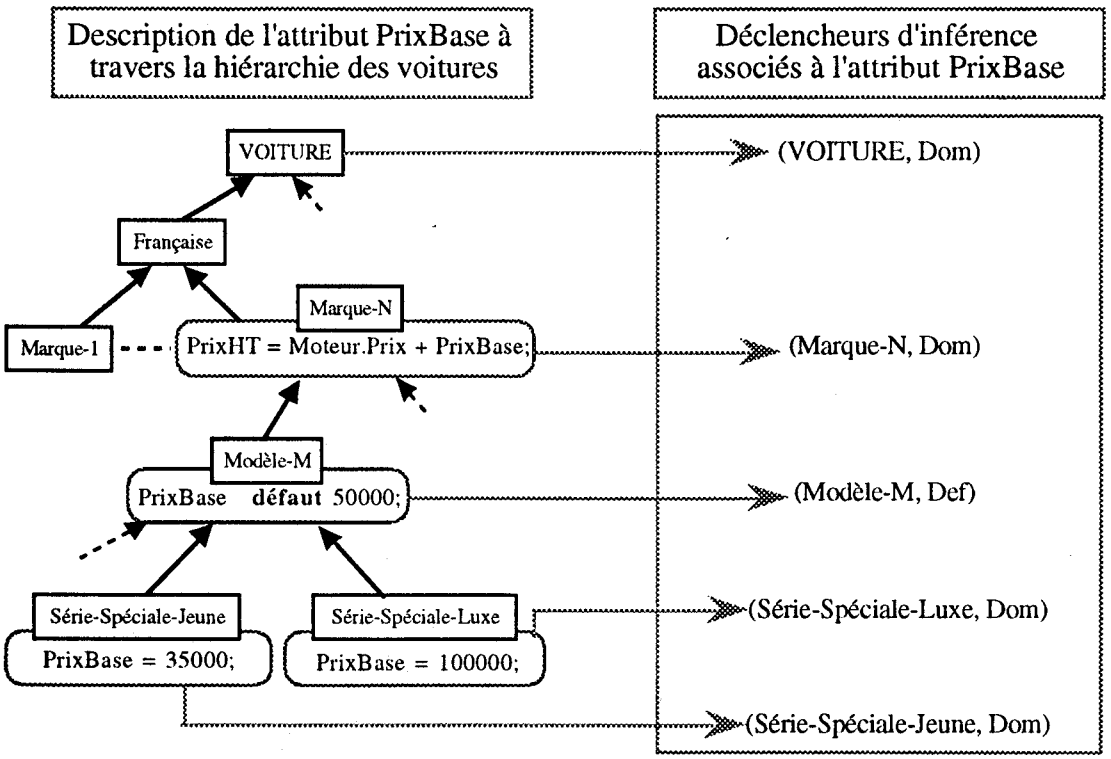

Figure VII.2 : Obtention des déclencheurs d'inférence associés à l'attribut PrixBase en fonction de sa description dans le concept VOITURE.

Sur l'ensemble des déclencheurs d'inférence de l'attribut PrixBase, deux priorités différentes sont envisagées (cf. Figure VII.3). Ces deux priorités permettent de résoudre le problème que rencontrait le masquage lors du traitement de la facette défaut de la classe *Modèle-M* (cf. §VII.2.3). La solution est la même pour les deux priorités, elle consiste à restreindre l'emploi de cette facette à un contexte dans lequel toute possibilité d'inférer l'attribut PrixBase par les contraintes de la classe *Série-Spéciale-Jeune*, ou de la classe *Série-Spéciale-Luxe*, est écartée. Ce qui signifie, en l'occurrence, qu'il est nécessaire que ces deux classes aient été préalablement déclarées impossibles.

Ces deux priorités se distinguent sur la prise en compte des informations que peut apporter l'utilisateur. En effet, la première priorité permet à l'utilisateur d'évaluer l'attribut PrixBase avant même que soit entrepris le raffinement de l'instance dans la hiérarchie, tandis que la deuxième priorité n'envisage la prise en compte des propositions de l'utilisateur qu'à partir du moment où la classe *Modèle-M* est déclarée impossible.

Le rôle de la facette défaut de la classe *Modèle-M* est très différent selon la priorité choisie. La première priorité est sur ce point moins contraignante puisque, l'utilisateur étant prioritaire, la facette défaut est ignorée si l'attribut PrixBase est déjà évalué lors du rattachement

d'une instance à la classe *Modèle-M*. Par contre, la deuxième priorité ne permet pas que l'attribut PrixBase d'une instance appartenant à la classe *modèle-M* puisse être différent soit de l'une des deux valeurs imposées par les classes *Série-Spéciale-Jeune* et *Série-Spéciale-Luxe*, soit de la valeur délivrée par la facette défaut de *Modèle-M*.

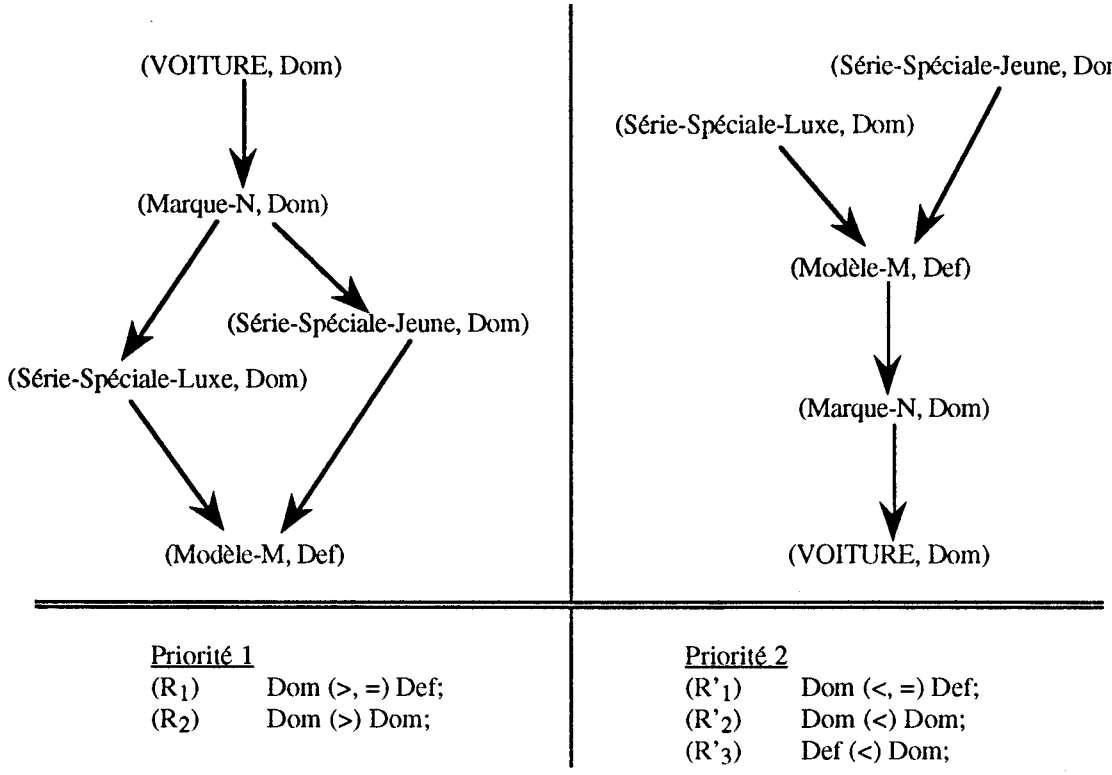

Figure VII.3 : Résultats de l'ordonnancement de l'ensemble des déclencheurs d'inférence de l'attribut PrixBase selon deux priorités différentes. La priorité 1) favorise en premier l'évaluation de PrixBase par l'utilisateur, en second l'évaluation par les contraintes soit de la classe *Série-Spéciale-Luxe*, soit de la classe *Série-Spéciale-Jeune*, et en dernier recours l'évaluation par la facette défaut de la classe *Modèle-M*. La priorité 2) se distingue de la priorité 1) en reléguant l'utilisateur en dernière position.

La cohérence d'une priorité est vérifiée par le système afin de garantir que l'ensemble de ses règles soit pertinent et induise bien une relation d'ordre stricte sur les déclencheurs. Le principe de cette vérification est rapidement présenté dans la partie suivante.

3.3.4. Propriétés et cohérence d'une priorité

La possibilité de définir ses propres règles de priorité est certes une liberté que le concepteur appréciera, mais il peut toutefois lui paraître difficile d'établir la pertinence des priorités qui en résultent. Un mécanisme de vérification de la cohérence d'une priorité est donc mis en place pour assister le concepteur lors de la définition des priorités.

La priorité est chargée d'établir un ordonnancement des connaissances de production associées à un attribut, il est donc primordial de s'assurer que les règles de priorité qui la composent décrivent bien une relation d'ordre strict. C'est-à-dire qu'il faut garantir que la relation $\rightarrow$ soit transitive et anti-symétrique. La transitivité est inhérente à la construction même de cette relation. Par contre, l'anti-symétrie n'est pas assurée par définition et dépend des règles définies par le concepteur.

Vérifier la cohérence de la priorité revient donc à vérifier que l'application de la priorité ne compromet pas l'antisymétrie de la relation $\rightarrow$. La vérification[1] est réalisée lors de l'application de la priorité et consiste à détecter la présence éventuelle d'un couple de déclencheurs pour lesquels la priorité est établie dans les deux sens. Le cas échéant, la priorité est rejetée.

Par exemple, l'application des règles suivantes R_0=(Def (>) Dom), R_1=(Dom (<) DP) et R_2=(DP (>) Def) sur les déclencheurs d_0=(C_0, DP), d_1=(C_1, Def), et d_2=(C_2, Dom), sachant que $C_0 > C_1 > C_2$, mène à une incohérence. En effet, l'application des règles donne :

1)	$d_0 \rightarrow d_1$	(R_2)
2)	$d_1 \rightarrow d_2$	(R_1)
3)	$d_0 \rightarrow d_2$	(Transitivité 1 et 2)
4)	$d_2 \rightarrow d_0$	(R_0)
5)	**Incohérence**	(Contradiction 3 et 4)

Pour conclure sur les priorités, il faut noter que l'obtention d'un ensemble ordonné de déclencheurs d'inférence pour chaque attribut constitue un contexte adéquat pour la mise en place d'un mécanisme de consultation, ou de visualisation, des stratégies d'évaluation. Un tel mécanisme fournirait un support d'aide appréciable lors de la mise au point de la base de connaissances.

Après cette présentation du modèle des connaissances de production basé sur la notion de déclencheur d'inférence, la partie suivante se propose d'en décrire l'exploitation dans la mise en place du contrôle d'évaluation des attributs d'une instance.

4. Mise en place du contrôle d'évaluation

Le contrôle d'évaluation des attributs d'une instance repose sur le principe suivant : chaque étape de raffinement de l'instance apporte sa propre contribution à la construction de l'instance. Le rôle du contrôle est donc de préciser en termes d'évaluation d'attributs la nature de cette contribution. Il doit, pour ce faire, assurer le développement de la stratégie d'évaluation de chaque attribut au rythme du raffinement de l'instance dans la hiérarchie de spécialisation.

Toute la difficulté d'un tel contrôle réside dans la gestion de concert de tous les attributs apparaissant dans une instance. La solution adoptée consiste à associer à chaque attribut un état qui caractérise l'étape d'évaluation dans laquelle il se trouve. Dès lors, le contrôle d'évaluation de chaque attribut doit assurer les différentes transitions d'états de chaque attribut (cf. §VII.4.1).

Ces transitions traduisent l'alternance de deux phases du contrôle : la sélection d'une inférence à déclencher et l'exécution d'une inférence. La sélection d'une inférence repose sur l'exploitation de l'ensemble des déclencheurs associé à un attribut. En effet, cet ensemble sert de support au contrôle pour, d'une part, suivre les développements de la stratégie en fonction du raffinement de l'instance et des échecs d'inférence et, d'autre part, établir en fonction de cet état les inférences qui prétendent en priorité au déclenchement (cf. §VII.4.2). Lorsqu'un attribut fait l'objet d'un déclenchement d'inférence, le contrôle doit assurer le bon déroulement de son exécution et, à l'instar du fonctionnement du détachement procédural, assurer la gestion d'éventuelles phases d'attente et de reprise de l'inférence (cf. 4.3).

4.1. Raffinement d'instance et évaluation des attributs

Le cadre dans lequel prend place le contrôle d'évaluation des attributs est celui du raffinement d'une instance. C'est-à-dire que le déclenchement et l'exécution des inférences s'organisent autour d'un apport progressif d'informations guidé par la classification de l'instance à travers la hiérarchie de spécialisation.

---

[1]La vérification de la cohérence d'une priorité peut aussi être établie en dehors de son contexte d'application. Dans ce cas, on cherche à vérifier que les règles de précédence ne peuvent compromettre l'anti-symétrie de $\rightarrow$ quelque soit l'ensemble de déclencheurs considéré. La cohérence est alors une propriété intrinsèque de la priorité. Cette solution, bien que plus pertinente, n'est pas présentée par souci de simplification.

4.1.1. Mise à jour et raffinement d'une instance

La procédure de mise à jour est l'opération par laquelle de nouvelles informations sont apportées à l'instance. Les informations qui peuvent être ainsi ajoutées à une instance sont la déclaration de nouveaux attributs, la pose de nouvelles contraintes sur les attributs, la déclaration de l'appartenance ou de la non-appartenance à une classe de la hiérarchie. Le rôle essentiel de la procédure de mise à jour consiste à assumer les conséquences de ces nouvelles informations et, plus particulièrement, leurs impacts sur l'état de chaque attribut de l'instance.

L'ordre de traitement des informations est primordial pour le contrôle d'évaluation des attributs. En effet, la stratégie d'évaluation d'un attribut étant fortement liée au raffinement de l'instance, il est nécessaire que toute mise à jour de l'instance soit traitée dans le respect de l'ordre imposé par la démarche classificatoire. Aussi, la procédure de mise à jour adopte les mesures suivantes :

1) L'apparition d'un nouvel attribut au sein de l'instance est traitée dans un contexte ne tenant compte que des contraintes imposées par l'utilisateur. Cette mesure signifie que la mise à jour traite en priorité l'intégration de nouveaux attributs au fur et à mesure de leur apparition. Celle-ci peut avoir lieu lors de la configuration initiale, du rattachement de l'instance à une classe ou de la demande de valeurs d'entrée par une inférence. Après avoir appliqué les contraintes de l'utilisateur, la procédure de mise à jour passe la main au contrôle d'évaluation pour chacun des nouveaux attributs.

2) Le rattachement de l'instance à une classe passe par le traitement préalable du rattachement de l'instance à chacune de ses sur-classes, en commençant par la classe racine. Dans une démarche incrémentale, cela signifie que la déclaration d'appartenance de l'instance à une classe entraîne le traitement préalable du rattachement de l'instance à toutes les classes se trouvant successivement entre la classe de rattachement courant (dénotée par le lien **est-un**) et la classe cible.

La configuration d'un attribut par l'utilisateur est restreinte[2] à la phase d'apparition de l'attribut dans l'instance. Toute pose ultérieure de contraintes sur un attribut sera considérée comme une tentative de modification de l'instance puisqu'elle peut remettre en cause l'état d'évaluation de l'attribut, voire même la classe de rattachement de l'instance.

De même, toute déclaration d'appartenance contredisant le rattachement courant est considérée comme une modification de l'instance. Le traitement des modifications sort du cadre du contrôle d'évaluation et ne sera donc pas pris en compte par la suite.

4.1.2. Organisation du contrôle d'évaluation d'un attribut

Etant donné l'ordre des mises à jour imposé à l'instance par le processus de raffinement, le contrôle organise l'évaluation de chaque attribut en alternant phase de sélection de la prochaine inférence à déclencher et phase d'exécution de l'inférence ainsi sélectionnée. Tant qu'aucune inférence ne permet l'évaluation de l'attribut, la transition d'une phase à l'autre est dépendante de l'état de raffinement de l'instance. Le fonctionnement du contrôle peut être schématisé par le graphe de transitions d'états suivant (cf. Figure VII.4).

---

[2]Cette restriction a pour objectif de simplifier la présentation du contrôle d'attribut en garantissant que la configuration de l'instance par l'utilisateur est une opération monotone. Il est toutefois possible de lever cette restriction en mettant en place une procédure de vérification chargée d'établir si une intervention de l'utilisateur remet en cause le contrôle d'évaluation des attributs. Dans ce cas, l'utilisateur pourrait apporter des informations sur l'instance à n'importe quelle étape de son raffinement

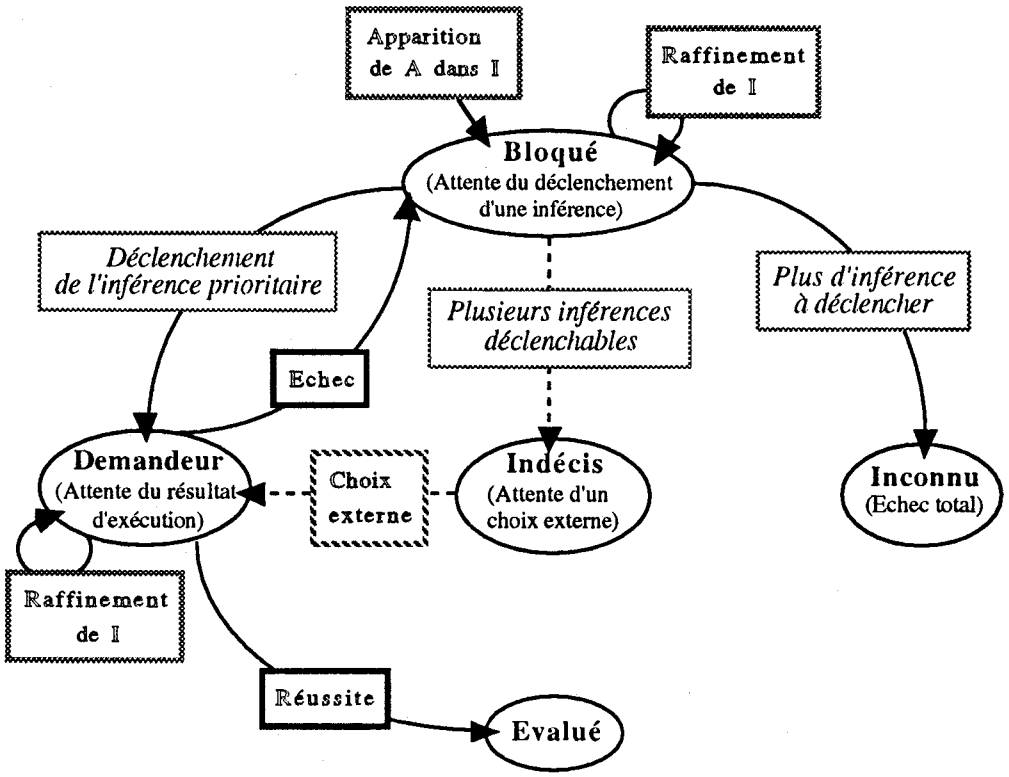

Figure VII.4 : Organisation du contrôle d'évaluation d'un attribut en fonction du raffinement d'une instance. Le contrôle permet la transition de l'attribut dans cinq états (bulles) suivant l'apparition d'événements liés directement au raffinement de l'instance (cadres gris épais), liés au résultat d'exécution d'une inférence (cadres noirs épais), liés à la sélection d'une inférence (cadre gris fins). L'état **Indécis** traduit le cas particulier d'un choix non-déterministe d'inférences provoqué par une priorité insuffisamment sélective. Pour résoudre ce problème, l'utilisateur peut exprimer son propre choix (transition "Choix externe", cadre en pointillés).

4.1.2.1. Etats d'évaluation des attributs

L'ensemble des états du contrôle reprend et étend l'ensemble des états mis en évidence dans l'étude du détachement procédural (cf. §VI.3). On rappelle que la gestion du détachement procédural reposait sur quatre états possibles pour un attribut : **Inconnu** (échec ou attente d'une opération d'évaluation), **Evalué** (réussite d'évaluation), **Demandeur** (Attente du résultat d'une inférence) et **Modifié** (gestion paresseuse des modifications). L'état **Modifié** sortant du cadre du raffinement, son traitement marginal et propre au système de maintien du raisonnement n'est pas considéré par la suite.

Dans le cadre du contrôle global, deux nouveaux états sont introduits pour représenter les blocages momentanés que peut rencontrer le contrôle lorsqu'il est dans la phase de sélection de la prochaine inférence à déclencher. Ces états dénotent donc deux nouvelles situations d'attente :

- **Bloqué** : cet état représente l'étape de transition entre le déclenchement de deux inférences. La tâche essentielle du contrôle lors de cette étape consiste à déterminer si une nouvelle inférence peut être déclenchée soit à l'issue d'un nouvel événement de raffinement, soit après l'échec d'exécution d'une inférence prioritaire.
- **Indécis** : cet état dénote une situation de choix indéterministe. En effet, une définition incomplète d'une priorité peut entraîner la sélection simultanée de plusieurs inférences à déclencher (par réduction de domaine, par défaut, ou par détachement procédural). Le contrôle est alors tributaire d'une décision externe (utilisateur).

De son côté, l'état **Inconnu** est réservé à la description d'un cas d'attente insoluble pour le contrôle : aucune sélection n'est possible car l'ensemble des inférences déclenchables est épuisé. Plus qu'une réelle attente d'évaluation, cet état dénote donc l'échec total de l'évaluation d'un attribut. L'état **Evalué** conserve, quant à lui, sa définition présentée lors de la description du détachement procédural : la réussite d'une inférence entraîne l'affectation de la valeur trouvée à l'attribut. Les états **Inconnu** et **Evalué** constituent les deux états finaux du contrôle d'évaluation d'un attribut.

4.1.2.2. Alternance des phases de sélection et d'exécution d'inférence

L'alternance des phases de sélection et d'exécution d'inférence se traduit au niveau de l'attribut par le passage alterné entre les états **Bloqué** et **Demandeur**.

Dans le cas de l'état **Bloqué**, le contrôle attend que le raffinement permette l'accès de l'instance à une inférence prioritaire. Dès que c'est le cas, l'attribut passe à l'état **Demandeur** et le contrôle déclenche l'inférence prioritaire.

L'attribut reste dans l'état **Demandeur** tant que l'exécution d'inférence ne peut aboutir ni sur un échec, ni sur une réussite. Dans ce cas, l'exécution est en attente d'informations qui peuvent être : les valeurs d'attributs nécessaires en entrée du détachement procédural ou la modification du domaine de valeurs des attributs impliqués dans une inférence par réduction de domaine.

L'échec d'une inférence provoque pour l'attribut le passage de l'état **Demandeur** à l'état **Bloqué**. Le contrôle vérifie alors si une nouvelle inférence prioritaire peut être déclenchée ; si ce n'est pas le cas, il reste en attente de nouvelles informations de raffinement permettant l'accès à un nouvelle inférence prioritaire ou passe dans l'état **Inconnu** s'il ne reste plus d'inférence sélectionnable.

4.1.2.3. Développement de la stratégie d'évaluation d'un attribut

Tant que l'attribut n'aboutit pas dans l'un des états finaux **Inconnu** ou **Evalué**, la stratégie d'évaluation d'un attribut se développe en fonction du raffinement de l'instance et des échecs d'inférences déclenchées. Cette stratégie est exprimée à travers l'ensemble des déclencheurs d'inférence qui sont associés à un attribut par la description des objets.

Du point de vue du raffinement, le développement de la stratégie se traduit par la découverte de nouveaux déclencheurs accessibles (leur classe d'accès est sûre pour l'instance) et/ou de nouveaux déclencheurs inaccessibles (leur classe d'accès est impossible pour l'instance). Alors que les déclencheurs inaccessibles dénotent des inférences qui ne peuvent pas être déclenchées, les déclencheurs accessibles décrivent des inférences dont le déclenchement est possible et dépend uniquement des priorités.

Du point de vue de l'échec d'une inférence, le développement de la stratégie correspond au traitement de l'abandon du déclencheur qui la représente. La caractéristique principale de ce déclencheur étant qu'il dénote le déclencheur accessible le plus prioritaire, son abandon provoque la désignation de nouveaux déclencheurs prioritaires et éventuellement la sélection d'une nouvelle inférence à déclencher.

Pour assurer l'alternance des phases de sélection et d'exécution des inférences, le contrôle doit conserver et maintenir à jour l'état de développement de la stratégie associée à chaque attribut. Etant donnée la contribution apportée par chacune de ces phases (raffinement ou échec), l'état de développement d'un attribut A d'une instance I peut être décrit par la partition de $\mathfrak{DL}(A)$, l'ensemble de ses déclencheurs, en trois catégories :

- **Desact(A, I)** : les **déclencheurs désactivés** correspondant aux inférences qui ne peuvent plus être déclenchées soit parce qu'elles l'ont déjà été, soit parce que la classe d'accès du déclencheur a été déclarée **impossible** par le raffinement de l'instance ;
- **Act(A, I)** : les **déclencheurs activés** correspondant aux inférences non déclenchées et dont la classe d'accès a été déclarée **sûre** par le raffinement ;

- **Inact(A, I)** : les **déclencheurs inactivés** dont la classe d'accès reste **possible**.

La gestion de l'état de la stratégie d'un attribut constitue la tâche de base du contrôle d'évaluation. Elle consiste à calculer à chaque nouvel événement pouvant intervenir soit lors de la phase de sélection, soit dans la phase d'exécution, le nouvel état de développement (cf. Figure VII.5).

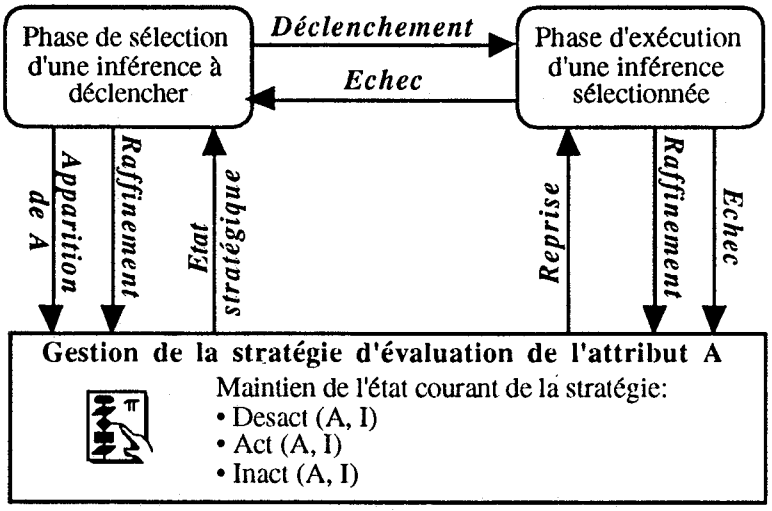

Figure VII.5 : Mise en place de l'alternance des phases de sélection et d'exécution d'inférence en fonction de la gestion de la stratégie d'évaluation d'un attribut. Chaque événement pouvant intervenir durant l'une de ces phases provoque le changement d'état de stratégie de A.

Cette gestion de fond lui permet :

- lorsque l'attribut est dans l'état **Bloqué** (phase de sélection), de décider quand et quelle nouvelle inférence peut être appliquée,
- lorsque l'attribut se trouve dans l'état **Demandeur** (phase d'exécution), de préparer la prochaine phase de sélection qui pourrait éventuellement succéder à un échec d'exécution de l'inférence en cours.

La mise en place effective de la gestion de la stratégie d'évaluation A d'une instance I tire parti des propriétés particulières de l'évolution des ensembles Desact(A, I), Act(A, I) et Inact(A, I) pour réduire l'ensemble des déclencheurs à prendre en compte pour représenter l'état stratégique d'un attribut A. Le principe de cette gestion est décrit en annexe.

4.2. Phase de sélection des inférences

La phase de sélection des inférences constitue l'étape par laquelle le contrôle gère l'enchaînement des tentatives d'inférence. Elle correspond au traitement de l'état **Bloqué** et de l'état particulier **Indécis** et permet la transition de l'attribut vers l'un des états **Inconnu** ou **Demandeur**. Les événements concernés par cette phase sont donc l'apparition de l'attribut dans l'instance, le raffinement de l'instance, l'échec d'une inférence et, éventuellement, la sélection d'une inférence par l'utilisateur dans le cas d'un choix non-déterministe.

L'objectif de cette phase est d'assurer que le contrôle déclenche les inférences dès qu'il le peut. Pour ce faire, le contrôle doit tenir compte de l'état courant de la stratégie d'un attribut et des priorités établies entre les déclencheurs. La sélection d'un déclencheur est assurée si :

- d'une part, **il fait partie des déclencheurs activés** et,
- d'autre part, **il est prioritaire**. La notion de priorité est, dans ce cas, restreinte à l'ensemble des déclencheurs qui n'ont pas encore été désactivés, c'est-à-dire l'ensemble des déclencheurs activés et inactivés.

Dans un premier temps, l'ensemble des déclencheurs prioritaires non désactivés est présenté en fonction de l'état de la stratégie d'un attribut. Dans un second temps, la définition du critère de sélection des déclencheurs **applicables** permet de caractériser les différents états de contrôle dans lesquels un attribut peut transiter à chaque étape de la phase de sélection.

4.2.1. Ensemble des déclencheurs prioritaires non désactivés

Lorsque le contrôle entre dans une nouvelle phase de sélection d'inférence, les déclencheurs pouvant faire l'objet de cette sélection sont limités à ceux qui n'ont pas encore été désactivés. En effet, puisque tout déclencheur inaccessible ou ayant déjà mené à un échec d'inférence représente une étape de la stratégie qui a été écartée, il ne peut donc plus être sélectionné.

Etant donné l'état de développement d'un attribut A d'une instance I, l'ensemble des déclencheurs restant sélectionnables, noté **Sel(A, I)**, est défini de la façon suivante :

- **Sel(A, I)** = $\mathfrak{DT}$(A) \ **Desact(A, I)** = **Act(A, I)** $\cup$ **Inact(A, I)**

L'ensemble des déclencheurs prioritaires pour cette phase de sélection sont donc ceux qui sont maximaux pour $\rightarrow$ dans **Sel(A, I)**. Soit **Max(A, I)** cet ensemble, il est défini de la façon suivante :

- **Max(A, I)**={ d$\in$ **Sel(A, I)** / $\forall$d'$\in$ **Sel(A, I)**, d'$\not\rightarrow$d}

La figure suivante (cf. Figure VII.6) décrit un état possible de la stratégie d'évaluation de l'attribut PrixBase pour une instance de voiture I. Cet état est caractérisé par la partition de $\mathfrak{DT}$(PrixBase) en déclencheurs activés, inactivés et désactivés. L'ensemble Max(PrixBase, I) est obtenu à partir de cette partition.

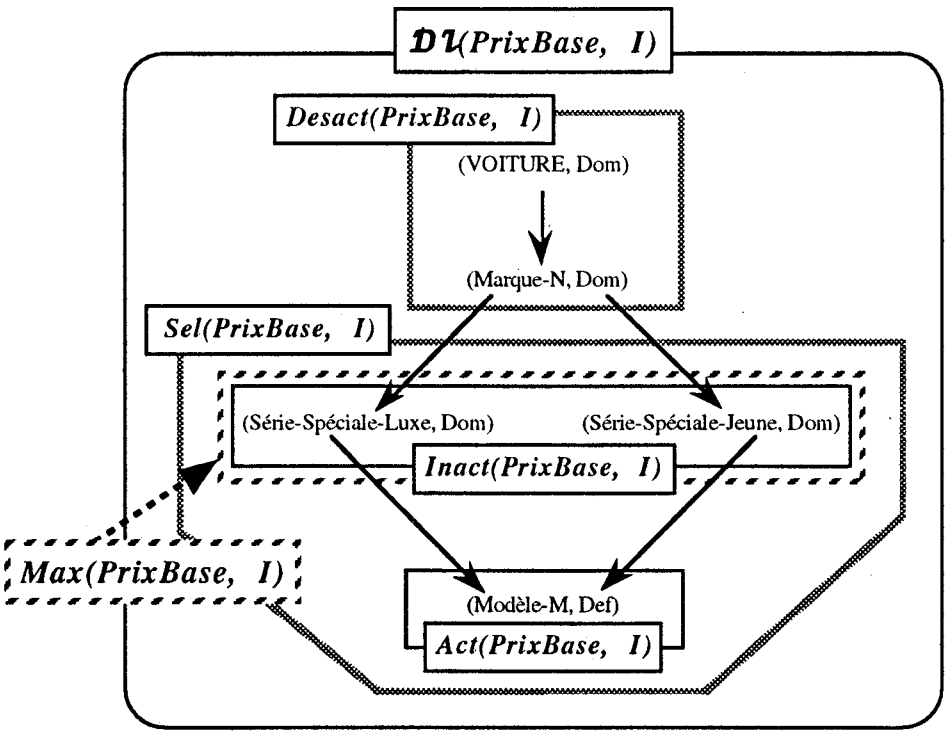

Figure VII.6 : Exemple d'obtention de Max(PrixBase, I) en fonction d'un état particulier de la stratégie d'évaluation de PrixBase. Cet état est décrit par la partition de l'ensemble $\mathfrak{DT}$(PrixBase) en Desact(PrixBase, I), Act(PrixBase, I) et Inact(PrixBase, I). Dans cet exemple, l'ensemble Max(PrixBase, I) est égal à l'ensemble Inact(PrixBase, I)

Dans cet exemple (cf. Figure VII.6), les événements qui ont mené à cet état stratégique de l'évaluation de PrixBase sont le raffinement de l'instance jusqu'à la classe Modèle-M (le

déclencheur (Modèle-M, Def) est activé) et l'échec de deux inférences par réduction de domaine (les déclencheurs (VOITURE, Dom) et (Marque-N, Dom) sont déjà désactivés). Puisque les classes Série-Spéciale-Luxe et Série-Spéciale-Jeune restent possibles pour l'instance I, les déclencheurs associés à ces classes sont inactivés. L'union de Inact(PrixBase, I) et Act(PrixBase, I) admet donc deux déclencheurs maximaux : (Série-Spéciale-Luxe, Dom) et (Série-Spéciale-Jeune, Dom). Ces deux déclencheurs forment l'ensemble Max(PrixBase, I).

Initialement, toutes les classes étant possibles pour l'instance I et aucun déclencheur n'étant déjà appliqué, **Desact(A, I)** est réduit à l'ensemble vide. Lorsque l'attribut A apparaît dans l'instance I, la valeur de **Max(A, I)** est donc l'ensemble $\mathbf{Max_0(A)}$ des déclencheurs maximaux de $\mathfrak{DT}(A)$:

- $\mathbf{Max_0(A)}=\{ d \in \mathfrak{DT}(A) / \forall d' \in \mathfrak{DT}(A), d' \not\rightarrow d \}$

Dans l'exemple d'ordonnancement de l'attribut PrixBase présenté ci-dessus, cet ensemble serait donc donné par $\mathbf{Max_0(PrixBase)}=\{$ (VOITURE, Dom) $\}$

Le recalcul de l'ensemble Max(A, I) à chaque nouvelle phase de sélection est évitée en assurant sa mise à jour lors de la gestion de l'état de stratégie de l'attribut A. Cette mesure permet de plus de réduire efficacement l'ensemble d'informations nécessaires au contrôle pour représenter l'état courant de la stratégie d'évaluation d'un attribut (cf. Annexe).

4.2.2. Sélection des inférences et états d'évaluation d'un attribut

Une fois que l'ensemble Max(A, I) est obtenu pour un attribut A d'une instance I qui se trouve dans l'état **Bloqué**, la sélection consiste à déterminer s'il existe des déclencheurs qui sont à la fois prioritaires et activés. Tout déclencheur vérifiant ces deux propriétés est dit **applicable** car il dénote une inférence candidate au déclenchement.

Soit **App(A, I)** l'ensemble des déclencheurs applicables de l'attribut A, cet ensemble se définit donc par :

- $\mathbf{App(A, I) = Max(A, I) \cap Act(A, I)}$

Tous les états pouvant être atteints à partir de l'état **Bloqué** peuvent être caractérisés en fonction des valeurs de App(A, I) et Max(A, I) :

App(A, I)	Max(A, I)	Etat	Interprétation
$\{d\}$	**Indifférent**	**Demandeur**	Un seul déclencheur actif et prioritaire étant applicable, l'inférence qui lui correspond est déclenchée.
$\{d_1,\ldots, d_i\}_{i>1}$	**Indifférent**	**Indécis**	Plusieurs déclencheurs sont applicables simultanément. Cet état dénote la présence d'un choix indéterministe introduit par une définition incomplète de la priorité. L'évaluation est tributaire d'une décision externe (utilisateur).
Ø	$\{d_i\}_{i\geq 1}$	**Bloqué**	Aucun déclencheur n'est applicable, l'évaluation de l'attribut est en attente d'un apport supplémentaire de connaissances sur le rattachement de l'instance.
	Ø	**Inconnu**	Aucun déclencheur n'est applicable et il n'existe plus de déclencheur actif. L'évaluation de l'attribut est en échec. Seule une révision de l'instance permettrait d'évaluer l'attribut.

Lorsque l'attribut A passe de l'état **Bloqué** à un nouvel état **Bloqué**, la phase de sélection attend qu'un nouveau raffinement permette de modifier l'état de la stratégie de l'attribut. A chacune de ces modifications, les ensembles Max(A, I) et App(A, I) sont mis à jour en conséquence. La phase de sélection est ainsi réappliquée jusqu'à ce que l'attribut transite dans un état différent de l'état **Bloqué**.

Lorsque l'attribut A passe dans l'état **Indécis** ($App(A, I)=\{d_1,\ldots, d_i\}_{i>1}$) le contrôle est en attente du choix par l'utilisateur d'un de ces déclencheurs applicables. Lorsque l'utilisateur s'acquitte de cette tâche, le contrôle force simplement l'ensemble App(A, I) au singleton contenant le déclencheur choisi. L'attribut passe alors dans l'état **Demandeur**, l'inférence est déclenchée et le cycle du contrôle d'évaluation reprend normalement.

4.3. Phase d'exécution des inférences

La phase d'exécution des inférences pour un attribut A prend place dès qu'un seul déclencheur d'inférence est sélectionné par le contrôle, $App(A, I)=\{d\}$. Cette phase intervient donc à l'issue d'une transition de l'état **Bloqué** à l'état **Demandeur**. Elle inclut la mise en place de l'inférence correspondant au déclencheur sélectionné et la gestion d'éventuelles phases d'attentes et de reprises de l'inférence. Cette phase se conclut soit par l'échec (retour à l'état **Bloqué**), soit par la réussite de l'inférence (passage à l'état **Evalué**).

4.3.1. Mise en place d'une inférence

Quelle que soit la source de connaissances utilisée, une inférence d'attribut est perçue de la même façon par le contrôle. En effet, à l'instar du fonctionnement du détachement procédural, le comportement de toute inférence se traduit au niveau du contrôle par l'un des états d'exécution suivants : **Echec**, **Réussite** ou **Attente**.

4.3.1.1. Application d'un déclencheur de source Défaut

Une inférence provenant d'une source de connaissances défaut (consultation d'une facette défaut) connaît un traitement trivial. Quand un déclencheur de ce type est appliqué, l'inférence de l'attribut se traduit simplement par une affectation à l'attribut de la valeur indiquée par la facette défaut. Cette facette est celle qui est associée à l'attribut dans la classe désignée comme classe d'accès du déclencheur.

L'attribut qui fait l'objet de cette inférence passe donc systématiquement de l'état **Demandeur** à l'état **Evalué**. C'est-à-dire que ce type d'inférence ne connaît qu'un état possible : l'état **Réussite**.

4.3.1.2. Application d'un déclencheur de source Détachement Procédural

Pour l'essentiel, l'exécution du détachement procédural reste celle présentée dans le chapitre précédent (cf. §VI.3). Néanmoins, son intégration dans le contexte général du contrôle d'évaluation d'un attribut nécessite une adaptation aux nouveaux états possibles d'un attributs : **Indécis** et **Bloqué**. En effet, on rappelle que l'état d'exécution du détachement est dépendant de l'état des attributs de l'instance qu'il requiert en entrée.

Etant donné que ces deux nouveaux états représentent une situation d'attente d'évaluation, ils jouent donc le même rôle que celui de **Demandeur**. C'est-à-dire que si l'un des attributs demandés en entrée du détachement procédural se trouve dans l'un de ces trois états d'attente, le détachement procédural est lui-même dans l'état **Attente**. L'attribut pour lequel le détachement procédural a été déclenché reste, quant à lui, dans l'état **Demandeur**.

Par ailleurs, dans le cadre du contrôle général, l'état **Inconnu** signifie l'échec total de l'évaluation d'un attribut. Aussi pour éviter un blocage du détachement procédural, et donc de l'attribut qu'il était sensé évalué, une mesure supplémentaire est adoptée : si l'un des attributs demandés en entrée du détachement procédural est dans l'état **Inconnu**, le détachement procédural passe dans l'état **Echec**. L'état **Réussite** du détachement procédural reste quant à lui inchangé (délivrance d'un résultat).

4.3.1.3. Application d'un déclencheur de source domaine

Le principe qui régit l'exécution des inférences par réduction de domaine s'appuie sur une exploitation particulière du module de gestion de contraintes, MICRO. Cependant, il rompt complètement avec le principe des inférences par contraintes proposé à l'occasion de l'intégration de ce module (cf. §VI.2).

4.3.1.3.1. Nécessité d'un nouveau couplage entre MICRO et TROPES

Telle qu'elle a été présentée, l'inférence d'une valeur d'attribut par les contraintes est le résultat d'une phase de réduction de domaine menée par MICRO ; le domaine étant réduit à un singleton la valeur trouvée est systématiquement affectée à l'attribut concerné. Il est donc important de noter que, selon ce principe, le contrôle des inférences d'attribut par contraintes appartient complètement au module MICRO. En d'autres termes, le système de raisonnement est dans l'impossibilité de contrôler les modifications que peut apporter le module MICRO à une instance de la base de connaissances. Dans le contexte d'un système où plusieurs mécanismes d'inférence coexistent, un tel privilège ne peut être accordé aux inférences par contraintes.

C'est donc pourquoi la solution proposée ici consiste à intégrer les inférences par contraintes dans le cadre général du contrôle d'évaluation des attributs. La différence fondamentale entre cette approche et la précédente réside dans le fait qu'il incombe maintenant au système de raisonnement par le biais du contrôle d'évaluation d'un attribut de décider quand une inférence par contrainte peut être tentée. Cette tentative se traduit par une consultation du module MICRO sur l'état du domaine de valeurs de l'attribut et, si besoin est, sur l'état du réseau de contraintes dans lequel il est impliqué. Afin d'éviter toute ambiguïté entre les deux approches, ce type d'inférences est appelé **inférences par réduction de domaine**.

Les inférences par réduction de domaine auxquelles peut prétendre un attribut sont dénotées par les déclencheurs de source domaine de sa stratégie d'évaluation. Le déclenchement de l'une d'entre elles constitue donc une étape stratégique précise qui indique au contrôle que la source de connaissances courante de l'attribut est son domaine de valeurs.

Une inférence par réduction de domaine peut ainsi être perçue comme la délivrance pour le module MICRO d'une autorisation d'inférer par contraintes la valeur d'un attribut particulier. Cette autorisation est maintenue tant que le contrôle ne peut établir l'échec ou la réussite de cette étape stratégique ; l'inférence est alors dans un état d'attente. Chaque inférence par réduction de domaine d'un attribut étant chargée par la stratégie de couvrir une phase spécifique de l'évolution du domaine de valeurs, la notion de réussite ou d'échec est donc définie en fonction de la capacité ou de l'incapacité de MICRO à réduire le domaine de l'attribut à un singleton durant cette phase.

L'ensemble de réductions que peut couvrir chacune des inférences par réduction de domaine est défini par le découpage en phases de raffinement qu'induit l'ensemble des déclencheurs de source domaine associés à l'attribut.

4.3.1.3.2. Découpage stratégique des réductions de domaine d'un attribut

Chaque réduction que fait subir le raffinement d'une instance au domaine de valeurs d'un attribut constitue une étape de développement d'une inférence par réduction de domaine. Correspondant à l'application d'un déclencheur de source domaine particulier, l'exécution d'une telle inférence est limitée aux réductions de domaine que recouvre ce déclencheur. Ces réductions correspondent à celles des étapes de raffinement de l'instance comprenant la classe d'accès du déclencheur et l'ensemble de ses sous-classes qui n'ouvrent pas l'accès à un autre déclencheur de source domaine.

Comme le montre l'exemple suivant (cf. Figure VII.7), l'évolution du domaine de valeurs de l'attribut PrixBase du concept VOITURE peut être décomposée en fonction des étapes de réduction dénotées par chaque déclencheur de source domaine.

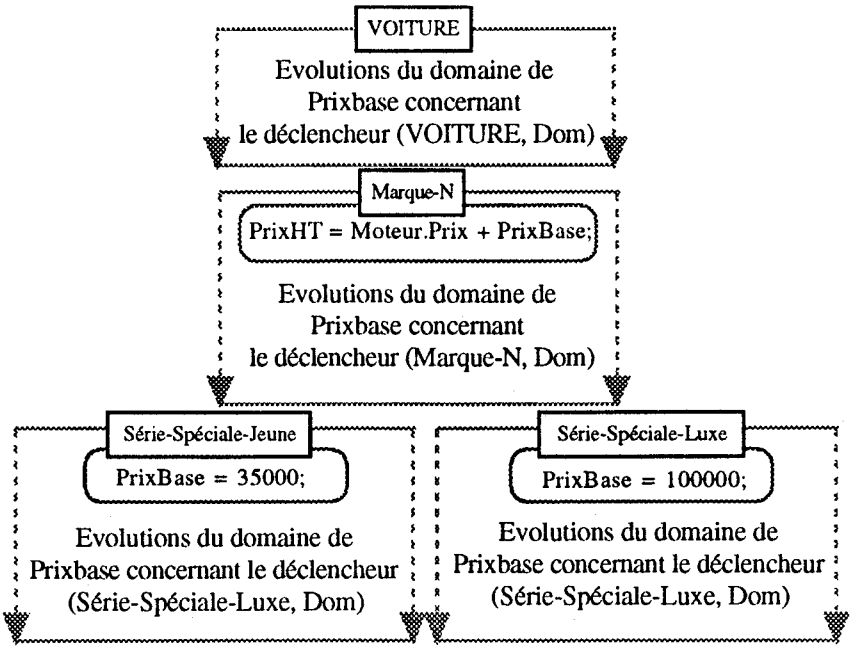

Figure VII.7 : L'ensemble des réductions de domaine que l'attribut PrixBase est susceptible de subir lors du raffinement d'une instance, est décomposé en fonction des déclencheurs de source domaine qui lui sont associés par la description du concept VOITURE. Le développement d'une inférence par réduction de domaine est limité aux évolutions du domaine que recouvre le déclencheur qui a permis sa mise en place.

Par rapport à la gestion des autres types d'inférences, la gestion des inférences par réduction de domaine pose donc un problème particulier au contrôle d'évaluation d'un attribut. En effet, il faut nécessairement restreindre une telle inférence à délivrer un résultat dans les limites de sa phase de réductions. En d'autres termes, si le domaine de valeurs d'un attribut n'est pas réduit à un singleton au sortir des étapes de raffinement permises par l'inférence, cette dernière doit être déclarée en échec.

La gestion du contexte d'exécution des inférences par réduction de domaine associées à un attribut s'appuie sur celle des états de sa stratégie d'évaluation (cf. Annexe). Elle consiste à suivre parallèlement les évolutions du domaine d'un attribut et le développement de la stratégie. Cette confrontation permet de signifier l'échec des inférences par réduction de domaine au fur et à mesure du raffinement de l'instance. L'échec de certaines inférences peut ainsi être constaté avant même qu'elles soient déclenchées.

Le découpage stratégique de l'évolution du domaine implique qu'à un état particulier du domaine ne peut correspondre qu'une inférence. Aussi, lorsque le raffinement permet d'activer un déclencheur de source domaine, le contrôle doit vérifier qu'aucun autre déclencheur de ce type n'est déjà activé ou en cours d'application. Dans le cas contraire, le contrôle doit rejeter l'un de ces déclencheurs, sa décision dépend de l'état du domaine et des priorités.

En revanche lorsqu'une inférence par réduction de domaine est déclenchée alors que l'instance est dans un état de raffinement correspondant à son contexte d'exécution, le contrôle doit établir l'état d'exécution de cette inférence.

4.3.1.3.3. Déclenchement d'une inférence par réduction de domaine

Le déclenchement d'une inférence par réduction de domaine est gérée par le contrôle suivant un schéma analogue à celui du détachement procédural. En effet, lorsqu'elle est déclenchée, une telle inférence peut aboutir dans l'un des trois états suivants : **Réussite**, **Echec**, ou **Attente**. L'état d'exécution est obtenu par le contrôle à l'issue de la consultation des contraintes gérées par le module MICRO.

Soit un attribut A d'une instance I, impliqué dans un réseau de contraintes R géré par MICRO, le déclenchement d'une inférence par réduction de domaine sur A consiste à consulter le domaine associé à l'attribut A dans R, deux résultats sont alors possibles :

- **le domaine est réduit à un singleton.**

  L'inférence aboutit sur un succès, son état d'exécution est donc **Réussite.** La valeur contenue par le singleton est affectée à l'attribut qui passe dans l'état **Evalué.**

- **le domaine possède plusieurs valeurs possibles.**

  L'inférence ne permet pas l'évaluation de l'attribut, deux états d'exécution sont possibles : soit **Attente**, soit **Echec.**

  De la même façon que le détachement procédural met en place les demandes de valeurs des attributs d'entrée et établit en fonction des réponses obtenues l'état d'exécution, l'inférence par réduction de domaine appliquée à A se base sur une consultation de l'ensemble, noté R(I), des attributs de I impliqués dans R. A l'issue de cette consultation, l'état d'exécution est obtenu de la façon suivante :

  - **Echec** : tous les attributs de R(I), excepté A, sont dans un état soit **Evalué,** soit **Inconnu.** En effet, toute réduction des domaines d'attributs appartenant à R(I) \ {A} étant impossible, toute réduction du domaine de A devient donc aussi impossible.
  - **Attente** : Il existe au moins un attribut A' de R(I), différent de A, se trouvant dans l'un des états d'attente d'évaluation suivants : **Demandeur, Bloqué** ou **Indécis.** En effet, la réduction du domaine de l'attribut A' étant possible, celle de A l'est aussi par propagation des réductions de A'.

Afin d'illustrer les situations d'attente, de réussite et d'échec, différents cas d'inférence par réduction de domaine sont étudiés à partir du réseau de contraintes réduit à $X=Y^2$:

- X est l'attribut faisant l'objet de l'inférence par réduction de domaine.

  Y est dans un état d'attente quelconque, X passe dans l'état d'attente **Demandeur.** Si Y passe dans l'état **Inconnu**, le réseau ne permet pas d'inférer X, l'inférence échoue. Si au contraire Y passe dans l'état évalué, Y=2, alors à l'issue des réductions de domaine menées par MICRO, l'inférence réussit et X peut être évalué, X=4.

- Y est l'attribut faisant l'objet de l'inférence par réduction de domaine.

  X est dans un état d'attente quelconque, Y passe dans l'état d'attente **Demandeur.** Si X est évalué, X=4, MICRO réduit le domaine de Y à {-2, 2}. Cette réduction ne rendant pas un singleton et tous les autres attributs du réseau (ici X) étant évalués, l'inférence de Y aboutit donc à un échec.

- X et Y font tous deux l'objet d'une inférence par réduction de domaine.

  Ils se retrouvent demandeurs l'un de l'autre à travers le réseau $X=Y^2$. Seul le raffinement de l'instance provoquant des modifications de X ou Y peuvent débloquer une telle situation. Dans ce cas, pour l'un ou l'autre, l'inférence déclenché devra échouer pour laisser la place au déclenchement d'une autre.

Une inférence par détachement procédural ou par réduction de domaine pouvant se trouver dans une situation d'attente, le contrôle doit garantir leur reprise quand un nouvel apport d'information le permet.

4.3.2. Gestion des attentes/reprises

La répercussion des mises à jour d'une instance peut provoquer la reprise de certaines inférences. La mise en place de la reprise d'un détachement procédural diffère légèrement de sa version initiale du fait de l'extension de l'ensemble d'états d'évaluation d'un attribut. Quant à elle, la reprise d'une inférence par réduction de domaine dépend, d'une part, des limites que lui impose la stratégie d'évaluation et, d'autre part, des modifications pouvant altérer un réseau de

contraintes. Enfin, l'expression du flot de données entre détachement procédural et instance demandeuse est révisée afin de déléguer au module MICRO la gestion de toutes les reprises.

4.3.2.1. Reprise du détachement procédural

Pour un attribut en attente du résultat d'un détachement procédural, on rappelle que les conditions de reprise sont fixées par la réussite ou l'échec de l'évaluation des attributs que la tâche en cours d'exécution requiert en entrée (cf. §VI.3.5.6.4).

La reprise d'un détachement procédural est provoquée lorsque l'un des attributs d'entrée de l'instance de tâche en cours d'exécution connaît une transition d'état vers l'un des états **Inconnu** ou **Evalué**. Cette modification d'état entraîne une nouvelle évaluation de l'état d'exécution de la tâche. Si tous les attributs d'entrée sont évalués, la reprise de la tâche est assurée. Sinon, en fonction des critères présentées en §VII.4.3.1.2, elle passe dans l'état d'**échec**, ou reste dans l'état d'**attente**.

4.3.2.2. Reprise d'une inférence par réduction de domaine

Pour un attribut en attente du résultat d'une inférence par réduction de domaine, la reprise peut intervenir lors :

- **du développement de la stratégie de l'attribut par le raffinement de l'instance.**

  Cette reprise est chargée de signifier l'échec de l'inférence quand les réductions auxquelles peut prétendre le domaine de l'attribut sont à la charge d'une autre inférence par réduction de domaine.

  Si une étape de raffinement permet de rattacher l'instance à une classe qui contraint explicitement l'attribut par des facettes de typage ou par des contraintes, un nouveau déclencheur de source domaine devient donc accessible pour l'instance. L'inférence en attente n'ayant pu évaluer l'attribut avant que le raffinement accède à une autre situation d'inférence par réduction de domaine, la reprise de l'inférence en attente consiste à la déclarer en **échec**. Si la phase de sélection succédant à cet échec rend prioritaire le nouveau déclencheur de source domaine, une nouvelle inférence par réduction de domaine sera donc déclenchée.

  Cette mesure permet d'éviter d'attribuer à un déclencheur de source domaine la réussite d'une évaluation qui serait due aux précisions apportées par le contexte d'un autre déclencheur de source domaine. Sa mise en place est assurée par le contrôle lors de la gestion de la stratégie d'un attribut (cf. Annexe).

- **d'une mise à jour du réseau de contraintes auquel appartient l'attribut.**

  Deux situations de mises à jour peuvent mener à une reprise de l'inférence par réduction de domaine :

  - Une propagation à travers le réseau de contraintes réduit à une seule valeur le domaine de l'attribut en attente :

    La reprise aboutit sur une **réussite**. La valeur désignée par le domaine est affectée à l'attribut. Ce dernier passe alors dans l'état **Evalué**

  - L'état d'un attribut du réseau de contraintes passe dans l'état **Evalué** ou **Inconnu**.

    La reprise consiste à remettre à jour l'état d'exécution de l'inférence ; c'est-à-dire trancher entre un nouvel état d'**attente** ou un état d'**échec** (cf. §VII.4.2.1.3). L'inférence sera en échec si, en dehors de l'attribut qu'elle doit évaluer, il ne reste plus d'attribut dans l'un des états d'attente suivants **Demandeur**, **Indécis** ou **Bloqué**.

4.3.2.3. Propagation des reprises

Dans la pratique, lors de la mise à jour des attributs d'une instance, la gestion des attentes/reprises d'inférence est organisée à partir des propagations gérées par le module de gestion de contraintes MICRO.

Pour ce faire, MICRO doit pouvoir aussi traiter les propagations de modifications permettant la reprise d'un détachement procédural. C'est pourquoi tout échange de valeurs entre un attribut de l'instance demandeuse et un attribut d'entrée/sortie de l'instance de tâche (cf. §VI.3.5.2) est établi par la mise en place d'une contraintes d'égalité.

Les liens de dépendances gérés par le détachement procédural étant alors représentés par des contraintes d'égalité, le flot de données entre l'instance demandeuse et l'instance de tâche est géré complètement par le module MICRO à partir des réseaux de contraintes. Le principe de propagation des reprises est donc mis en œuvre à partir de la modification d'un réseau de contraintes.

5. Objet composite et évaluation des attributs

Cette partie se propose d'étudier le contrôle d'évaluation des attributs dans le contexte d'une catégorie particulière d'objet **complexe** : les **objets composites**. On rappelle qu'un objet **complexe** est un objet qui possède au moins un attribut dont la valeur est un autre objet, une liste ou un ensemble d'objets. Lorsque cet attribut intervient dans la composition de l'objet complexe (comme, par exemple, l'attribut *Moteur* du concept *VOITURE*) et ne traduit pas simplement une propriété (comme, par exemple, l'attribut *Type* de ce même concept), il est dit **attribut composant** (ou encore de nature composant, cf. §II.4.1.1). L'objet complexe devient alors **composite**.

Telle qu'elle est définie dans le modèle TROPES [Mari91], la relation de composition d'objet permet de déléguer à un objet composite la création de ses propres composants. C'est-à-dire qu'un objet composite possède pour ses attributs de nature composant une stratégie d'évaluation prédéfinie.

L'intégration de différents mécanismes d'évaluation d'attributs (non composant) nécessite l'extension de ce principe afin de définir les conditions dans lesquelles un objet composite peut échanger des connaissances avec les composants qu'il crée. Cette nouvelle problématique rentre dans le cadre général du problème de l'évaluation des attributs. Son étude est considérée à deux niveaux différents :

- l'évaluation des attributs composant d'un objet composite : il s'agit de définir dans quelles conditions un objet composite peut créer ses composants.
- l'évaluation des attributs des objets composants : il s'agit de définir quand et comment un objet composite peut enrichir ou bénéficier des connaissances contenues par ses composants.

L'évaluation des attributs composant est traitée selon le principe proposé dans le cadre de l'étude du raisonnement classificatoire dans le modèle TROPES [Mari93]. Les spécificités de la description d'un objet composite (le concept composite) sont rappelées afin de présenter les différentes conditions dans lesquelles un objet composite met en place la création de ses composants (cf. §VII.5.1).

Le problème de l'évaluation des attributs des composants (cf. §VII.5.2) fait, quant à lui, l'objet d'une proposition (cf. §VII.5.3) qui vient étendre le principe du contrôle d'évaluation des attributs présentés précédemment dans ce chapitre. L'idée principale de cette extension consiste à permettre la redéfinition de la stratégie d'évaluation des attributs d'un composant par tout concept composite qui les introduit dans sa description.

5.1. Concept composite et instance d'objet composite

Dans le modèle TROPES, la description d'un concept d'objet composite permet de préciser progressivement, à travers l'affinement de ses attributs composants, les connaissances qu'un objet composite peut posséder initialement sur ses composants (cf. §VII.5.1.1).

L'intégration d'un langage de contraintes au modèle TROPES (cf. §VI.2) permet, de plus, d'exprimer les relations que peuvent entretenir les attributs de l'objet composite et les attributs de ses composants (cf. §VII.5.1.2).

La manipulation d'objet composite tire parti de la sémantique spécifique associée à la relation de composition pour définir à partir d'un objet composite les conditions dans lesquelles ses composants peuvent être créés (cf. §VII.5.1.3).

5.1.1. Affinement d'un attribut composant

La déclaration d'un attribut composant au niveau du concept d'un objet composite comporte deux informations indiquant respectivement le concept dans lequel cet attribut prend ses valeurs (les instances composantes) et le constructeur précisant si l'attribut est mono-valué (constructeur *un*) ou multi-valué (constructeur *liste* ou *ensemble*). Par exemple, l'attribut composant *Moteur* du concept *VOITURE* est mono-valué (un seul moteur par voiture) et prend ses valeurs dans le concept *MOTEUR*, tandis que l'attribut composant *Portes* est multi-valué (plusieurs portes par voiture) et prend des valeurs dans le concept *PORTE*.

Dans une classe de la hiérarchie de spécialisation du concept composite, l'affinement d'un attribut composant peut provenir d'une restriction de son type (désignation d'une classe à laquelle les instances composantes doivent appartenir), et/ou d'une contrainte sur sa cardinalité lorsque l'attribut est multi-valué.

Par exemple, les attributs composants *Moteur* et *Portes* du concept *VOITURE* peuvent être affinés ainsi pour la classe de voitures *Peugeot-106-Sport* :

Moteur **dans** Moteur-Sport ;
Portes **dans** Porte-P106 ;
card 4 ;

Les valeurs que peuvent prendre les attributs composants *Moteur* et *Portes* sont des instances qui doivent appartenir respectivement à la classe des *Moteur-Sport* du concept *MOTEUR* et à la classe *Porte-P106* du concept *PORTE*. L'attribut *Portes* est multi-valué et sa cardinalité est fixée à 4.

Lorsque l'attribut composant représente un ensemble ou une liste d'objets, on peut aussi affiner son type par **éclatement**. L'éclatement a pour but de décrire des sous-ensembles, ou des sous-listes, d'éléments simples dont l'ensemble indivisible forme la valeur de l'attribut composant. Dans [Mari91], deux types d'éclatement possibles ont été définis :

- un éclatement de la description de l'attribut, qui permet une décomposition et la vision immédiate de ses éléments.

  Portes **dans** Porte-avant-P106 | Porte-arrière-P106 ;
  card 2 | 2 ;

  L'ensemble représenté par l'attribut *Portes* se divise en deux sous-ensembles : Porte-avant-P106 et Porte-arrière-P106. Chacun de ces sous-ensembles a sa cardinalité fixée à 2.

- un éclatement de l'attribut lui-même, ce qui permet de distinguer chacune de ses parties et, le cas échéant, l'éclatement récursif du composant.

  Portes

  Portes-avant **dans** Porte-avant-P106 ;
  card 2 ;

  Portes-avant **dans** Porte-arrière-P106 ;
  card 2 ;

L'intégration de contraintes au sein du modèle TROPES a permis d'étendre les possibilités d'affinement d'un attribut composant en permettant d'établir des contraintes impliquant les attributs d'un composant à travers la description de l'objet composite. Cette extension a donné lieu à une adaptation de l'éclatement d'attribut composant.

5.1.2. Contraintes entre attributs d'objets composite et composant

L'exemple du concept VOITURE (cf. §VII.2.1.1) décrit plusieurs situations dans lesquelles l'attribut composant *Moteur* est affiné par des contraintes se trouvant dans les classes de ce concept (les classes *VOITURE* et *Marque-N*). Ce type d'affinement de l'attribut composant permet d'exprimer des contraintes impliquant aussi bien des attributs d'un objet du concept VOITURE que d'un objet du concept *MOTEUR* (cf. §VII.3.1).

Dans le cadre d'un modèle de tâche dans lequel la notion de tâche est représentée par un objet composite [Gens&93] [Gens&94], l'affinement par éclatement d'un attribut composant multi-valué a été redéfini. Le principe de cet éclatement consiste à déclarer des *sous-attributs* de l'attribut composant multi-valué. Chacun de ces *sous-attributs* décrit l'accès à l'un des éléments de l'ensemble ou de la liste de composants que représente l'attribut composant. L'éclatement permettant l'accès à ces composants, il permet aussi d'exprimer des contraintes de classes impliquant leurs attributs.

L'éclatement d'un attribut composant multi-valué est déclaré à l'aide de la facette **eclat** qui est suivie d'une description des sous-attributs similaire à la description d'une classe. En effet, cette description comporte une zone dédiée à l'affinement des sous-attributs et une zone dédiée à la déclaration de contraintes. Un sous-attribut est mono-valué et son type est nécessairement un sous-type de celui que possède l'attribut composant lors de son éclatement. Le nombre de sous-attributs déclarés est nécessairement inférieur ou égal à la cardinalité de l'attribut composant. Enfin, l'éclatement d'un attribut composant dans une classe peut être affiné dans une sous-classe par affinement des sous-attributs, ajout de contraintes et/ou ajout de nouveaux sous-attributs (si la cardinalité de l'attribut composant le permet).

L'exemple suivant montre le cas d'un objet composite, représentant une tâche, qui exploite l'éclatement d'un attribut composant pour représenter la décomposition de la tâche en sous-tâches.

```
{  Tâche-principale
     attributs :
        entrée1   dans  T-entrée1;
        entrée2   dans  T-entrée2;
        décomposition   dans  TACHE;
                        card 2;
           eclat {
              sous-attributs :
                 Sous-T1  dans  T-sous-T1    ; première sous-tâche
                 Sous-T2  dans  T-sous-T2    ; seconde sous-tâche
              contraintes :
                 ; flot de données entre Sous-T1 et Sous-T2
                 Sous-T1.sortie-T1 = Sous-T2.entrée-T2;
           };
        sortie-TP dans  T-sortie-TP;
     contraintes :
        ; flot de données entre Tâche-principale et et Sous-T1
        entrée1 = décomposition.Sous-T1.entrée1;
        entrée2 = décomposition.Sous-T1.entrée2;
        ; flot de données entre Tâche-principale et et Sous-T1
        sortie-TP = décomposition.Sous-T2.sortie-T2
}
```

Dans cet exemple, l'attribut *décomposition* est un attribut composant de constructeur *liste* et de type *TACHE*. Cet attribut subit un éclatement permettant d'indiquer sa décomposition en une séquence de deux sous-tâches. Les contraintes au niveau de la classe *Tâche-principale* permettent d'exprimer le flot de données, les entrées et sorties, entre la tâche principale et les sous-tâches. Les contraintes déclarées au niveau de l'éclatement permettent, quant à elles, d'exprimer le flot de données entre les deux sous-tâches.

5.1.3. Instance composite

D'après [Mari91] [Mari93], l'objet (instance) qui représente la valeur d'un attribut composant, le **composant**, est *dépendant de manière existentielle et exclusive* de l'objet composite qu'il compose. Le composant n'a été créé que pour l'objet composite et ne peut appartenir à aucun autre objet composite. Les connaissances du type d'un attribut composant et de sa cardinalité suffisent pour désigner la ou les valeurs que peut prendre cet attribut.

L'idée majeure, lors de la manipulation d'un objet composite, est de charger le système de créer, d'après les descriptions des attributs composants, les instances qui deviendront les valeurs de ces attributs. L'évaluation d'un attribut composant dépend de son état d'affinement :

- si l'attribut composant est mono-valué, le système crée une instance composante dans le concept dans lequel l'attribut prend ses valeurs, puis cette instance est rattachée à la classe désignée par le type affiné de l'attribut composant
- si l'attribut composant est multi-valué et que sa cardinalité est connue, alors le système crée toutes les instances correspondant aux valeurs de cet attribut (création des composants).
- si l'attribut composant est multi-valué et éclaté dans la description de classe de rattachement de l'objet composite, chaque sous-attribut connu donne lieu à la création d'une instance de composant. Si la cardinalité n'est pas fixée, l'attribut composant n'est que partiellement évalué, seules certaines de ses valeurs (les instances composantes dénotées par les sous-attributs) sont connues.
- sinon, la création des composants ne peut avoir lieu. Elle est différée jusqu'à ce que l'attribut composant fasse l'objet d'informations supplémentaires le plaçant dans l'un des cas précédents.

Afin de faciliter l'exploitation de l'éclatement, l'ordre des éléments contenu par un attribut composant multi-valué est pertinent. En effet, lors d'un appariement d'un objet composite avec une classe contenant la description d'un éclatement, les éléments de l'attribut composant multi-valué sont confrontés dans leur ordre d'apparition avec les sous-attributs dans leur ordre de description.

Dans l'exemple des tâches présenté dans la partie précédente (cf. §VII.5.1.2), la capacité d'un objet composite à créer ses composants lui-même permet de ramener l'exécution d'une tâche à un processus de construction d'objet composite [Gens&92]. En effet, il suffit de créer et de rattacher une instance de tâche à la classe *Tâche-principale* pour que des instances de ses sous-tâches soient créées en séquence. La cardinalité de l'attribut *décomposition* étant connue, l'objet composite *Tâche-principale* peut alors créer ses composants qui constitueront les valeurs des deux sous-attributs déclarés par l'éclatement de *décomposition*.

Un dernier problème reste toutefois à résoudre : le problème de l'évaluation des attributs des composants. En effet, si le composite permet l'évaluation automatique des attributs composants par création d'instances, l'évaluation des attributs de ces instances composantes reste néanmoins à définir. Le mécanisme de création d'objet composite proposé initialement par Olga Mariño repose sur une version du modèle TROPES qui laisse à l'utilisateur le soin d'évaluer les attributs d'une instance. Dans ce contexte, l'évaluation des attributs des composants consiste donc simplement à demander à l'utilisateur la valeur des attributs du composant au fur et à mesure qu'ils apparaissent dans ce dernier.

L'introduction de plusieurs moyens d'évaluation d'attributs et, notamment, la possibilité d'établir des contraintes entre les attributs des instances de composite et de composant, permet

d'envisager de nouvelles possibilités d'évaluation d'attributs. Puisque le concept composite englobe dans sa description les conditions d'échange d'informations avec ses composants, la stratégie d'évaluation de ses attributs prend déjà en compte les situations dans lesquelles les composants peuvent intervenir. En revanche, comme l'exige le principe de description modulaire, un concept composant est décrit indépendamment du rôle qu'il peut jouer dans un concept composite. Dans ce cas, la stratégie d'évaluation des attributs d'un objet composant ne tient pas compte des informations que peut apporter un concept composite. La partie suivante est consacrée au problème que pose le contrôle d'évaluation des attributs d'un objet lorsqu'il participe à la composition d'un autre objet.

5.2. Problème de l'évaluation des attributs des composants

La relation entre le concept décrivant l'objet composite et les concepts décrivant les composants est une relation de type *maître-esclave* [Mari93] : l'accès aux composants d'un objet composite se fait toujours à partir du concept composite. Puisque l'objet composite contrôle aussi lui-même la création de ses composants, on peut plus généralement affirmer que la configuration des composants est toujours contrôlée par l'objet composite. En effet, toute information concernant un composant provient nécessairement de l'objet composite, elle est transmise au composant par le biais de l'attribut composant dont il est la valeur.

5.2.1. Utilisateur, objet composite et configuration des composants

L'objet composite peut être perçu par un composant comme un utilisateur puisqu'il permet :

- d'apporter, grâce aux affinements de type de l'attribut composant, des informations concernant l'appartenance de l'instance composante aux classes de sa hiérarchie de spécialisation.
- de contraindre, grâce aux contraintes de classes du concept composite, les attributs de l'instance composante.

Aussi, le raffinement de l'objet composite constitue une démarche permettant d'accroître les connaissances qu'il possède sur ses composants. En effet, plus l'objet composite sera raffiné, plus la description de ses attributs composants sera affinée.

Toutefois, le raffinement de l'objet composite ne peut suffire à configurer une instance composante. D'une part, certains attributs du composant peuvent ne pas apparaître dans la description du concept composite. D'autre part, l'apparition d'un attribut du composant dans une contrainte de classe de l'objet composite ne signifie pas nécessairement que le composite va contribuer à son évaluation. Il se peut au contraire que ce soit l'attribut du composant qui permette l'évaluation d'attributs de l'objet composite. L'exemple de la modélisation du flot de donnée des tâches (cf. §VII.5.1.2) montre que les échanges d'informations entre composants et objet composite peuvent être effectivement bilatéraux.

Une partie de la configuration d'un composant doit pouvoir être configurée par l'utilisateur lui-même. Dans certains cas, il est même souhaitable que la configuration d'un composant soit complètement assurée par l'utilisateur. Par exemple, dans le contexte d'une application d'aide au diagnostic médical, une maladie peut être représentée par un objet composite possédant comme composant un objet *Symptôme*. Le diagnostic requiert que l'utilisateur (le médecin) puisse configurer lui-même ce composant en fonction des observations qu'il a pu faire sur son patient. Dans ce cas, le raffinement de l'objet composite n'est pas exploité pour configurer le composant, c'est au contraire la configuration du composant qui permet le raffinement de l'objet composite.

Pour permettre à l'utilisateur d'apporter les informations qu'il possède sur un composant, le principe adopté reprend celui de la demande d'information systématique proposée initialement (cf. §VII.5.1.3). Toutefois, le type de connaissances apportées par l'utilisateur est adapté à celui défini dans le cadre du contrôle d'évaluation d'attributs (cf. §VII.3.1) : l'utilisateur contraint le composant. Ainsi, durant le raffinement d'un objet composite, les demandes d'informations à l'utilisateur obéissent au schéma suivant :

- Si un attribut composant apparaît dans l'instance composite, le système demande à l'utilisateur les connaissances de configuration que celui-ci possède sur cet attribut :
  - l'attribut composant est mono-valué, l'utilisateur peut poser des contraintes sur les attributs de l'instance composante et apporter des informations sur l'appartenance de cette instance aux classes de sa hiérarchie d'appartenance.
  - l'attribut composant est multi-valué, l'utilisateur peut contraindre la cardinalité (si elle n'est pas déjà connue) de cet attribut, ainsi qu'apporter des informations concernant l'appartenance de toutes les instances composantes aux classes de leur hiérarchie de spécialisation.
- Si un attribut d'un composant apparaît dans une instance composante lors de la création ou d'un raffinement, l'utilisateur est requis. Il peut, comme lors du raffinement de n'importe quelle instance, poser des contraintes sur cet attribut.

Grâce à cette stratégie d'obtention d'informations, l'objet composite permet à l'utilisateur d'apporter progressivement les informations qu'il possède sur les composants au fur et à mesure du raffinement de l'objet composite. La configuration des composants est ainsi gérée complètement par l'objet composite puisque les connaissances acquises auprès de l'utilisateur les concernant sont obtenues lors de l'évaluation des attributs composants et de leurs affinements.

L'ordre des connaissances ainsi apportées par un objet composite à un composant constitue un problème pour le contrôle d'évaluation des attributs du composant. La partie suivante est dédiée à la description de ce problème et à l'étude de solutions possibles.

5.2.2. Problème du raffinement d'un objet composite

L'objet composite constitue pour une instance composante une source de connaissances externe. Or, le contrôle d'évaluation des attributs, tel qu'il a été défini en fonction du raffinement de l'instances (cf. §VII.4.1), repose sur le fait que les apports de connaissances externes obéissent à un ordonnancement précis : toute connaissance externe sur un attribut est fournie lors de son apparition dans l'instance. Tout ajout de connaissances à un attribut qui est ultérieur à cette phase d'apparition et qui ne répond pas à une requête explicite du système est considéré comme une tentative de modification de l'instance.

Conceptuellement, le principe sur lequel se base ce choix consiste à affirmer que lorsqu'un agent externe est interrogé par le système sur la configuration d'un attribut, il fournit toutes les connaissances qu'il possède sur cet attribut. S'il désire fournir de nouvelles informations sur cet attribut, c'est donc qu'il veut le modifier. Techniquement, le contrôle d'évaluation de l'attribut exploite ce principe pour établir si l'agent externe connaît ou ne connaît pas la valeur de l'attribut et donc définir un critère d'échec de l'évaluation de l'attribut par l'agent externe.

Dès lors qu'un tel échec peut être constaté, le contrôle d'évaluation écarte définitivement l'agent externe des mécanismes candidats à l'évaluation de l'attribut. Il développe en conséquence la stratégie d'évaluation de l'attribut. Si l'agent externe était considéré comme une source de connaissances prioritaire pour l'attribut, son élimination permettra peut-être au contrôle d'évaluer l'attribut en consultant une autre source de connaissances (par exemple, l'emploi d'une facette défaut).

Si l'agent externe entreprend dans ces conditions de nouveaux ajouts de connaissances sur l'attribut, il remet directement en question les développements de la stratégie qui ont pu être déduits grâce à son échec. Le contrôle d'évaluation perçoit donc cette intervention comme un comportement non monotone de l'agent externe.

Le problème que pose un objet composite au contrôle d'évaluation des attributs d'un composant correspond exactement au schéma d'un agent externe possédant un comportement non monotone. En effet, les différents affinements appliqués à un attribut composant lors du raffinement d'un objet composite permettent la pose de contraintes sur les attributs d'une instance composante après la phase d'introduction de ces attributs dans l'instance.

La figure suivante (cf. Figure VII.8) illustre, à partir de l'exemple des voitures, le problème que peut poser le raffinement d'un objet composite au contrôle d'évaluation des attributs.

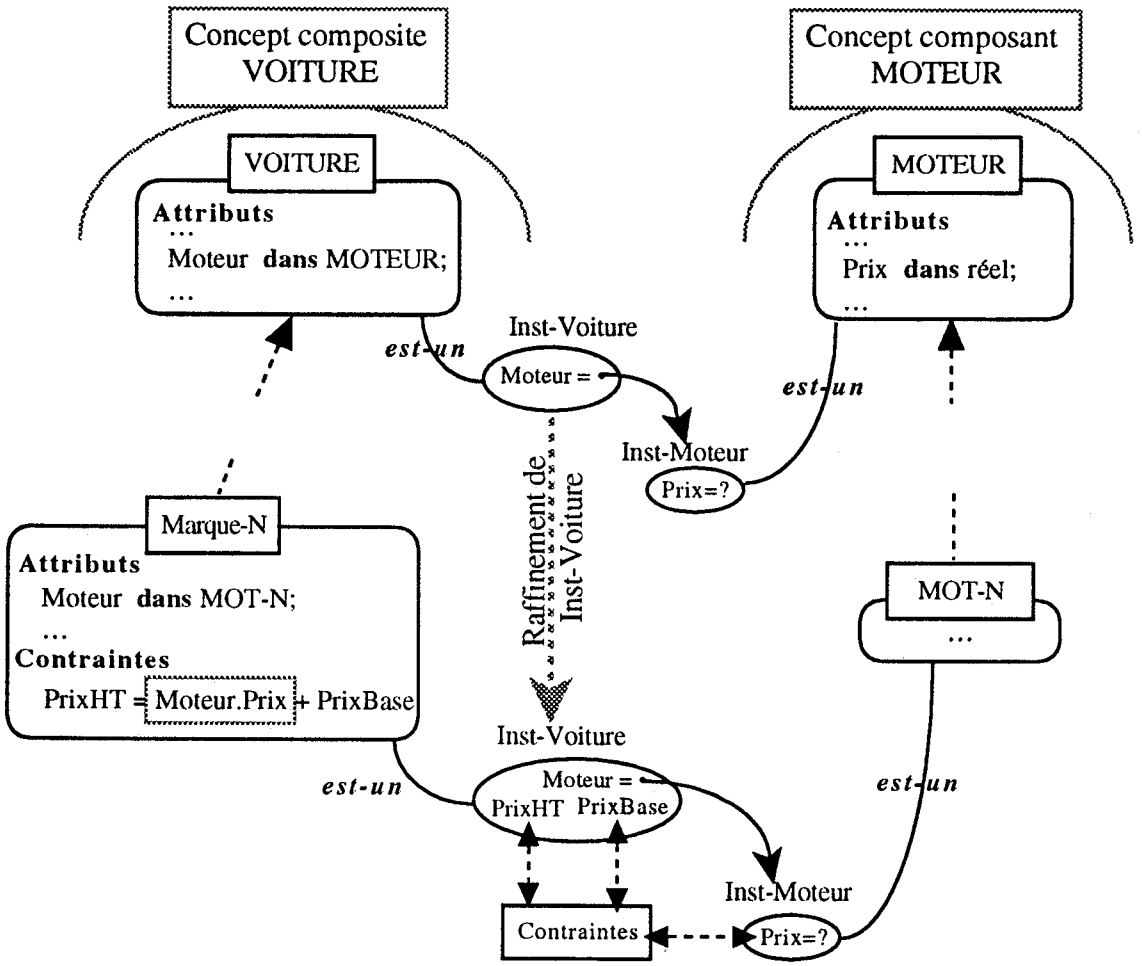

Figure VII.8 : Illustration du problème que pose le raffinement d'un objet composite au contrôle d'évaluation des attributs de ses composants. Lors de sa création, l'instance de voiture *Inst-Voiture* évalue son attribut composant *Moteur* en déclenchant la création d'une instance de moteur, appelée *Inst-Moteur*. L'affinement de l'attribut *Moteur* ayant pour type la classe *MOTEUR*, l'instance *Inst-Moteur* y est rattachée. Cette opération déclenche l'introduction de l'attribut *Prix* dans *Inst-Moteur*. Lors du raffinement de l'instance composite *Inst-Voiture*, son rattachement à la classe *Marque-N* déclenche la pose de contraintes entre les attributs *PrixHT* et *PrixBase* de cette instance et l'attribut *Prix* de l'instance *Inst-Moteur*. Cette contrainte, externe pour l'instance *Inst-Moteur*, constitue pour le contrôle d'évaluation de l'attribut *Prix* du moteur une tentative de modification puisqu'elle a été posée ultérieurement à la phase d'introduction de l'attribut *Prix* dans l'instance *Inst-Moteur*.

Pour résoudre le problème que connaît le contrôle d'évaluation d'attributs dans le cas d'un objet composite, trois solutions différentes peuvent être proposées :

- Mise en place d'une procédure de révision d'instance capable de traiter les cas où le contrôle d'évaluation d'attributs fait l'objet d'interventions externes (dans ce cas, la pose de contraintes externe sur les composants) qui ne respectent pas le schéma prévu par le raffinement de l'instance.
  - *Avantages* : une telle procédure de révision d'instance rentre dans le cadre général du système de maintien du raisonnement. Elle doit donc être normalement fournie par le système pour prendre en compte et répercuter dans la base de connaissances

tout type de modifications, notamment celles concernant l'évaluation des attributs (cf. §III).

- *Inconvénients* : faire appel à une procédure de révision d'instance à chaque fois que l'objet composite pose des contraintes sur ses composants peut être coûteux et parfois inutile. En effet, dans ce cas, le processus de révision consiste à restituer le contexte dans lequel la pose de contraintes aurait dû être réalisée, puis à vérifier si l'ajout de contraintes change le résultat précédent et enfin, s'il y a lieu, à répercuter les modifications.

• Différer la création des composants à un contexte de raffinement du composite dans lequel plus aucune contrainte ne peut être posée sur les attributs des composants.

- *Avantages* : cette solution assure que le contrôle d'évaluation des attributs des composants ne fera pas l'objet de révision. Sa mise en place peut simplement consister à associer aux attributs composants de l'objet composite leur propre stratégie de création de composants. La gestion des créations de composants rentre alors dans le cadre général du contrôle d'évaluation des attributs. Cette stratégie peut être exprimée à partir de déclencheurs chargés de représenter les situations dans lesquelles un attribut composant peut être évalué (création des composants). Les priorités sur ces déclencheurs sont alors fixées de telle façon que toute création de composant soit entreprise lorsque l'état de raffinement de l'objet composite ne permet plus d'accès possible à de nouvelles contraintes sur les attributs des composants.

- *Inconvénients* : différer la création des composants lors du raffinement de l'objet composite prive ce dernier d'information pertinente lors de son appariement avec les classes de la hiérarchie de spécialisation. D'une part, l'objet composite ne peut s'appuyer sur ses composants pour guider le raffinement (par exemple, dans le diagnostic médical, l'obtention du composant *symptôme* est important pour construire l'objet *maladie*) et, d'autre part, la cohérence de l'objet composite ne peut être garantie tant que ses composants n'ont pu être créés. Il peut alors arriver que la création des composants mène à une incohérence. Dans ce cas, il peut être difficile de mettre en place une procédure de révision puisque l'apport de connaissances a pu être conséquent avant que l'objet composite ait pu créer tous ses composants.

• Prise en compte dans la stratégie d'évaluation des attributs composants des possibilités d'évaluation que définissent les contraintes de classes de la hiérarchie de spécialisation de l'objet composite.

- *Avantages* : comme pour la solution précédente, l'évaluation des attributs d'un composant n'entraîne pas de traitement de révision. De plus, cette solution ne retardant pas la création des composants, elle permet d'en bénéficier pour garantir la cohérence et guider le raffinement de l'objet composite.

- *Inconvénients* : cette solution impose que le concept composant intègre dans la stratégie d'évaluation de certains de ses attributs des informations qui sont liées uniquement au contexte de description de l'objet composite. La description d'un concept composant n'est donc pas indépendante de l'utilisation qui va en être faite. De plus, si ce concept composant intervient dans la description de plusieurs concepts composites, il va alors être *pollué* par des informations provenant de chacun de ces concepts.

La solution qui est adoptée par la suite s'inspire de la troisième solution. En effet, cette solution a l'avantage de remédier aux inconvénients des deux premières et ses propres inconvénients peuvent être surmontés simplement.

5.3. Contrôle d'évaluation des attributs d'un objet composant

Pour mettre en place la solution proposée, deux problèmes doivent être résolus. Le premier problème réside dans la façon d'intégrer et de représenter dans la stratégie d'évaluation d'un attribut d'un concept composant les informations concernant ses possibilités d'évaluation

dans un objet composite (cf. §VII.5.3.1). Le second problème consiste à trouver un moyen d'éviter qu'un concept composant intègre des informations, en l'occurrence celles concernant ses attributs impliqués dans la description d'un concept composite, qui ne sont pas propres à sa description (cf. 5.3.2).

5.3.1. Concept composant et affinement d'un attribut composant

La stratégie d'évaluation d'un attribut A d'un concept composant Ω est donnée par le couple $(\mathfrak{DI}(A), \rightarrow_A)$, où $\mathfrak{DI}(A)$ est l'ensemble de déclencheurs d'inférence associé à l'attribut A par la description de Ω et $\rightarrow_A$ est l'ordre de priorité entre les déclencheurs de A obtenu par application des règles de priorité définies pour A.

Pour prendre en compte au niveau de la stratégie d'évaluation de l'attribut A les nouvelles possibilités d'évaluation qu'un concept composite Ω_C introduit, il faut :

- exprimer ces possibilités sous la forme de déclencheurs d'inférence de A ;
- définir précisément la priorité de ces nouveaux déclencheurs par rapport à ceux existant déjà dans $\mathfrak{DI}(A)$.

Les possibilités d'évaluation de l'attribut A dans le concept composite sont liées à l'affinement d'un attribut composant A_C qui prend ses valeurs dans le concept Ω. Chaque affinement de A_C dans lequel l'attribut A apparaît (A_C.A est impliqué dans une contrainte de classe du concept composite Ω_C) constitue une possibilité d'évaluation de l'attribut A. La source de connaissances qui caractérise une telle situation est le **domaine** (inférence par réduction de domaine) puisque A est alors impliqué dans une contrainte.

5.3.1.1. Accès des déclencheurs créés par le concept composite

Pour l'attribut A, il y aura donc autant de déclencheurs d'inférence de source domaine créés que de situations dans lesquelles A est impliqué dans l'affinement de l'attribut composant A_C. Pour distinguer et compléter ces déclencheurs créés pour A, il faut indiquer leur classe d'accès respective.

Soit $\mathbf{C}$ la classe du concept composant Ω correspondant au type de l'attribut composant A_C lorsqu'il fait l'objet de contraintes de classe impliquant l'attribut A dans la classe $\mathbf{C}_C$ du concept composite Ω_C. Cet affinement de l'attribut A_C constitue pour l'attribut A une situation de création de déclencheur d'inférence de source domaine ; soit *d* ce déclencheur.

Une instance composante n'aura accès au déclencheur *d* qu'à une seule condition : l'instance composite doit appartenir à la classe $\mathbf{C}_C$. Mais, pour appartenir à la classe $\mathbf{C}_C$, l'instance composite doit avoir pour composant une instance qui appartient nécessairement à la classe $\mathbf{C}$. De ce fait, l'accès du déclencheur *d* est défini par l'appartenance de l'instance composite à la classe $\mathbf{C}_C$ et l'appartenance de l'instance composante à la classe $\mathbf{C}$.

Tout déclencheur *d* de l'attribut A du concept composant Ω créé grâce à l'affinement de l'attribut composant A_C du concept composite Ω_C est alors défini ainsi :

- $d = (\mathbf{C}/\mathbf{C}_C, \text{Dom})$

où :

- $\mathbf{C}$ est la classe de Ω définissant le type de l'attribut A_C dans la classe $\mathbf{C}_C$ du concept Ω_C ;
- l'expression $\mathbf{C}/\mathbf{C}_C$ indique que pour pouvoir accéder au déclencheur *d*, une instance de Ω doit appartenir à la classe $\mathbf{C}$ et être le composant d'une instance de Ω_C qui appartienne à la classe $\mathbf{C}_C$.

Dans l'exemple des voitures, l'attribut *Prix* du composant moteur se voit ainsi associer, par la description de la classe *Marque-N* du concept *VOITURE*, le déclencheur d'inférence suivant : (*MOT-N / Marque-N*, Dom).

5.3.1.2. Ordre de priorité des déclencheurs créés par le concept composite

Soit *d* un déclencheur d'un attribut A d'un composant créé par un concept composite, tel que $d = (\mathbf{C}/\mathbf{C}_C, \text{Dom})$. L'appartenance d'une instance composante à la classe $\mathbf{C}$ étant nécessaire pour envisager l'application de *d*, cette information peut être exploitée pour définir la priorité qu'il faut accorder à *d* dans la stratégie d'évaluation de A.

Soit $\mathfrak{Dl}(A, \Omega_C)$ l'ensemble des déclencheurs que le concept composite Ω_C permet de créer pour l'attribut A du concept composant Ω. Pour prendre en compte la description de Ω_C dans la stratégie de l'attribut A, il faut alors intégrer les déclencheurs de $\mathfrak{Dl}(A, \Omega_C)$ à l'ensemble $\mathfrak{Dl}(A)$ en établissant, de plus, leur ordre de priorité. La stratégie de l'attribut A sera alors donnée par le couple $(\mathfrak{Dl}(A) \cup \mathfrak{Dl}(A, \Omega_C), \rightarrow_A)$.

Pour arriver à ces fins, il faut que la priorité $\rightarrow_A$ puisse être aussi applicable aux déclencheurs de $\mathfrak{Dl}(A, \Omega_C)$. Cette priorité obtenue grâce à l'application des règles de priorité repose sur les relations de spécialisation, de généralisation ou d'égalité établies entre les classes d'accès des déclencheurs (cf. §VII.3.3.1). L'application de ces règles doit être étendue au cas des déclencheurs de $\mathfrak{Dl}(A, \Omega_C)$ qui introduisent aussi une classe du concept composite dans la description de leur accès.

Soient $\mathbf{C}$ et $\mathbf{C}'$ deux classes du concept composant Ω, $\mathbf{C}_C$ et $\mathbf{C}'_C$ deux classes du concept composite Ω_C, les relations de comparaison des classes d'accès utilisées par les règles de priorité sont étendues de la façon suivante :

- $(\mathbf{C}/\mathbf{C}_C) > (\mathbf{C}'/\mathbf{C}'_C) \Leftrightarrow (\mathbf{C} > \mathbf{C}')$ ou $[(\mathbf{C} = \mathbf{C}')$ et $(\mathbf{C}_C > \mathbf{C}'_C)]$
- $(\mathbf{C}/\mathbf{C}_C) < (\mathbf{C}'/\mathbf{C}'_C) \Leftrightarrow (\mathbf{C} < \mathbf{C}')$ ou $[(\mathbf{C} = \mathbf{C}')$ et $(\mathbf{C}_C < \mathbf{C}'_C)]$
- $(\mathbf{C}/\mathbf{C}_C) < \mathbf{C}' \Leftrightarrow (\mathbf{C} < \mathbf{C}')$ ou $(\mathbf{C} = \mathbf{C}')$
- $(\mathbf{C}/\mathbf{C}_C) > \mathbf{C}' \Leftrightarrow (\mathbf{C} > \mathbf{C}')$

Les trois premières équivalences reposent sur le principe que l'affinement d'un attribut composant A_C dans la classe $\mathbf{C}_C$ peut être considéré comme la description d'une sous-classe virtuelle de la classe $\mathbf{C}$ représentant le type de A_C (cf. Figure VII.9).

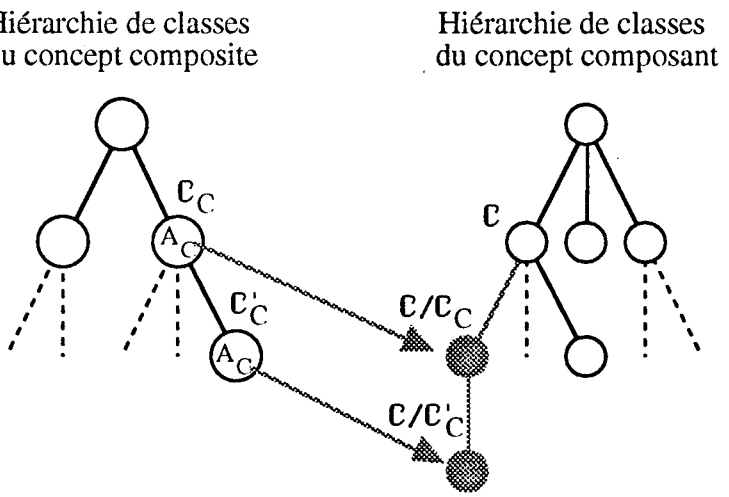

Figure VII.9 : Les trois premières équivalences concernant les relations de comparaisons entre accès de déclencheurs traduisent le fait que les affinements d'un attribut composant (A_C) dans certaines classes ($\mathbf{C}_C$ et $\mathbf{C}'_C$) du concept composite peuvent être considérés comme des classes virtuelles ($\mathbf{C}/\mathbf{C}_C$ et $\mathbf{C}/\mathbf{C}'_C$) du concept composant.

Cette perception de l'expression $\mathbf{C}/\mathbf{C}_C$ est cohérente car les valeurs que peut prendre l'attribut composant doivent appartenir à $\mathbf{C}$ et vérifier de plus les contraintes imposées par l'affinement de A_C dans $\mathbf{C}_C$.

La dernière équivalence est, quant à elle, introduite pour soutenir que la classe virtuelle C/C_C est une sur-classe de toutes les sous-classes de C. Cette assertion n'est en général pas fondée et peut s'avérer fausse : des instances appartenant aux sous-classes de C peuvent ne pas appartenir à la classe virtuelle C/C_C.

Pourquoi soutenir alors une assertion qui peut s'avérer fausse ?

L'intérêt de cette assertion est d'obtenir un ordonnancement des déclencheurs de l'attribut A du concept composant qui corresponde à la démarche de raffinement d'une instance du concept composite.

Le rattachement d'un objet composite à la classe C_C implique que son instance composante devrait appartenir nécessairement à la classe C, puis à la classe virtuelle C/C_C d'après l'affinement de l'attribut composant. Puisque la classe C/C_C n'a pas d'existence réelle dans le concept composant (elle n'existe qu'en tant qu'accès d'un déclencheur), la classe de rattachement effective de l'instance composante est la classe C. Il est cohérent de penser que la poursuite du raffinement de l'objet composite puisse permettre le raffinement de l'instance composante dans l'une des sous-classes de C. Du point de vue de l'objet composite, la classe virtuelle C/C_C est sous-classe de C et sur-classe *potentielle* de toutes les sous-classes de C (cf. Figure VII.10).

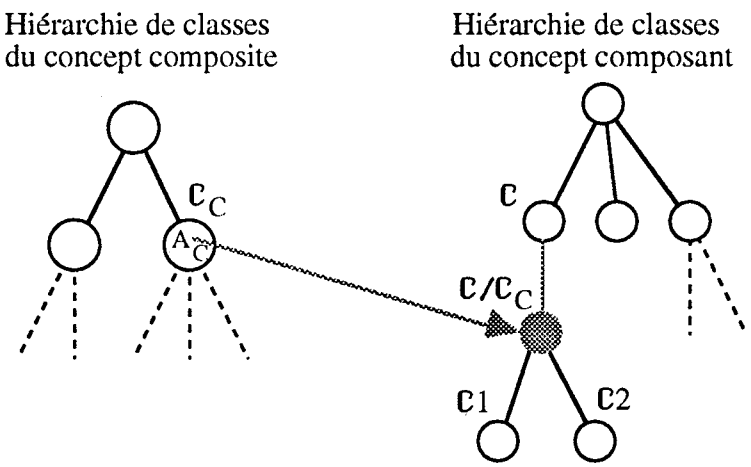

Figure VII.10 : La dernière équivalence concernant les relations de comparaisons entre accès de déclencheurs permet de supposer que la classe virtuelle (C/C_C) représentant l'affinement d'un attribut composant (A_C) dans une classe (C_C) du concept composite est une sur-classe de toutes les sous-classes (C1 et C2) de la classe (C) qu'elle spécialise dans le concept composant.

En soutenant cette assertion, il est alors possible d'établir un ordre de priorité entre le déclencheur d'accès C/C_C et tous les déclencheurs qui pourraient avoir pour accès une sous-classe de C. C'est-à-dire que l'ordre de priorité pour ce déclencheur est établi de la même façon que s'il avait eu simplement pour accès C.

En l'absence d'une telle assertion, il n'est pas possible d'établir un ordre de priorité entre le déclencheur d'accès C/C_C et tous les déclencheurs qui pourraient avoir pour accès une sous-classe de C. Dans ce cas, il est difficile de garantir le comportement correct du contrôle d'évaluation de A. Soit d=(C/C_C, Dom) un déclencheur de l'attribut A créé par l'affinement d'un attribut composant dans la classe C_C et soit d'=(C', S) un autre déclencheur de A tel que C' soit une sous-classe de C et S soit sa source de connaissances. Si d et d' ne sont pas comparables par $\rightarrow_A$, le contrôle d'évaluation peut faire face aux situations suivantes :

- si l'instance composite permet le raffinement de son instance composante dans la classe virtuelle C/C_C, puis dans la classe C', le déclencheur d sera activé et pourra donc être appliqué avant le déclencheur d'.

- si l'utilisateur indique à l'instance composite que son instance composante appartient à la classe C', puis raffine l'instance composite dans la classe virtuelle C/C_C, c'est le résultat contraire qui est observé : le déclencheur *d'* sera activé et pourra donc être appliqué avant le déclencheur *d*.

Si l'on suppose que le déclencheur *d'* décrit une inférence par facette défaut qui ne doit être appliquée qu'en dernier recours, le second cas de figure décrit un scénario non souhaitable. Inversement, si *d'* correspond à une inférence par réduction de domaine qui devrait être prioritaire sur toute autre inférence, le premier cas de figure décrit un scénario non souhaitable. L'assertion postulant que C/C_C est une sur-classe de C' permet de résoudre ce type de problème.

Que se passe-t-il lorsque l'assertion s'avère fausse ?

Dans ce cas, la situation se traduit par l'appartenance de l'instance composante à une sous-classe de C et la non-appartenance de l'instance composite à la classe C_C. Le seul problème à déplorer dans ce cas réside dans le fait que si le déclencheur qui a pour accès C/C_C est prioritaire sur un déclencheur ayant pour accès une sous-classe de C, le contrôle devra attendre que la classe C_C soit déclarée impossible pour l'objet composite avant de pouvoir appliquer le déclencheur d'inférence moins prioritaire. Toutefois, cet inconvénient devrait être atténué par le fait qu'il ne devrait pas apparaître souvent. En effet, savoir que le composant d'un objet composite appartient à une sous-classe de C dénote un état de connaissances de l'objet composite qui est généralement suffisant pour savoir qu'il n'appartient pas à la classe C_C.

Exemple du prix d'un moteur de voiture

La description de l'attribut *Prix* dans le concept *MOTEUR* fournit l'ensemble des déclencheurs $\mathfrak{Dl}(Prix)$ (cf. Figure VII.11).

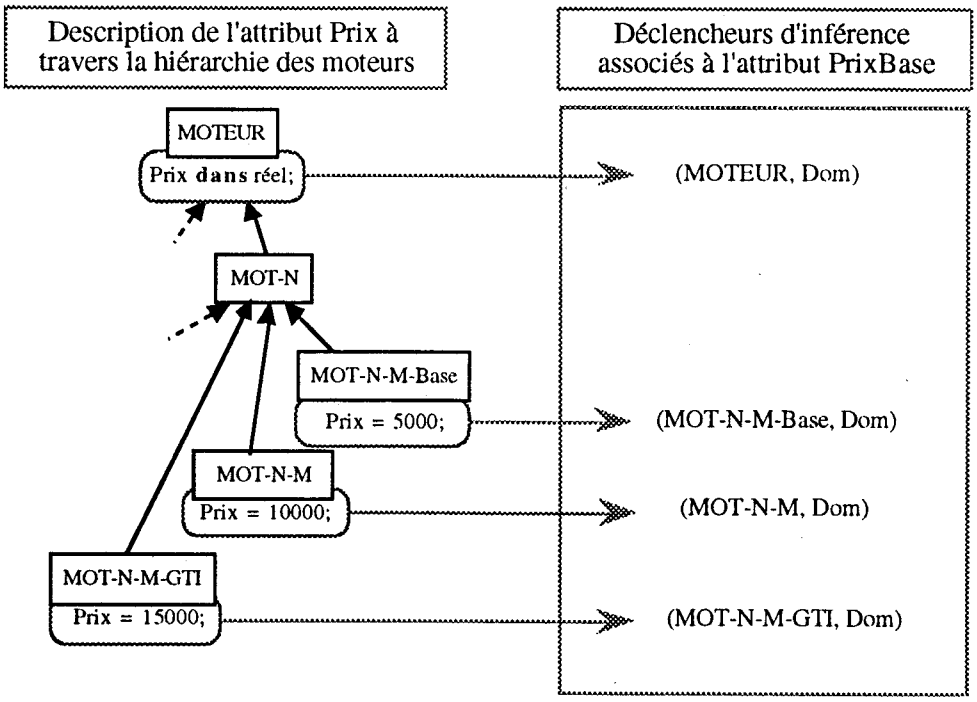

Figure VII.11 : Obtention de $\mathfrak{Dl}(Prix)$ à partir du concept *MOTEUR*.

Pour cet attribut, deux priorités différentes sont envisageables pour ordonner l'ensemble de ses déclencheurs d'inférence (cf. Figure VII.12). L'une favorise les moyens d'évaluation au niveau de raffinement le plus général (donc l'utilisateur), tandis que l'autre favorise les moyens d'évaluation aux niveaux de raffinement les plus spécialisés.

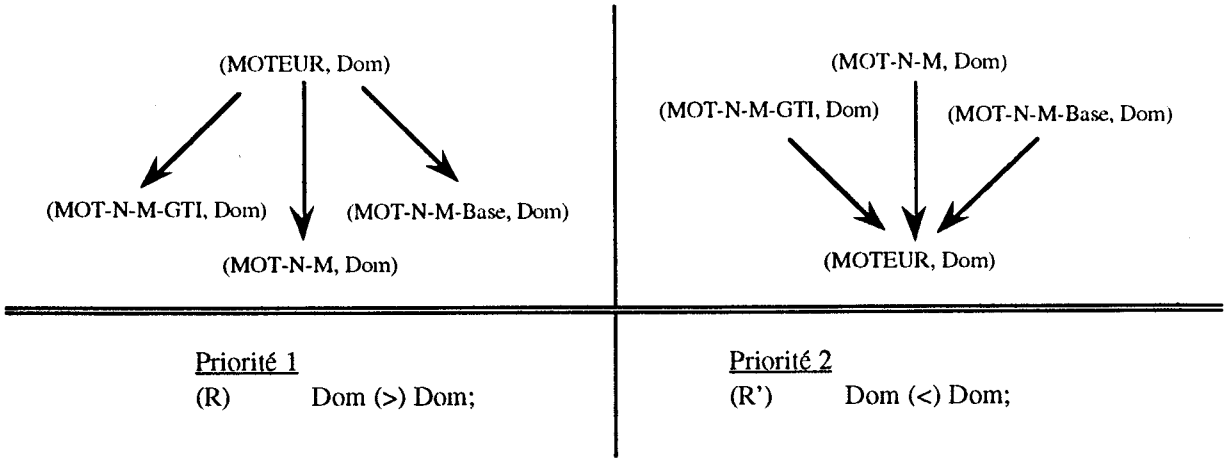

Figure VII.12 : Les deux priorités envisageables pour l'attribut *Prix* donne lieu à deux ordonnancements différents de $\mathfrak{Dl}(Prix)$.

Le concept composite *VOITURE* introduit l'attribut *Prix* du concept composant *MOTEUR* par l'affinement de l'attribut composant *Moteur* dans la classe *Marque-N*. Dans cet classe, l'attribut *Moteur* a pour type la classe *MOT-N* du concept *MOTEUR*. Par conséquent, un nouveau déclencheur de l'attribut *Prix* est créé : (*MOT-N / Marque-N*, Dom).

Ce déclencheur prend place avec les autres déclencheurs de $\mathfrak{Dl}(Prix)$ pour former la nouvelle stratégie d'évaluation de l'attribut *Prix* (cf. Figure VII.13). La priorité s'applique au nouveau déclencheur de la même façon que si sa classe d'accès avait été la classe *MOT-N*.

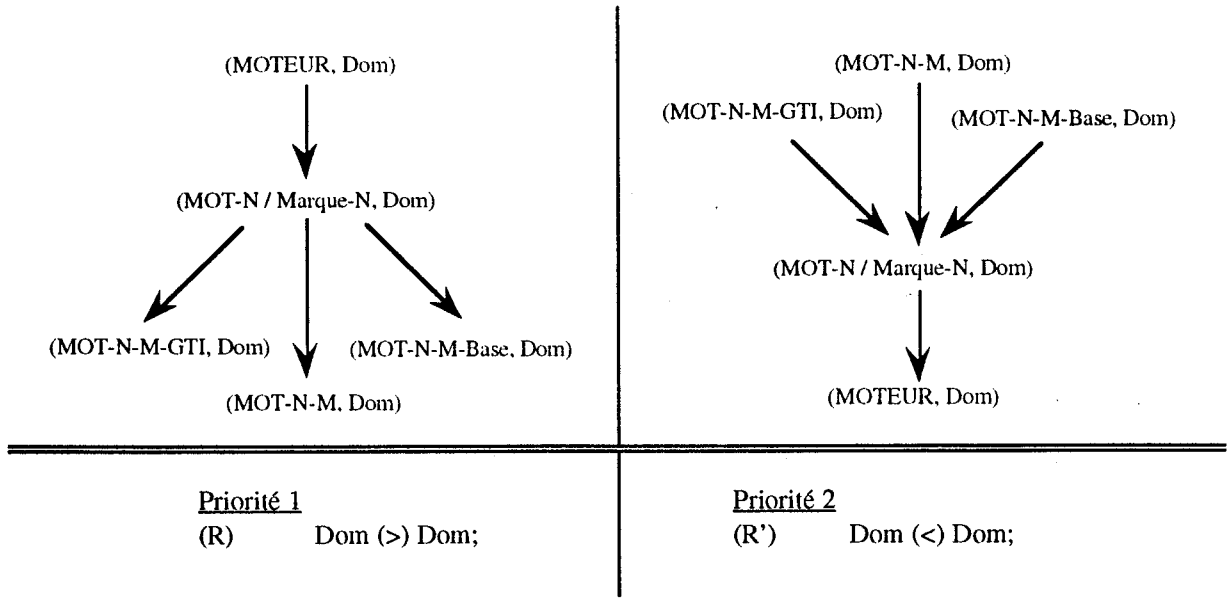

Figure VII.13 : Introduction dans $\mathfrak{Dl}(Prix)$ du déclencheur qui a été créé par le concept *VOITURE* en fonction des deux priorités possibles.

Soit *Ma-voiture* une instance du concept *VOITURE* qui est initialement configurée de la façon suivante :

- *Ma-voiture* ∈ *Marque-N*;
- *Ma-voiture.PrixHT*=40000 et *Ma-voiture.PrixBase*=35000;
- *Ma-voiture.Moteur* ∈ *MOT-N*.

On rappelle que dans la classe *Marque-N* l'attribut *Prix* du composant moteur est impliqué dans la contrainte suivante :

- *PrixHT = Moteur.Prix + PrixBase*

En fonction de la priorité associée à l'attribut *Prix*, cette configuration donnera le résultat suivant :

- Priorité 1 :

  Le déclencheur créé par le concept *VOITURE* est appliqué, la contrainte de la classe *Marque-N* permet l'inférence par réduction de l'attribut *Prix* du moteur.

  - *Ma-voiture.Moteur.Prix* = 5000

- Priorité 2 :

  Le déclencheur créé par le concept *VOITURE* est activé mais non prioritaire, l'évaluation de l'attribut *Prix* du composant n'a pas lieu. Pour qu'elle ait lieu, il faut que l'instance de moteur soit déclarée impossible pour toutes les sous-classes de *MOT-N*.

  Il faut noter que le domaine de l'attribut *Prix*, de par la contrainte dont il fait l'objet est réduit à la seule valeur 5000. Cette valeur rendra nécessairement impossible les classes *MOT-N-M* et *MOT-N-M-GTI* pour l'instance de moteur. En revanche, la classe *MOT-N-M-Base* reste possible.

Cet exemple montre comment la prise en compte de déclencheurs créés par un concept composite pour un attribut d'un concept composant permet d'adapter la stratégie d'évaluation de cet attribut au raffinement d'une instance composite. La partie suivante propose un moyen de préserver la description d'un concept composant des modifications ainsi apportées par un concept composite.

5.3.2. Concept composite et évaluation des attributs d'un composant

La stratégie d'évaluation d'un attribut A d'un concept composant Ω doit être étendue pour chaque attribut composant A^j_{iC} de chaque concept composite $\Omega^j{}_C$ dans lequel A subit les affinements de A^j_{iC}. Ainsi, pour chacun de ces cas, il faut envisager l'intégration de l'ensemble $\mathfrak{DL}(A, A^j_{iC}, \Omega^j{}_C)$ à la stratégie de A dans son concept de déclaration Ω. L'ensemble des déclencheurs considéré par la stratégie comprend alors aussi bien les déclencheurs propres au concept Ω que ceux qui ont pu être créés à partir des concepts composites $\Omega^j{}_C$.

Il est cependant évident que lorsqu'une instance du concept Ω exploite la stratégie d'évaluation de l'attribut A, seuls les déclencheurs correspondant à son contexte de création seront pertinents. Deux types de situations sont possibles :

- soit l'instance n'est pas un composant d'un objet composite, auquel cas seuls les déclencheurs de $\mathfrak{DL}(A)$ sont pertinents pour la stratégie de A ;
- soit l'instance constitue la valeur d'un attribut composant A_C d'un seul objet composite appartenant au concept composite Ω_C, auquel cas seuls les déclencheurs de $\mathfrak{DL}(A) \cup \mathfrak{DL}(A, A_C, \Omega_C)$ sont pertinents pour la stratégie d'évaluation de A.

Pour éviter de *polluer* la stratégie d'évaluation de l'attribut au sein de son concept de définition et de rendre ainsi plus lourde la gestion du contrôle, toute stratégie altérée par un concept composite intègre la description de ce concept. Ce principe de redéfinition locale de la stratégie d'un attribut au sein du concept qui la modifie permet de préserver l'indépendance du concept composant vis-à-vis des concepts composites qui peuvent l'exploiter. Cette redéfinition locale est mise en place :

- en associant à l'attribut A dans son concept de définition Ω uniquement sa stratégie interne, c'est-à-dire le couple $(\mathfrak{DL}(A), \rightarrow_A)$.

- en associant à chaque attribut composant A_C d'un concept composite Ω_C la description de la nouvelle stratégie ($\mathfrak{DI}(A) \cup \mathfrak{DI}(A, A_C, \Omega_C)$, $\rightarrow_A$) de l'attribut A quand celui-ci apparaît dans les affinements de A_C. La déclaration d'un attribut composant au niveau du concept composite contient une table associant la nouvelle stratégie à chaque attribut du composant qui a fait l'objet d'une redéfinition de stratégie.

Lorsqu'un attribut A apparaît dans une instance I, sa stratégie d'évaluation est obtenue de la façon suivante :

- si I constitue la valeur d'un attribut composant A_C du concept composite Ω_C, la stratégie d'évaluation de l'attribut A sera celle qui lui est associée dans la déclaration de A_C, ($\mathfrak{DI}(A) \cup \mathfrak{DI}(A, A_C, \Omega_C)$, $\rightarrow_A$), si elle existe, sinon celle définie dans son propre concept, ($\mathfrak{DI}(A)$, $\rightarrow_A$).
- si I ne constitue pas la valeur d'un attribut composant, la stratégie d'évaluation de A est celle définie dans son concept de définition, ($\mathfrak{DI}(A)$, $\rightarrow_A$).

Dans l'exemple de l'attribut *Prix* du concept *MOTEUR*, le principe de redéfinition locale s'applique donc de la façon suivante :

- Au niveau du concept *MOTEUR*, l'attribut *Prix* conserve sa stratégie initiale :

  ($\mathfrak{DI}(Prix)$, $\rightarrow_{Prix}$)
- Au niveau du concept *VOITURE*, l'attribut composant *Moteur* se voit associé dans sa déclaration l'association :

  ($Prix$, ($\mathfrak{DI}(Prix) \cup \{(MOT\text{-}N / Marque\text{-}N, \text{Dom})\}$, $\rightarrow_{Prix}$).

Lorsqu'un attribut composant est multi-valué, la création de déclencheurs pour un attribut de composant provient nécessairement d'un contexte d'éclatement de l'attribut composant. On tire alors parti du fait de l'ordre de déclaration des sous-attributs pour distinguer les différents cas de redéfinition de stratégie d'un attribut. Dans la déclaration de l'attribut composant, une association se présente alors ainsi :

- (A, *i*, S(A)) où A est l'attribut du composant concerné, S(A) la stratégie redéfinie et *i* la i-éme instance de l'attribut composant multi-valué qui est concernée par cette redéfinition de stratégie.

6. Conclusion

Ce chapitre présente un modèle permettant d'exprimer et de contrôler la stratégie d'évaluation d'un attribut en fonction du raffinement d'une instance. L'originalité de l'approche suivie réside dans le rôle accordé à la stratégie d'évaluation de chaque attribut dans la description des objets.

En effet, au même titre que les éléments de descriptions chargés de définir la structure des objets et le type de leurs attributs, la stratégie d'évaluation des attibuts est considérée comme une contrainte à laquelle doit se soumettre toutes les instances appartenant à un concept. C'est-à-dire que dès qu'un attribut apparaît dans une instance, sa valeur est nécessairement le résultat du développement de la stratégie qui lui est associée dans la description des objets. Quand un mécanisme d'inférence est déclenché, soit il fournit une valeur sûre de l'attribut (la cohérence de l'instance est mise en jeu), soit son exécution échoue (le contexte d'évaluation décrit par l'instance ne correpond pas à celui requis par l'inférence).

La stratégie associée à un attribut repose sur un recensement de toutes les possibilités d'évaluation que peut introduire la description d'un concept. Une telle possibilité se caractérise par le type d'inférence, plus généralement appelé la *source de connaissancès*, pouvant être à l'origine de l'évaluation et par les conditions que doit réunir nécessairement l'instance pour en bénéficier.

Trois sources de connaissances sont possibles pour un attribut : le détachement procédural, le domaine de valeurs d'attribut et le défaut. La source de type domaine prend en

compte l'utilisateur, les contraintes posées par un objet composite sur les attributs de ses composants et, enfin, les contraintes inter-attributs et les facettes de typage décrites dans les classes. En effet, chacun de ces intervenants peut participer à l'évaluation d'un attribut par le biais des contraintes ; c'est-à-dire par réduction du domaine de valeurs de l'attribut.

Le recensement des possibilités d'inférence d'un attribut consiste à repérer dans la description du concept chaque situation dans laquelle l'attribut peut faire appel à l'une de ces sources de connaissances. Une telle situation est représentée par la notion de *déclencheur d'inférence*. Le déclencheur précise le niveau de raffinement permettant à l'instance l'accès à cette situation et le type de source correspondant. L'utilisateur et le détachement procédural correspondent à des situations d'inférence accessibles à partir de la racine de la hiérarchie de spécialisation, tandis que les autres situations d'inférences dépendent de l'accès de l'instance aux classes dans lesquelles elles sont décrites.

La stratégie d'évaluation d'un attribut pouvant regrouper des situations d'inférences provenant de sources diverses et correspondant à des niveaux de raffinement différents, elle tire parti d'un ordre de priorité défini par le concepteur pour les organiser en un ordre partiel. Inspirée du principe de masquage appliqué dans les langages de *frames*, la priorité permet d'exprimer en fonction du type de source et du niveau d'accès l'ordre de déclenchement des inférences.

Contrairement au principe de masquage des langages de *frames*, la priorité est établie statiquement et permet d'obtenir un ordonnancement global des déclencheurs de chaque attribut. Cet ordonnancement fixe la stratégie d'évaluation d'un attribut pour toutes les instances du concept et se caractérise par la possibilité de prendre en compte aussi bien les situations d'inférence déjà accessibles par l'instance que celles qui ne le sont pas encore. Notamment, quand l'ordonnancement permet de rendre prioritaires des déclencheurs d'un attribut qui n'ont toujours pas été accédés par l'instance, l'évaluation de l'attribut dans l'instance est en attente d'informations supplémentaires sur le raffinement de l'instance.

A partir de ce modèle, le contrôle d'évaluation permet de distinguer différents états d'évaluation d'un attribut lors du raffinement d'une instance dans la hiérarchie de spécialisation. Par exemple, le contrôle permet de mettre en évidence le cas d'attributs pour lesquels l'évaluation ne peut plus être réalisée grâce au raffinement de l'instance. Dans ce cas, leur évaluation n'est plus possible sans un apport externe de connaissances à l'instance.

Enfin, le principe du contrôle d'évaluation des attributs est étendu au cas des objets composites. Il s'agit dans ce cas de prendre en compte au niveau des attributs d'un composant les nouvelles situations d'inférence qu'introduit la description du concept composite. Le principe consiste à redéfinir dans le concept composite la stratégie d'évaluation de chaque attribut de composants apparaîssant au niveau de la description du composite. Lors du raffinement d'une instance composite, un attribut d'un composant se réfère à la stratégie d'évaluation qui lui est associée dans le concept composite. Cette stratégie prend en compte aussi bien les possibilités d'inférence définies par le concept composant que celles introduites par le concept composite.

Le modèle de contrôle d'évaluation d'attributs proposé permet d'intégrer et de gérer des mécanismes d'évaluation qui ont des comportement forts différents, il est adapté au processus de raffinement d'instance dans la hiérarchie de classes et il permet d'impliquer la stratégie d'évaluation des attributs dans la définition cohérente des objets. Dans ces conditions, le modèle TROPES reste un système classificatoire.

Toutefois, la classification d'une instance est maintenant limitée par la stratégie d'évaluation décrite dans un concept pour chacun de ses attributs. En effet, dès qu'elle rencontre un attribut dont l'évaluation dépend de la poursuite du raffinement, la classification d'instance ne peut plus progresser. L'état de classification de l'instance est alors incomplet puisque la classification découvre des classes possibles ; c'est-à-dire des classes pour lesquelles l'appartenance de l'instance ne peut être ni prouvée, ni réfutée. La frontière délimitée par ces classes possibles ne peut être franchie par la classification. Elle a mis en évidence un problème d'identification/construction d'instance : l'instance est incomplète (un attribut au moins n'a pas de valeur) et son identité n'est pas complètement connue.

Le chapitre suivant (cf. §VIII) propose la mise en place d'un raisonnement hypothétique permettant de s'affranchir des limites de la classification. Dans ce contexte, le contrôle d'évaluation des attributs est exploitée pour définir en termes d'évaluation d'attributs le critère d'arrêt de l'exploration hypothétique d'une hiérarchie ; c'est-à-dire le problème d'identification/construction d'instance à résoudre.

Chapitre VIII
IDENTIFICATION/CONSTRUCTION D'INSTANCE PAR ASSISTANCE HYPOTHETIQUE

1. Introduction

L'objectif de ce chapitre est de proposer à un agent extérieur (utilisateur humain ou programmé) un support adéquat lui permettant d'exploiter la description des connaissances pour résoudre ses problèmes d'identification/construction d'instances. L'approche consiste à mettre en place un mécanisme, dit d'*assistance hypothétique*, permettant de décharger l'agent extérieur de la démarche d'exploration de la base de connaissances qu'il doit parfois entreprendre pour découvrir les différentes ressources qu'elle lui propose en réponse à ses problèmes.

Il convient dans un premier temps de rappeler la problématique générale à laquelle un mécanisme d'identification/construction d'instance entend répondre. A ces fins, un rapide rappel des limites des systèmes qui adoptent la classification d'instance comme support de construction d'instance (cf. §IV et §V) permet d'introduire le contexte de la nouvelle approche (cf. §VIII.2.1). Selon le schéma classificatoire adopté, ces limites se traduisent soit par des contraintes de construction imposées aux hiérarchies d'objets (cf. §VIII.2.1.1), soit par une situation paradoxale dans laquelle les rôles sont inversées : l'utilisateur doit résoudre le problème de construction d'instance que lui pose le système (cf. §VIII.2.1.2).

La solution adoptée constitue une extension des systèmes classificatoires de la seconde catégorie (cf. §VIII.2.2). Son principe consiste à offrir à l'utilisateur un mécanisme d'assistance à partir duquel il peut explicitement formuler le problème qu'il veut résoudre (cf. §VIII.2.2.1). Le problème étant défini, le rôle de l'assistance est d'explorer et de proposer à l'utilisateur les différentes solutions que permet de construire la description des objets (cf. 2.2.2).

1.1. Identification/construction et classification d'instance

La principale difficulté à laquelle se heurtent généralement les systèmes qui proposent une solution au problème d'identification/construction d'instance, provient de la façon même de l'aborder. L'approche suivie par ces systèmes consiste à ramener ce problème à celui de la classification d'une instance incomplète dans une hiérarchie d'objets (cf. §IV.4). Puisque la classification d'une instance relève d'un raisonnement de type déductif, son exploitation dans le cas d'une instance incomplète est limitée.

La construction d'une instance par classification est abordée en distinguant deux types de systèmes : ceux qui permettent de conjuguer classification et inférence de valeurs d'attributs et ceux qui ne le permettent pas. Dans le premier cas, la résolution d'un problème de construction d'instance est contrôlée complètement par le mécanisme de classification. Dans le second cas, l'utilisateur doit mener sa propre démarche d'identification/construction d'instance en cherchant à résoudre les problèmes soulevés par la classification de l'instance.

1.1.1. Systèmes proposant une résolution par la classification

La stratégie que ces systèmes proposent pour la résolution d'un problème d'identification/construction d'instance consiste à identifier un problème particulier par rapport à un ensemble de types de problèmes pour lesquels une méthode de résolution est connue. L'identification du problème est basée sur la confrontation de ses données d'entrée avec la description des types de problème. Une fois que le type du problème posé est identifié, la solution peut être construite en employant la méthode de résolution fournie par ce type.

1.1.1.1. Contraintes imposées à la description d'une hiérarchie de classes

Pour mettre en place la construction d'une instance selon ce principe, ces systèmes doivent autoriser le déclenchement d'inférences d'attributs au cours de la classification. Afin de garantir la continuité du processus classificatoire, l'organisation d'une hiérarchie de classes doit obéir à des contraintes strictes.

Disposant d'un mécanisme de classification général et bien défini, le système FROME s'impose à ce titre comme un modèle de référence (cf. §IV.4.2.2). Les conditions de déclenchement d'inférence lors de la classification d'une instance y sont précisément définies. En effet, le principe adopté par ce système consiste à distinguer dans les descriptions de classes divers types de connaissances : celles pour l'identification (attributs déterminants sur lesquels repose l'appartenance à une classe) et celles pour la production (attributs non déterminants dont la valeur peut être déduite lors de la classification).

Il apparaît donc que deux contraintes sont imposées à la description d'une hiérarchie d'objets pour qu'elle puisse supporter un raisonnement de type identification/construction :

- **contrainte horizontale** : chaque classe doit rendre compte de sa participation au processus global en indiquant les attributs qu'elle nécessite en entrée et ceux qu'elle propose éventuellement en sortie.
- **contrainte verticale** : la hiérarchie de classes doit être bâtie de telle façon que l'enchaînement des inférences assure la progression du processus de classification.

Il faut noter que de tous les systèmes qui cherchent à mettre en place un raisonnement d'identification/construction d'instance à partir du mécanisme de classification, FROME est le seul à poser clairement les contraintes que cette approche impose à la description des objets. Toutefois, les autres systèmes de ce type (SHOOD, SHIRKA, etc.) imposent implicitement les mêmes contraintes aux hiérarchies de classes.

1.1.1.2. Notions de problèmes et de solutions

Dans cette approche, la notion de problème est représentée par l'ensemble des attributs d'une classe désignés comme entrées, tandis que la notion de solution est décrite par l'ensemble des attributs désignés comme sorties. Ces deux notions, liées par la structure de la classe, sont indissociables. Par conséquent, les hiérarchies sont construites en figeant les objets dans une représentation orientée par les notions d'entrées et de sorties. En d'autres termes, une hiérarchie d'objets est spécialisée dans la résolution d'une catégorie précise de problèmes.

Lorsqu'un utilisateur veut résoudre un problème d'identification/construction, il lui suffit de lancer la classification d'une instance à partir de la hiérarchie de classes correspondant au type de problème qu'il veut résoudre. Le système amène l'utilisateur à spécifier son problème particulier en réclamant les valeurs des attributs désignés comme entrées. Si la classification de l'instance permet d'évaluer tous les attributs de sortie de l'instance, l'ensemble de ces attributs constitue la solution produite par le système.

En revanche, si l'utilisateur s'avère incapable d'évaluer tous les attributs attendus en entrée, le système ne peut résoudre le problème. Le type de problème soumis par l'utilisateur ne coïncide avec aucun des types de problèmes décrits par la hiérarchie d'objets.

1.1.1.3. Cas d'un problème à solutions multiples

L'approche par classification ne permet pas de prendre en compte la résolution de problème à solutions multiples. En effet, cette approche aborde généralement le problème d'identification/construction d'instance par la mise en œuvre d'un raisonnement déductif dirigé par les données (cf. §I.1.3.2). La construction d'une solution est dépendante du résultat de l'exploration de la hiérarchie de classes que permettent les données caractérisant le problème. Or, pour traiter des problèmes à solutions multiples, il faut d'une part pouvoir représenter la pluralité des solutions et d'autre part permettre au raisonnement de développer les différentes solutions.

Le système SHOOD propose dans ce sens une adaptation originale du mécanisme de classification (cf. §IV.4.2.1). Dans cette approche, le développement de plusieurs solutions est motivé par la détection de conflits d'inférences d'attributs lors de la classification. Un conflit d'inférence intervient quand un attribut déjà évalué par une inférence fait l'objet d'une nouvelle inférence. La résolution du conflit consiste à créer une copie de l'instance dans laquelle la nouvelle valeur inférée peut prendre place.

Dans cette approche, la notion de conflit d'inférence est implicitement utilisée pour décrire une relation de disjonction entre deux classes de la hiérarchie. Si la classification détecte un conflit, la création d'une copie vise à rétablir la cohérence en proposant que chaque classe en disjonction possède sa propre version de l'instance. Dans ces conditions, le système peut donc générer plusieurs solutions pour un même problème.

L'intérêt de cette approche est d'étendre le raisonnement classificatoire vers un raisonnement hypothétique dans lequel plusieurs possibilités de raffinement peuvent être envisagées simultanément. Toutefois, dans le cadre de SHOOD, ce raisonnement est défini spécialement pour résoudre les situations de conflit d'inférence et peut remettre en cause la sémantique même de la spécialisation de classes. En effet, une situation de conflit pouvant apparaître lors du raffinement de l'instance d'une classe vers une de ses sous-classes, la résolution de cette situation revient à affirmer que cette classe et sa sous-classe sont disjointes.

1.1.2. Systèmes proposant une résolution par l'utilisateur

Un système qui propose un mécanisme général de classification d'instance sans imposer de contraintes particulières à la description des hiérarchies d'objets ne peut exploiter les inférences de valeurs d'attributs lors du processus de classification (cf. §IV.2). Dans ce cas, la classification s'avère pour l'utilisateur un support de construction d'instance dont la tâche consiste essentiellement à dévoiler progressivement la structure de l'instance. Durant ce processus, que le modèle permette ou non l'expression de connaissances de production dans la description des objets, l'évaluation des attributs reste à la charge de l'utilisateur. Les limites de cette approche sont atteintes quand l'utilisateur ne possède pas toutes les informations nécessaires à une étape de raffinement, la classification de l'instance s'arrête.

L'arrêt de la classification dans de telles conditions prive l'utilisateur du support de construction d'instance que lui offrait le système. En effet, le système se trouve alors bloqué puisque, d'une part, il ne peut poursuivre le raffinement du fait de l'état incomplet de l'instance et, d'autre part, le niveau de raffinement de l'instance ne lui donne pas accès au connaissances de production nécessaires pour la compléter. Seule l'intervention de l'utilisateur peut permettre de débloquer cette situation.

Le problème auquel est confronté l'utilisateur se traduit par un ensemble d'attributs de l'instance dont les valeurs restent inconnues et un ensemble de classes déclarées possibles par la classification. Puisque l'utilisateur n'a pu évaluer lui-même les attributs lors de la classification, son intervention consiste à désigner parmi les classes possibles celle susceptible de fournir un moyen d'évaluer les attributs inconnus. C'est-à-dire qu'il doit alors procéder lui-même à l'exploration de la hiérarchie de classes pour trouver une nouvelle classe de rattachement pour l'instance.

La tâche d'exploration que doit mener l'utilisateur est ardue puisque, pour fixer son choix de rattachement de l'instance, il doit pouvoir évaluer préalablement la pertinence de chaque classe explorée en confrontant sa description à l'ensemble des attributs qu'il doit ou veut évaluer. De plus, si plusieurs solutions sont envisageables et s'il veut pouvoir les comparer, il devra les essayer une à une.

Dans cette approche, le problème soumis à l'utilisateur correspond à un problème général d'identification/construction d'instance. La suite de ce chapitre se propose de mettre en place un système d'assistance permettant à l'utilisateur de se décharger de ce genre de problème.

1.2. Nouvelle approche du problème

Le principe de base de cette nouvelle approche consiste à offrir à l'utilisateur la possibilité d'exprimer son problème de construction d'instance sous la forme d'une requête adressée au système d'*assistance hypothétique*. Dans cette requête, le problème est précisé en indiquant les informations de l'instance que l'utilisateur désire connaître : *les buts à atteindre*.

1.2.1. Définition d'un problème d'identification/construction

Un problème d'identification/construction d'instance apparaît quand un utilisateur s'interroge sur la façon d'obtenir la valeur de certains attributs de l'instance. L'objectif de l'utilisateur est donc l'évaluation de certains attributs et son problème est de découvrir la ou les *configurations* de l'instance susceptibles d'atteindre cet objectif. Le terme de *configuration* fait ici référence à l'ensemble des connaissances que l'utilisateur peut communiquer au système de raisonnement à propos d'une instance. Ces connaissances sont fournies soit directement lors de la création de l'instance, soit indirectement lors de demandes d'informations complémentaires formulées par le système.

A partir du système de raisonnement, l'utilisateur ne dispose ni d'un moyen d'exprimer le but qui motive sa démarche de configuration, ni d'un mécanisme lui permettant d'explorer plusieurs configurations de la même instance. A cet effet, un *système d'assistance hypothétique* est couplé au système de raisonnement afin que l'utilisateur puisse se décharger d'une partie de la configuration de l'instance qu'il ne maîtrise pas.

Le système d'assistance hypothétique doit offrir à l'utilisateur le moyen d'exprimer son problème et le moyen de le résoudre. Vis-à-vis du système de raisonnement, le système d'assistance joue le même rôle que l'utilisateur, il cherche à configurer l'instance. Il possède, de plus, la capacité de gérer plusieurs versions de la même instance lorsque la recherche de solutions le mène à considérer diverses possibilités de configuration.

1.2.1.1. Description d'un problème

Pour spécifier un problème d'identification/construction d'instance, l'utilisateur doit fournir les deux informations suivantes au système d'assistance :

- une instance de la base de connaissances partiellement configurée ;
- la description de la *contrainte de productivité* que l'instance doit satisfaire.

La notion de *contrainte de productivité* est introduite pour décrire en termes d'évaluation d'attributs de l'instance les conditions que toute démarche de configuration doit continuellement vérifier et finalement satisfaire. Une telle contrainte est constituée par l'ensemble des attributs de l'instance que le système d'assistance doit chercher à évaluer. Ces attributs sont appelés les **buts** du système d'assistance.

Le recours au terme de contrainte s'explique par les nuances suivantes :

- une contrainte de productivité est **satisfaite** quand les attributs décrits comme buts sont effectivement évalués.
- une contrainte de productivité est **insatisfaisable** lorsque l'un au moins des attributs désignés comme buts ne peut plus être évalué par le système de raisonnement (le système a épuisé tous les moyens possibles d'évaluation, le contrôle est donc dans un état d'échec total).

1.2.1.2. Caractérisation d'une solution

Pour obtenir une solution à un problème, le système d'assistance doit mettre en place un mécanisme visant à explorer les différentes possibilités de configuration de l'instance que la description des objets permet d'envisager. Chaque possibilité de configuration donne lieu à la création d'une version de l'instance pour laquelle le système vérifie la contrainte de productivité.

Dans ces conditions, une version de l'instance constitue une solution particulière à un problème d'identification/construction si et seulement si elle vérifie les conditions suivantes :

- elle est cohérente ;
- elle satisfait la contrainte de productivité posée.

Résoudre un problème d'identification/construction d'une instance consite pour le système d'assistance a exploré toutes les possibilités de configuration de l'instance. La solution à ce problème est l'ensemble des versions de l'instance qui en sont des solutions possibles ; c'est-à-dire celles qui sont cohérentes et satisfont la contrainte de productivité décrivant le problème.

1.2.1.3. Exemple de problème d'identification/construction d'instance

Appliquée à la base de connaissances des voitures (cf. §VII.2.2), la notion de contrainte de productivité offre au vendeur la possibilité d'exploiter le système d'assistance pour obtenir des réponses à des interrogations comme la suivante :

"Quelle(s) voiture(s) puis-je proposer à mon client sachant qu'il veut que sa voiture soit française, à essence,… et que son budget ne dépasse pas les 45000 ?"

Pour ce faire, le vendeur doit préciser à quelles conditions il estime qu'une proposition du système constitue une solution à sa requête. Il reformule donc cette interrogation en un problème de configuration d'instance de voiture pour lequel il spécifie une configuration initiale à valider et un ensemble de buts à satisfaire.

Par le choix des buts, le vendeur fixe la contrainte de productivité que doit satisfaire au minimum une instance pour être considérée comme une solution. Par exemple, s'il juge que l'obtention du prix TTC s'avère un critère de sélection suffisamment précis pour caractériser le terme "voiture" utilisé dans sa question, il désignera l'attribut PrixTTC comme but. La requête du vendeur peut alors être formulée de la façon suivante :

Rechercher les configurations possibles de l'instance voiture telles que :

• **[Déclaration de l'ensemble des buts]** :

{PrixTTC} ;

• **[Configuration initiale de l'instance]** :

{(voiture *est-un* Française), (type="essence"),… , (PrixTTC ≤ 45000)}

Le choix de PrixTTC comme but se justifie dans cet exemple par le fait qu'il faut que le système possède déjà une connaissance approfondie d'une instance de voiture pour pouvoir en évaluer le prix. Toutefois, il convient de souligner que cette contrainte de productivité n'assure en rien que les versions d'instance obtenues soient complètes (des attributs peuvent rester à valeur inconnue). En effet, seule la découverte des informations minimales permettant de supporter l'évaluation du prix TTC est imposée par cette contrainte de productivité. Ainsi, un attribut qui n'interviendrait pas dans l'évaluation de ce prix, comme l'attribut Couleur par exemple, peut rester inconnu. Cela permet au vendeur de faire abstraction d'informations qu'il ne considère pas pertinentes dans le choix d'une voiture.

Par contre, les informations, comme le prix (et plus généralement le modèle) du moteur, le prix de base, l'appartenance de la voiture à une marque, à un modèle et à une série, etc., sont des informations qu'il faudra nécessairement obtenir pour permettre l'évaluation du prix TTC.

1.2.2. Résolution par recherche de configurations possibles

La situation à laquelle doit faire face le système d'assistance hypothétique est celle d'un utilisateur bloqué à une étape de raffinement ; celui-ci n'étant plus en mesure de fournir d'informations concernant l'évaluation des attributs restant inconnus. Les différentes combinaisons de connaissances que peut et doit essayer le système d'assistance sont fixées par les différentes possibilités d'évolution de l'instance dans la description des objets.

En explorant systématiquement cette description, le système d'assistance peut alors constituer progressivement toutes les combinaisons de connaissances de configuration existantes, vérifier leur cohérence et s'assurer de leur pertinence grâce à la contrainte de productivité. La construction incrémentale des différentes solutions, c'est-à-dire des versions solutions de l'instance, est ainsi assurée lors de cette exploration.

1.2.2.1. Mise en place incrémentale de la résolution

Le processus de construction des solutions se décompose en une succession d'étapes de configuration. Chaque étape de configuration a pour objectif de répondre aux problèmes soulevés par le raisonnement quand il ne peut poursuivre le raffinement de l'instance. Les raisons d'arrêt du raffinement dans un tel contexte classificatoire peuvent être de deux types :

- Plusieurs moyens d'évaluation sont déclenchables pour un attribut. Le système de raisonnement ne disposant pas d'information permettant de choisir l'un d'eux, il reste indécis.
- L'évaluation d'un attribut ne peut être assurée dans le contexte de raffinement courant. Pour tenter d'évaluer cet attribut, le système de raisonnement peut nécessiter de nouvelles connaissances concernant le raffinement de l'instance. Son indécision porte donc cette fois-ci sur le choix de raffinement de l'instance.

Le système d'assistance peut remédier à ces situations en considérant les différentes hypothèses envisageables pour chaque cas. Ces hypothèses représentent, selon le cas, les différents choix de moyens d'évaluation d'un attribut ou les différents états de raffinement pour l'instance. En les combinant, de nouvelles étapes de configuration à explorer et à valider sont obtenues.

Pour déterminer parmi ces nouvelles étapes de configuration celles qui décrivent effectivement une solution au problème initial, le système d'assistance doit vérifier si l'application des connaissances rassemblées par chacune d'elles n'introduit pas d'incohérence dans l'instance (vérification de la **cohérence**) et si, de plus, elle entraîne l'évaluation des attributs de l'instance désignés comme buts (**satisfaction** de la contrainte de productivité).

Pour effectuer cette double vérification, le système d'assistance propose les configurations possibles une à une au système de raisonnement. Pour préserver et distinguer les différentes solutions entre elles, l'application de chaque configuration est mise en place sur une version différente de l'instance. Une telle version de l'instance est alors dite **hypothétique**.

Le système de raisonnement procède sur une version hypothétique comme il le fait pour n'importe quelle instance, il lui applique la configuration proposée et déduit de nouvelles informations. L'état de la version résultante permet au système d'assistance de bénéficier à la fois de l'état de cohérence de l'instance et de l'état d'évaluation des attributs de l'instance.

Si une version de l'instance ne contredit pas et ne satisfait pas la contrainte de productivité, elle devient un nouveau problème à résoudre et le système d'assistance poursuit la résolution à partir d'elle. Si la version satisfait la contrainte de productivité, elle constitue une solution au problème initial et vient enrichir l'ensemble des versions solutions. Enfin, si une version ne peut plus satisfaire la contrainte de productivité ou s'avère incohérente, elle est abandonnée.

La constitution d'une configuration menant à une solution peut passer par les créations successives de plusieurs versions hypothétiques. Chaque version de cette séquence se distingue de celle qui la précède par les connaissances supplémentaires, c'est-à-dire les hypothèses que le système d'assistance a dû apporter. Le système d'assistance développe ainsi chaque voie de configuration (cf. Figure VIII.1) qu'il peut constituer jusqu'à ce qu'il ne soit plus en mesure de proposer de nouvelles connaissances de configurations, ou que chaque voie ait abouti soit à une incohérence, soit à une situation dans laquelle la contrainte de productivité est insatisfaisable, soit à une solution.

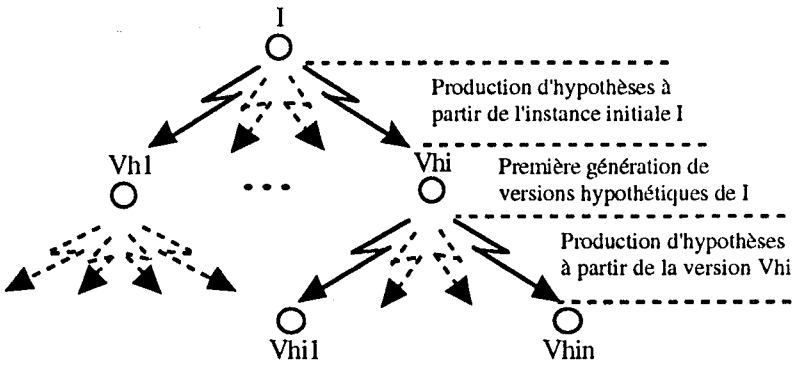

Figure VIII.1 : Exploration des possibilités de configuration d'une instance. Après avoir vérifié la satisfaisabilité de la contrainte de productivité sur une version, le système d'assistance construit les prochaines étapes de configurations à partir des hypothèses que l'état de raffinement de la version permet d'envisager. La combinaison des différentes hypothèses ainsi émises fournit au système d'assistance les prochaines étapes de configuration à entreprendre. Celles-ci prennent place dans de nouvelles versions hypothétiques.

L'étape sur laquelle repose le succès de cette démarche est celle de la production et de la combinaison des hypothèses. En effet, une combinaison d'hypothèses constitue un pas élémentaire d'exploration de l'espace de solutions. Ce pas doit donc être suffisamment fin pour qu'aucune solution ne soit oubliée. Pour ce faire, le système d'assistance doit gérer séparément l'exploitation des deux types d'hypothèses dont il peut disposer : les hypothèses concernant le raffinement de l'instance ou les hypothèses concernant l'évaluation des attributs.

1.2.2.2. Etape de configuration et combinaisons d'hypothèses

Lors de l'exploration des solutions, le système d'assistance est amené à considérer la production d'hypothèses suivant deux axes possibles (cf. Figure VIII.2).

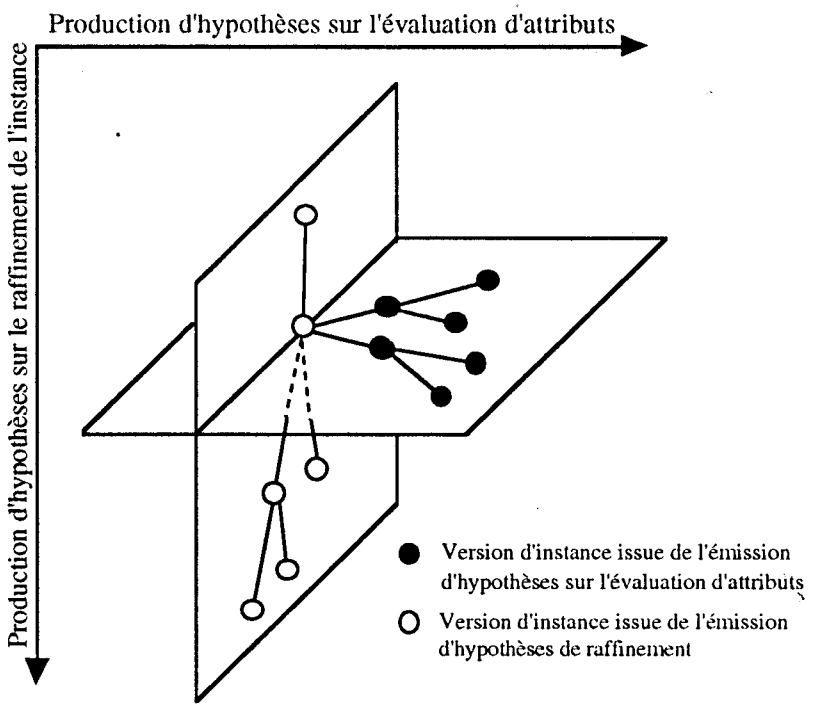

Figure VIII.2 : Le système d'assistance explore toutes les évolutions possibles d'une version. Ces évolutions peuvent être le fruit de deux types de production d'hypothèses.

Chacun de ces axes correspond à un type d'hypothèses différent que le système d'assistance exploite pour progresser vers les solutions : hypothèses concernant l'évaluation des attributs ou hypothèses concernant le raffinement de l'instance. Le système d'assistance fait appel à ces deux types d'hypothèses pour deux raisons différentes.

Les hypothèses sur l'évaluation d'un attribut visent autant à explorer les différentes possibilités d'évaluation d'un attribut dans l'état actuel de raffinement qu'à garantir la cohérence de la voie d'exploration suivie par le système d'assistance. En effet, lorsqu'un attribut fait l'objet d'un choix de moyens d'évaluation, il est crucial de s'assurer qu'il en existe au moins un dont l'application n'introduira pas d'incohérence dans l'instance. Lors de l'exploration systématique des différentes hypothèses d'évaluation, l'objectif du système d'assistance consiste donc à s'assurer que la voie d'exploration en cours de développement est cohérente. Lorsqu'un attribut est aussi un but à atteindre, l'objectif de cette production est alors double : la cohérence et l'évaluation du but.

Les hypothèses sur le raffinement permettent, quant à elles, au système d'assistance de découvrir de nouveaux contextes d'évaluation quand l'état courant de l'instance ne permet pas l'évaluation des attributs désignés comme buts. Ce type d'hypothèses peut introduire de nouvelles contraintes sur l'instance qui influent sur sa cohérence.

Du point de vue de la résolution, la prise en compte simultanée d'hypothèses sur le raffinement et sur l'évaluation d'attributs peut avoir de fâcheuses conséquences. En effet, si les hypothèses sur l'évaluation des attributs permettent de satisfaire la contrainte de productivité à elles seules, les hypothèses de raffinement peuvent en contre-partie remettre en cause la solution par les nouvelles contraintes qu'elles peuvent introduire sur l'instance.

Aussi, avant d'envisager les hypothèses de raffinement, le système d'assistance doit donc établir si des combinaisons d'hypothèses concernant l'évaluation des attributs peuvent mener soit à une solution, soit à une incohérence. Lors de la constitution de nouvelles étapes de configuration, le système d'assistance adopte donc la stratégie suivante :

- s'il est possible de produire à partir de la version d'instance courante des hypothèses concernant l'évaluation de certains attributs, alors les prochaines étapes de configuration à explorer sont formées par l'ensemble des combinaisons possibles de ces hypothèses.
- sinon les prochaines étapes de configuration à explorer correspondent aux différentes hypothèses de raffinement.

Il explore ainsi toutes les situations de configuration possibles en alternant des phases de production de chaque type d'hypothèses (cf. Figure VIII.3).

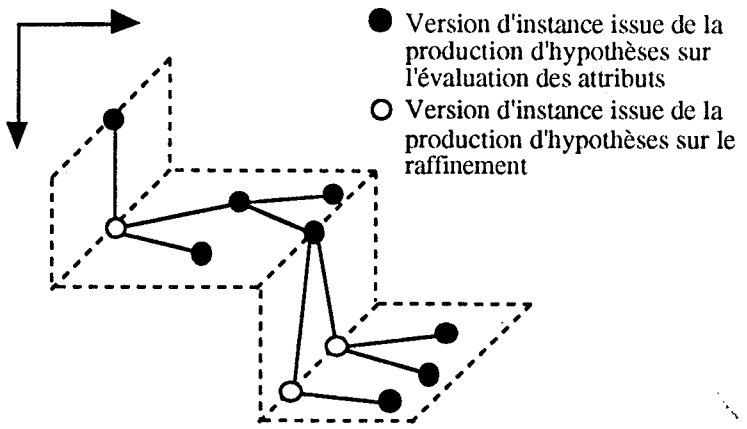

Figure VIII.3 : Le système d'assistance explore en priorité les combinaisons d'hypothèses sur l'évaluation des attributs, repoussant la prise en compte des hypothèses de raffinement aux niveaux des versions résultantes qui ne satisfont toujours pas la contrainte de productivité et ne sont pas incohérentes.

Cette exploration peut être étendue en fonction du critère de cohérence que l'on souhaite appliquer.

1.2.2.3. Critères de cohérence

D'après la définition d'une version d'instance solution, seule l'évaluation des attributs désignés comme buts est imposée. Toutefois, la cohérence de la version d'instance étant aussi requise, elle constitue aussi une contrainte que l'ensemble des informations de la version doit satisfaire. Deux niveaux de cohérence peuvent être distingués pour caractériser la notion de version d'instance solution :

- **Cohérence faible** : la version est **faiblement** cohérente si l'ensemble des informations qu'elle contient forme un tout cohérent ; c'est-à-dire qu'aucune incohérence n'a été détectée lors de la configuration.
- **Cohérence forte** : la version est **fortement** cohérente s'il est possible de compléter sa configuration afin d'atteindre un état complet et cohérent de l'instance. Elle implique la cohérence faible.

L'exploitation du critère de cohérence faible permet uniquement de garantir que toute version solution obtenue représente un état de l'instance ne contenant aucune incohérence lorsqu'elle satisfait la contrainte de productivité. Une version d'instance solution pouvant correspondre à un état incomplet d'instance (seule l'évaluation des attributs désignés comme buts étant requise), sa cohérence reste toutefois conditionnée par l'existence d'au moins une configuration permettant de la compléter.

Le critère de cohérence forte vise à étendre le problème de cohérence à celui des évolutions possibles d'une version. Deux problèmes de cohérence peuvent se poser à une instance lorsque l'on envisage ses évolutions possibles :

- L'évaluation de certains attributs reste dépendante de la sélection d'un moyen d'évaluation ou de l'évaluation d'autres attributs. Pour chacun de ces attributs, l'obtention d'une valeur peut être source d'incohérence. Puisque leur évaluation est dépendante du raffinement de l'instance, peut-on garantir qu'il existe un raffinement de l'instance permettant d'écarter tout problème de cohérence ?
- La cohérence de l'état de classification est parfois conditionnée par la satisfaction de propriétés particulières de la hiérarchie. Si la classe de rattachement courant possède une propriété d'exclusivité ou d'exhaustivité, comment être sûr qu'elle pourra être satisfaite ?

La cohérence forte d'une version peut parfois n'être ni établie, ni réfutée. En effet, une version vérifiant à la fois la contrainte de productivité et la contrainte de cohérence faible peut très bien posséder les caractéristiques suivantes :

- Elle est terminale pour l'exploration ; plus aucune hypothèse ne pouvant être émise au vu de son état.
- Certains attributs sont dans un état d'attente du résultat d'une inférence (état d'évaluation demandeur).

Dans ce cas, la satisfaction de la contrainte de cohérence forte est dépendante du résultat délivré par les inférences en attente d'exécution. Puisque la version est terminale pour l'exploration, l'aboutissement de ces inférences est dépendant d'une intervention de l'utilisateur. La version est potentiellement une solution selon le critère de cohérence forte.

L'application du critère de cohérence forte peut être vue comme une extension de la recherche des solutions faiblement cohérentes. Cette extension de l'exploration de configurations au delà de la recherche de versions solutions permet d'obtenir pour chacune d'elles l'ensemble des versions qui satisfont complètement ou potentiellement la contrainte de cohérence forte. A l'issue de cette exploration, toute version solution pour laquelle chaque extension s'est achevée sur un état incohérent de l'instance peut être écartée de l'ensemble des

solutions. Dans la suite, seul le critère de cohérence faible est appliquée lors de la recherche de solution.

Avant de présenter en détail la mise en place du mécanisme d'identification/construction d'instance, la partie suivante présente l'intégration de l'assistance hypothétique au sein du système TROPES.

2. Principe d'intégration de l'assistance hypothétique

Dans TROPES, excepté lorsqu'un utilisateur modifie directement la base de connaissances, un raisonnement correspond à une suite d'inférences déductives qui viennent enrichir la base de connaissances (cf. Figure VIII.4).

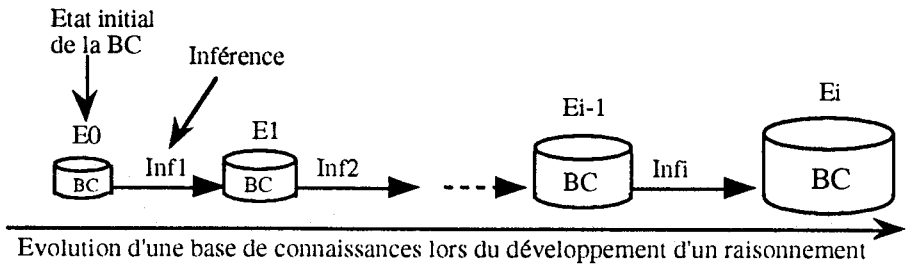

Figure VIII.4 : Chaque inférence réalisée par le moteur d'inférence est monotone. Les connaissances ainsi déduites enrichissent la base de connaissances.

La mise en place de l'assistance hypothétique nécessite d'étendre le modèle TROPES afin d'offrir, d'une part, des mécanismes capables d'exploiter une situation de connaissances incomplètes pour proposer un ensemble de réponses possibles — des **hypothèses** — et, d'autre part, les moyens de gérer l'introduction de ces connaissances hypothétiques au sein de la même base de connaissances.

Le résultat d'une telle extension est donc la possibilité de faire connaître à un raisonnement plusieurs développements possibles. Chaque phase d'introduction d'hypothèses, appelée phase d'**assistance hypothétique**, peut être vue comme la mise en évidence de plusieurs évolutions possibles de la base de connaissances (cf. Figure VIII.5).

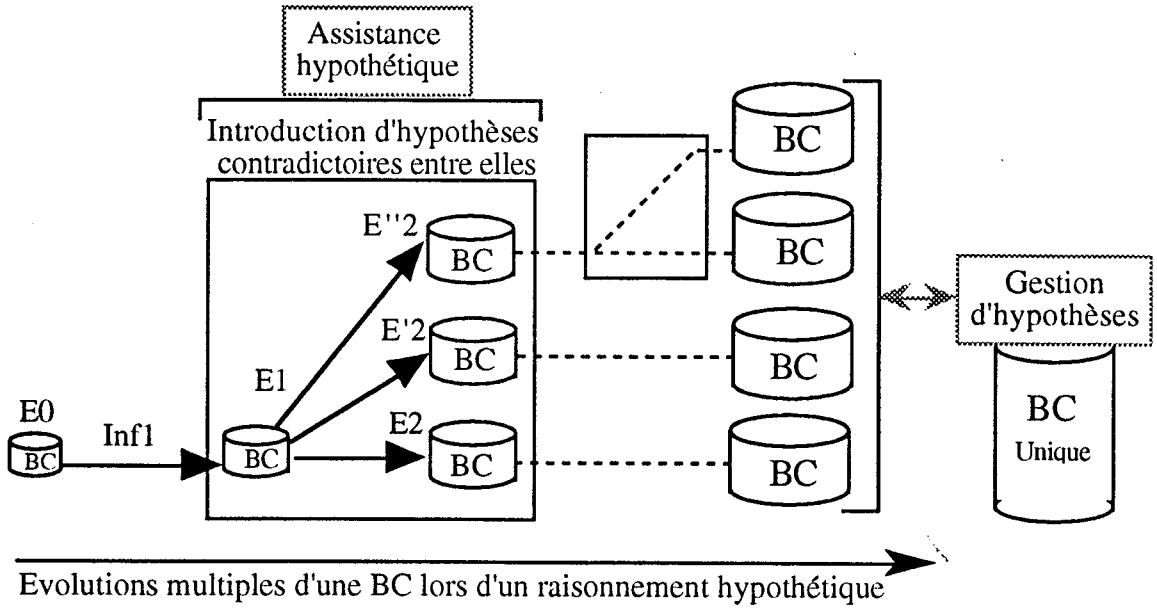

Figure VIII.5 : L'assistance hypothétique fait connaître des évolutions différentes à la base de connaissances. La gestion des hypothèses d'une base de connaissances doit permettre à un raisonnement d'explorer ces différents développements possibles de façon cohérente.

Les hypothèses produites par une phase d'assistance étant des réponses différentes à une même question, elles sont contradictoires, et ne doivent pas être considérées simultanément par le raisonnement déductif de TROPES ; ceci afin que la cohérence reste garantie.

2.1. Représentation des hypothèses dans TROPES

Cette présentation générale suggère plusieurs caractéristiques importantes à prendre en compte pour l'intégration de la composante hypothétique :

- Les différents développements menés par un raisonnement sont indépendants et donnent leur propre interprétation de l'évolution de la base par les connaissances qu'ils produisent. Les connaissances manipulées par un développement forment son **contexte**.
- Une hypothèse est une connaissance contextuelle ; c'est-à-dire qu'elle n'est visible que dans le contexte des développements dans lesquels elle a été introduite. Lorsque le raisonnement évolue dans le contexte d'un développement, il considère les hypothèses du contexte comme des connaissances à part entière. Toute connaissance produite à partir d'une hypothèse possède le même comportement hypothétique que l'hypothèse.
- Chaque développement du raisonnement est caractérisé par l'ensemble des hypothèses qui lui ont été fournies. Cet ensemble d'hypothèses est appelé l'**environnement hypothétique** du contexte développé.
- Les hypothèses sont introduites dans la base de connaissances à chaque fois que le raisonnement fait face à une situation de connaissances incomplètes. Le fonctionnement du système peut être vu comme une alternance de phases de raisonnement déductif et de phases d'assistance au raisonnement par production d'hypothèses.

Le principe général que suggèrent ces caractéristiques consiste à étendre le système TROPES vers un fonctionnement multi-contextes tel que le permet un ATMS (cf. §III.3.2). C'est-à-dire qu'il doit être capable de représenter plusieurs contextes, de créer des nouveaux contextes à chaque fois que des nouveaux environnements hypothétiques sont introduits et d'explorer l'espace formé par ces contextes pour les développer. Le développement d'un contexte, son environnement hypothétique étant fixé, correspond au fonctionnement normal de TROPES. La constitution de nouveaux environnements hypothétiques intervient lors de la phase d'assistance hypothétique, c'est-à-dire lorsqu'une situation de connaissances incomplètes peut-être résolue par l'introduction d'hypothèses.

Cette proposition de fonctionnement se concrétise dans la description de l'architecture générale fournie dans la partie suivante.

2.2. Architecture générale

L'intégration de capacités hypothétiques au système TROPES nécessite des aménagements particuliers dans son organisation générale. Le schéma proposé (cf. Figure VIII.6) montre comment est étendue l'ancienne architecture de TROPES.

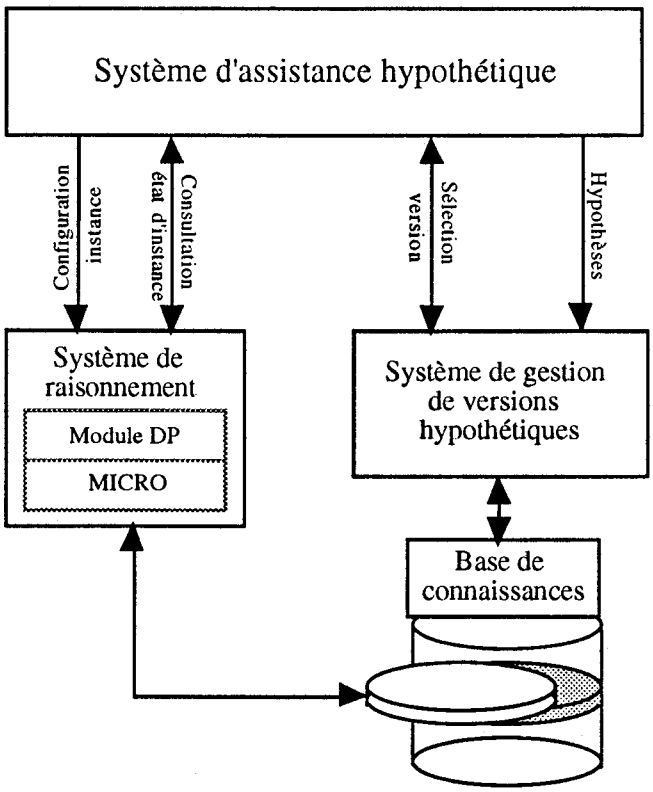

Figure VIII.6 : Architecture générale du système proposée pour la gestion d'un raisonnement hypothétique. Le système d'assistance hypothétique fournit les mécanismes de production d'hypothèses et de contrôle d'exploration des hypothèses. Le système de gestion de versions hypothétiques permet de gérer et de structurer la base de connaissances en versions cohérentes induites par les hypothèses produites. Suivant les instructions du système d'assistance, le système de gestion de versions hypothétiques fournit au système de raisonnement la version de la base de connaissances à partir de laquelle le raisonnement doit se développer.

Un raisonnement hypothétique (développement de plusieurs contextes) mené par le système d'assistance hypothétique se déroule sur deux niveaux :

- Le niveau du raisonnement normal de TROPES : système de raisonnement et une **version** de la base de connaissances. Une **version** correspond à une association environnement-contexte, c'est-à-dire les connaissances de la base de connaissances auxquelles on peut accéder sous certaines hypothèses. Le système de raisonnement a pour charge la mise en œuvre du raisonnement déductif à partir de la version de la base de connaissances dont il dispose.
- Le niveau d'assistance hypothétique du raisonnement : le système d'assistance hypothétique, le système de gestion de versions hypothétiques et la base de connaissances globale.

Le premier niveau correspondant au fonctionnement normal de TROPES (sans la notion d'hypothèse), seul le niveau d'assistance hypothétique est abordé par la suite. Le système d'assistance hypothétique est présenté à travers la mise en place du mécanisme de résolution de problèmes d'identification/construction, c'est-à-dire la *configuration d'instance par assistance hypothétique* (cf. §VIII.3). Le système de gestion de versions hypothétiques repose sur le principe de gestion de versions hypothétiques d'instances inspiré des techniques de gestion d'hypothèses de l'ATMS et présenté dans le chapitre III. Son interface avec le système d'assistance est présentée par la suite (cf. §VIII.3.1.2).

3. Configuration par assistance hypothétique

Cette partie se propose de décrire la mise en place d'un mécanisme de résolution de problème d'identification/construction d'instance à partir du système d'assistance hypothétique. Après une présentation générale de l'algorithme (cf. §VIII.3.1), les étapes principales de ce mécanisme sont détaillées (cf. §VIII.3.2 et §VIII.3.3).

3.1. Mise en place du mécanisme de résolution

La résolution d'un problème d'identification/construction d'instance est prise en charge par le système d'assistance hypothétique grâce à un mécanisme de configuration hypothétique d'instance. Ce mécanisme permet à un utilisateur de formuler son problème en désignant l'instance concernée et la contrainte de productivité imposée (cf. §VIII.2.2.1). L'instance désignée appartient à la base de connaissances et est déjà partiellement configurée.

Avant de proposer l'algorithme de résolution, les primitives de base des différents systèmes composant l'architecture étendue de TROPES sont préalablement spécifiées (cf. §VIII.3.1.1 à §VIII.3.1.3). Tirant parti de ces primitives, le schéma d'algorithme, présenté ensuite (cf. §VIII.3.1.4), montre comment les différentes ressources de TROPES sont exploitées pour mettre en place un mécanisme de configuration d'instance par assistance hypothétique.

Afin d'éviter toute confusion entre les primitives offertes par chaque système, une convention de nommage est adoptée. Cette convention consiste simplement à préfixer le nom d'une primitive par l'acronyme du système dont elle provient. Les préfixes sont les suivants : SR pour système de raisonnement, SGVH pour système de gestion de versions hypothétiques et SAH pour système d'assistance hypothétique.

3.1.1. Primitives du système de raisonnement

Une primitive que le système de raisonnement doit nécessairement offrir à tout utilisateur externe, et en particulier au système d'assistance hypothétique, est celle permettant de configurer une instance. Cette primitive, appelée ***SR_Config***, est définie ainsi :

- ***SR_Config*** :

  $Instance \times \mathcal{L}_{Config} \rightarrow Instance \cup \{\perp_{Inst}\}$

où :

- *Instance* est l'ensemble des instances de la base de connaissances que peut accéder le système de raisonnement.
- $\mathcal{L}_{Config}$ est le langage utilisé par un utilisateur pour exprimer un apport de connaissances sur une instance. Une expression de ce langage est équivalente à la donnée d'un ensemble de connaissances élémentaires pouvant être :
  - $(I \in C)$ ou $(I \notin C)$:
    La déclaration d'appartenance ou de non-appartenance de l'instance I à une classe C ;
  - $Cstr<A_1..., A_i>$:
    La déclaration d'une contrainte inter-attributs ;
  - $A \in \mathcal{A}(I)$:
    L'introduction de A dans l'ensemble $\mathcal{A}(I)$ des attributs de l'instance I ;
  - $A \leftarrow d$:
    Le choix d'une connaissance de production, dénotée par son déclencheur d, pour un attribut A. Pour que ce choix soit possible, l'attribut A doit être dans l'état **indécis** (cf. §VII.4.2.2). Dans ce cas, le déclencheur d appartient aux déclencheurs d'inférence applicables App(A, I).

- $\perp_{Inst}$ représente une instance incohérente.

La primitive ***SR_Config*** a pour fonction d'interpréter et de valider un nouvel apport de connaissances sur l'instance désignée en entrée. En retour, elle rend soit une instance dans lesquelles prennent place toutes les informations que le système de raisonnement a pu déduire à partir des connaissances apportées, soit la valeur $\perp_{Inst}$ si une incohérence a été détectée.

Le système de raisonnement, en tant que système de représentation, offre aussi une palette de primitives permettant de consulter les informations contenues par une instance : ensemble des attributs de l'instance, valeur d'un attribut particulier, l'état du contrôle d'évaluation d'un attribut, les classes d'appartenance sûres, impossibles ou possibles de l'instance, etc.

3.1.2. Primitives du système de gestion de versions hypothétiques

L'unité de base manipulée au niveau de l'assistance hypothétique est la version hypothétique d'instance. Le système de gestion de versions hypothétiques associe à chaque instance I qui fait l'objet d'un développement hypothétique l'ensemble $\mathcal{V}(I)$ de ses versions hypothétiques, son *bassin hypothétique* (cf. §III.5).

Une version hypothétique d'instance comporte l'ensemble des informations nécessaires au développement du raisonnement hypothétique mené par le système d'assistance hypothétique. Une version peut être représentée comme un triplet (I, H, CP) où :

- I est l'état de l'instance correspondant à la version, c'est-à-dire le *contexte* de la version,
- H un ensemble d'hypothèses, appelé *environnement hypothétique* de la version,
- CP la contrainte de productivité que le système d'assistance cherche à satisfaire au niveau de cette version. Une contrainte de productivité est représentée par un sous-ensemble d'attributs désignant les buts de l'utilisateur restant à atteindre.

La notation pointée suivante est utilisée dans la suite pour exprimer les différents accès que le système d'assistance peut entreprendre sur une version V=(I, H, CP) :

- *V.Inst* représente l'accès à l'état de instance I correspondant à la version V ;
- *V.Env* représente l'accès à l'ensemble d'hypothèses H de la version V ;
- *V.Cont* représente l'accès à la contrainte de productivité CP de la version V.

Parmi les primitives fournies par le système de gestion de versions hypothétiques, deux sont essentielles pour la mise en place du mécanisme de configuration par assistance hypothétique :

- ***SGVH_charger_version_dans_BC*** :

  $Version\_instance \rightarrow Instance$

  permettant de remplacer l'état d'une instance de l'ensemble *Instance*, géré par le système de raisonnement, par l'état associé à l'une de ses versions hypothétiques qui appartient à l'ensemble des versions d'instances *Version_instance*.

- ***SGVH_Construire_version*** :

  $Version\_instance \times Hypothèse^* \rightarrow Version\_instance^*$

  permettant de construire de nouvelles versions d'une instance à partir d'une version d'instance précise et d'un ensemble d'hypothèses. Une hypothèse est une connaissance de configuration élémentaire (cf. §VIII.3.1.1), limitée aux cas suivants : une déclaration d'appartenance ou de non-appartenance à une classe, ou le choix d'une connaissances de production pour un attribut. Les nouvelles versions sont obtenues en construisant toutes les combinaisons d'hypothèses ne comportant pas d'hypothèses contradictoires.

Le système de gestion de versions hypothétiques offre la possibilité au système d'assistance de partitionner l'ensemble $\nu(I)$ en plusieurs sous-ensembles dont deux sont nécessaires lors de la configuration d'une instance par assistance hypothétique :

- $\nu_{Sol}(I)$, l'ensemble des versions de l'instance I constituant une solution pour le système d'assistance, et
- $\nu_{Nouv}(I)$, l'ensemble des versions de l'instance I qui sont déjà créées mais restent encore à explorer par le système d'assistance.

3.1.3. Primitives du système d'assistance hypothétique

En dehors du mécanisme de configuration d'instance par assistance hypothétique, le système d'assistance hypothétique possède deux fonctions de base : la vérification pour une instance d'une contrainte de productivité et la production d'hypothèses à partir d'une instance.

- ***SAH_Vérification_Contrainte*** :

  $Instance \times Contrainte\_productivité \rightarrow Contrainte\_productivité \cup \{\perp_{Cont}\}$

  Cette primitive confronte une contrainte de productivité à une instance ; c'est-à-dire qu'elle vérifie que les attributs désignés comme buts sont encore évaluables. En sortie, cette primitive rend soit la valeur $\perp_{Cont}$ si la contrainte s'avère insatisfaisable, soit une nouvelle contrainte de productivité permettant de définir à partir de l'état d'évaluation de l'instance les buts restant à atteindre. Dans ce dernier cas, deux situations sont possibles :

  - **la nouvelle contrainte de productivité n'est pas vide**, la contrainte initiale reste alors satisfaisable. La nouvelle contrainte de productivité est obtenue en éliminant les buts déjà atteints.
  - **la nouvelle contrainte de productivité est vide**, tous les buts ont été atteints et la contrainte de productivité initiale est donc satisfaite.

- ***SAH_Produire_hypothèses*** :

  $Instance \rightarrow Hypothèse^*$

  Cette primitive permet de produire un ensemble d'hypothèses à partir d'une instance. Suivant l'état de l'instance, les hypothèses produites concerneront soit l'évaluation des attributs, soit le raffinement de l'instance.

Ces deux primitives constituent les deux étapes principales du mécanisme de configuration par assistance hypothétique. Leur fonctionnement est décrit dans la suite de ce chapitre (cf. §VIII.3.2 et §VIII.3.3).

3.1.4. Algorithme de configuration d'instance par assistance hypothétique

L'algorithme suivant (cf. Figure VIII.7) permet de construire à partir de la description d'un concept toutes les versions hypothétiques d'une instance qui satisfont la contrainte de productivité fixée en paramètre. Une étape de cet algorithme consiste en la sélection d'une version hypothétique de l'instance, la validation des hypothèses associées à cette version, la vérification de la contrainte de productivité, puis la production de nouvelles hypothèses pour former de nouvelles versions à explorer si nécessaire.

Les nouvelles versions ainsi construites partagent l'état courant de l'instance et la nouvelle contrainte de productivité à satisfaire. Elles se distinguent les unes des autres par la combinaison de nouvelles hypothèses qui leur est associée et qui devra être validée à une prochaine étape.

Le processus se termine lorsque le système d'assistance ne peut plus produire aucune hypothèse à partir des versions hypothétiques déjà créées ; c'est-à-dire qu'il ne peut plus créer de nouvelles versions hypothétiques à valider. L'ensemble des versions solutions est construit au fur et à mesure que les versions créées sont validées, celles qui satisfont la contrainte de productivité sont ajoutées à cet ensemble de solutions.

SAH_Config_Hypothétique (I : une instance;
CP : une contrainte de productivité)

Début

; INITIALISATION
; Ces deux ensembles sont fournis au système d'assistance
; par le système de gestion de versions hypothétiques.

$\mathcal{V}_{Nouv}$ = {(I, Ø, CP)} ; Ensemble initial des versions à traiter
$\mathcal{V}_{Sol}$ = Ø ; Ensemble initial des versions solutions

; TRAITEMENT GLOBAL

Pour tout ($v_{cour} \in \mathcal{V}_{Nouv}$) **faire**

; L'instance v_{cour}.Inst est chargé dans la version de la BC accessible au SR

SGVH_charger_version_dans_BC (v_{cour}) ;
$\mathcal{V}_{Nouv} = \mathcal{V}_{Nouv} \setminus \{v_{cour}\}$;

; les hypothèses v_{cour}.Env sont appliquées à l'instance v_{cour}.Inst

v_{cour}.Inst = ***SR_Config*** (v_{cour}.Inst, v_{cour}.Env) ;

Si (v_{cour}.Inst ≠ $\perp_{Inst}$) **alors**

; les hypothèses ne mènent pas à une incohérence
; alors la contrainte de productivité est vérifiée

v_{cour}.Cont = ***SAH_Vérification_Contrainte*** (v_{cour}.Inst, v_{cour}.Cont) ;

Choix

- (v_{cour}.Cont = Ø) :

  ; la contrainte est satisfaite, la version v_{cour} constitue une solution
  $\mathcal{V}_{Sol} = \mathcal{V}_{Sol} \cup \{v_{cour}\}$;

- (v_{cour}.Cont ≠ $\perp_{Cont}$) **et** (v_{cour}.Cont ≠ Ø) :

  ; La contrainte est non satisfaite. L'état de l'instance permet
  ; de construire l'ensemble $\mathcal{H}_{cour}$ des nouvelles hypothèses à explorer.
  ; Enfin, de nouvelles versions sont construites à partir de ces hypothèses.

  $\mathcal{H}_{cour}$ = ***SAH_Produire_hypothèses*** (v_{cour}.Inst) ;
  $\mathcal{V}_{Nouv}$ = $\mathcal{V}_{Nouv} \cup$ ***SGVH_Construire_versions*** (v_{cour}, $\mathcal{H}_{cour}$) ;

Fchoix

Finsi

Finpour

Retourner ($\mathcal{V}_{Sol}$) ; l'ensemble des versions solutions est rendu en sortie

Fin

Figure VIII.7 : Algorithme général de configuration par assistance hypothétique

3.2. Vérification et maintenance de la contrainte de productivité

La notion de contrainte de productivité permet au système d'assistance de contrôler l'activité de configuration d'instance. Le processus de résolution n'a d'autre but que celui de satisfaire cette contrainte.

Pour exprimer sur une instance I une contrainte de productivité CP(I), l'utilisateur désigne un sous-ensemble de l'ensemble $\mathcal{A}(I)$ des attributs d'une instance I :

- $CP(I) \subseteq \mathcal{A}(I)$

Du point de vue du contrôle d'évaluation des attributs, l'ensemble $\mathcal{A}(I)$ peut être décomposé en la partition suivante :

- $\mathcal{A}_E(I)$: sous-ensemble des attributs évalués de l'instance I, c'est-à-dire qui sont dans l'état d'évaluation **Evalué**.
- $\mathcal{A}_I(I)$: sous-ensemble des attributs inconnus de l'instance I, c'est-à-dire qui sont dans l'état d'évaluation **Inconnu**. On rappelle que dans ce cas le contrôle d'évaluation a épuisé tous les moyens d'évaluation connus.
- $\mathcal{A}_A(I)$: sous-ensemble des attributs en attente d'évaluation ; c'est-à-dire qui sont dans l'un des états d'évaluation suivants : **Demandeur** (une inférence en cours d'exécution est en attente d'informations), **Bloqué** (attente d'un accès à un moyen d'évaluation prioritaire), **Indécis** (attente d'une décision concernant un choix exhaustif de moyens d'évaluation applicables).

Tirant parti de ces informations, le système d'assistance peut donc établir l'état de productivité de l'instance I de la façon suivante :

- $[CP(I) \subseteq \mathcal{A}_E(I)] \Leftrightarrow$ CP(I) est **satisfaite** par l'instance I ;
- $[CP(I) \cap \mathcal{A}_I(I) \neq \emptyset] \Leftrightarrow$ CP(I) est **insatisfaisable** pour l'instance I ;
- sinon, c'est-à-dire $[CP(I) \subseteq \mathcal{A}_E(I) \cup \mathcal{A}_A(I)]$, CP(I) reste à vérifier par l'instance I

La vérification et la maintenance d'une contrainte de productivité sont donc assurées par l'application de la fonction ***SAH_Vérification_Contrainte*** à chaque version qui vient d'être traitée par le système d'assistance.

Le corps de cette fonction se résume aux lignes suivantes :

SAH_Vérification_Contrainte (I : une instance;
CP: une contrainte de productivité)

Début

Si $[CP(I) \cap \mathcal{A}_I(I) \neq \emptyset]$ **alors retourner** ($\perp_{Cont}$)

sinon retourner ($CP(I) \setminus \mathcal{A}_E(I)$)

Fin

Si la contrainte n'est pas insatisfaisable, cette fonction fournit la nouvelle contrainte de productivité que le système d'assistance se charge d'imposer aux versions qui succéderont à celle qui vient d'être traitée. Sinon, elle rend la contrainte insatisfaisable $\perp_{Cont}$.

3.3. Gestion de la production d'hypothèses

La phase de production d'hypothèses est mise en place par la fonction du système d'assistance appelée ***SAH_Produire_hypothèses***. Pour les besoins de la résolution, cette fonction se spécialise en fonction de l'état de l'instance en un mécanismes de production soit d'hypothèses d'évaluation des attributs, soit d'hypothèses de raffinement de l'instance (cf. §VIII.1.2.2.2).

La stratégie de production adoptée rend prioritaire la production d'hypothèses sur l'évaluation des attributs. La première étape de cette fonction consiste donc à déterminer si l'instance peut faire l'objet de ce genre d'hypothèses. Ce sera le cas si des attributs de l'instance sont dans un état **Indécis**.

Soient I une instance et $\mathcal{A}_{Ind}(I)$, tel que $\mathcal{A}_{Ind}(I) \subseteq \mathcal{A}_A(I)$, l'ensemble des attributs de l'instance I dont l'état a été déclaré **Indécis** par le contrôle d'évaluation. Selon l'état de

$\mathcal{A}_{Ind}(I)$, la fonction ***SAH_Produire_hypothèses*** détermine le type de production d'hypothèses à entreprendre :

SAH_Produire_hypothèses (I : une instance)

Début

Si [$\mathcal{A}_{Ind}(I) \neq \emptyset$] **alors** ***SAH_Prod_hyp_attribut*** (I)

sinon ***SAH_Prod_hyp_raffinement*** (I);

Fin

Les principes de production propres à chaque type d'hypothèses, mis en place par les fonctions ***SAH_Prod_hyp_attribut*** et ***SAH_Prod_hyp_raffinemen***, sont présentés dans les deux parties suivantes.

3.3.1. Hypothèses liées à l'évaluation d'un attribut

Si une instance fait l'objet d'une phase de production d'hypothèses sur l'évaluation des attributs, il existe au moins un de ses attributs dans l'état d'évaluation **Indécis**. Dans ce cas, la production d'hypothèses à pour objectif l'exploration de tous les moyens d'évaluation applicables pour chaque attribut indécis.

3.3.1.1. Principe général

Soit A un attribut de l'instance I, tel que $A \in \mathcal{A}_{Ind}(I)$. Du point de vue du contrôle, l'attribut A se caractérise par un ensemble App(A, I) de déclencheurs applicables dont la cardinalité est strictement supérieure à 1 :

- $App(A, I) = \{d_1 \ldots, d_i\}$ avec $i \geq 2$.

Cet ensemble décrit un choix exhaustif de moyens d'évaluation qui sont applicables pour A dans l'état courant de l'instance I. Ce choix, normalement résolu par l'utilisateur, se traduit dans le contexte du système d'assistance par l'émission d'hypothèses sur l'attribut A. L'ensemble de ces hypothèses, noté $\mathcal{H}(A, I)$, est donc construit de la façon suivante :

- $\mathcal{H}(A, I) = \{(A \leftarrow d), d \in App(A, I)\}$

Lors de son application dans une version de l'instance I, une hypothèse de configuration $(A \leftarrow d)$ a pour effet de réduire l'ensemble App(A, I) au singleton {d} : App(A, I)={d}. Dans cette version, l'attribut A fera donc l'objet de l'exécution du moyen d'évaluation correspondant au déclencheur d. Les hypothèses de $\mathcal{H}(A, I)$ sont contradictoires entre elles et ne peuvent prendre place dans une même version.

La phase de production d'hypothèses consiste donc à traiter ainsi chaque attribut de l'instance dont l'état d'évaluation est **Indécis**. Le résultat de cette phase sur l'instance I est donc l'ensemble $\mathcal{H}(I)$ des couples $(A, \mathcal{H}(A, I))$ associant à chaque attribut A indécis son ensemble d'hypothèses :

- $\mathcal{H}(I) = \{(A, \mathcal{H}(A, I)), A \in \mathcal{A}_{Ind}(I)\}$

La donnée de $\mathcal{H}(I)$ au système de gestion de versions hypothétiques provoque la création d'autant de versions de l'instance que de combinaisons pouvant être construites en choisissant une hypothèse parmi celles de chaque attribut indécis.

3.3.1.2. Proposition d'extension des situations d'indécision

En théorie, la production de ce genre d'hypothèses est possible. Toutefois, en pratique, une situation de choix entre différents types de mécanismes d'évaluation pour un attribut correspond à une stratégie d'évaluation incomplètement définie. Il semble peu vraisemblable qu'un concepteur cherche à exprimer pour certains attributs le choix exhaustif de valeurs à travers le choix de divers types d'inférence.

En revanche, le concepteur peut vouloir exprimer ce choix à partir d'un seul mécanisme d'évaluation. Dans ce cas, l'exécution de ce mécanisme peut rendre plusieurs résultats

possibles. L'indécision du contrôle ne porte alors plus sur le choix du type de mécanisme mais sur le choix du résultat.

Dans l'exemple des voitures, le détachement procédural associé à l'attribut *PrixTTC* d'une voiture pourrait proposer différents calculs représentant aussi bien le prix au comptant que les prix lorsque des formules de crédit sont appliquées. Un client peut vouloir étudier les différentes solutions de paiement qui s'offre à lui. Dans ce cas, lors de la configuration de la voiture, chaque calcul de prix devra être considéré comme une hypothèse d'évaluation de l'attribut *PrixTTC*.

Le contrôle d'évaluation des attributs, tel qu'il a été présenté, ne prend pas en compte l'indécision au niveau du résultat que peut délivrer un mécanisme d'évaluation d'attribut. Cependant, en termes de contrôle, cette indécision peut être traitée en considérant qu'un déclencheur dont l'application rend plusieurs résultats correspond à un déclencheur qui connaît un raffinement en autant de déclencheurs que de résultats (cf. Figure VIII.8). L'extension du contrôle consiste simplement à permettre pour un attribut la transition de l'état **Demandeur** (exécution d'une inférence) à l'état **Indécis** (plusieurs déclencheurs applicables).

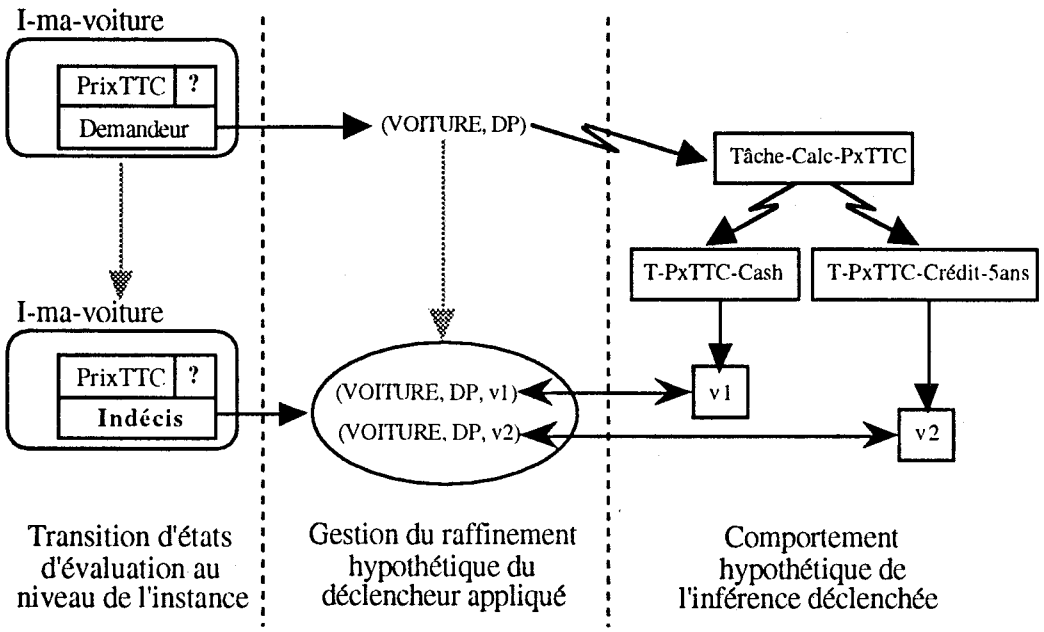

Figure VIII.8 : Proposition d'extension du contrôle pour prendre en compte le comportement hypothétique d'une inférence. Ici, l'attribut PrixTTC fait appel à son détachement procédural. Ce dernier propose deux calculs possibles : calcul du prix au comptant et calcul du prix à crédit (sur cinq ans). Au niveau de la gestion de la stratégie, le comportement hypothétique du DP se traduit par la création dynamique de deux déclencheurs chargée d'exprimer le choix entre les deux résultats. Au niveau de l'instance, l'attribut PrixTTC passe alors de l'état demandeur à l'état indécis.

Pour l'utilisateur, la résolution de cette indécision revient à choisir un résultat parmi ceux rendus par le déclencheur appliqué. Dans un mode hypothétique, ce choix se traduit par autant d'hypothèses d'évaluation de l'attribut que de résultats trouvés.

3.3.2. Hypothèses liées au raffinement de l'instance

Il s'agit dans ce cas d'établir en fonction de l'état de raffinement d'une instance l'ensemble des états de raffinement qui peuvent (directement) lui succéder. Chacun de ces états correspond à une hypothèse de raffinement possible que le système d'assistance va explorer.

L'état de raffinement d'une instance I se caractérise par sa classe de rattachement courant, la classe sûre la plus spécialisée dénotée par le lien **est-un** de I, et l'ensemble de ses sous-classes directes possibles, noté $\mathbf{SC}_P(I)$; les sous-classes restantes étant alors nécessairement

impossibles. L'ensemble des états de raffinement pouvant dériver de l'état courant correspondent à l'ensemble des combinaisons cohérentes de marquage dans lesquelles chaque sous-classe possible obtient le statut sûr ou impossible.

Une hypothèse H de raffinement se présente donc comme l'ensemble d'assertions suivant :

- $H=\{h_i / \forall i, C_i \in \mathcal{SC}_P(I), h_i=(I \in C_i) \text{ ou } h_i=(I \notin C_i)\}$

  C'est-à-dire que pour chaque sous-classe C_i possible, il y a production soit de l'hypothèse ($I \in C_i$), soit de l'hypothèse ($I \notin C_i$).

Théoriquement, il y a $2^{|\mathcal{SC}_P(I)|}$ combinaisons de marquages possibles à partir de l'état courant de l'instance I. Toutefois, toutes ces combinaisons ne sont pas nécessairement cohérentes au vu des propriétés que la classe de rattachement courant peut posséder. Dans le modèle TROPES, l'exclusivité est une propriété implicite de toute décomposition de classe en sous-classes. Dans ce cas, toute combinaison de marquage proposant au moins deux sous-classes sûres devient donc incohérente.

Les propriétés considérées par la suite décrivent les différentes nuances qui peuvent être apportées à une décomposition exclusive de classe en sous-classes. Les règles de production d'hypothèses sont établies en fonction de chacune d'elles.

3.3.2.1. Hypothèses de raffinement à partir d'une classe exhaustive

Si la classe C de rattachement courant de l'instance I possède la propriété d'exhaustivité, l'instance I appartient alors nécessairement à l'une des sous-classes possibles de C. Puisque les règles de propagation de marques de la classification n'ont pas permis de déterminer quelle sous-classe de C est sûre, l'ensemble $\mathcal{SC}_P(I)$ des sous-classes de C qui sont possibles pour I est donc de cardinalité strictement supérieure à 1.

Tout état de raffinement pouvant succéder à cette situation peut être représenté comme le marquage sûr de l'une des sous-classes possibles ; la propagation de marques assurant que les autres sous-classes seront impossibles. Une hypothèse de raffinement dans ce cas se traduit par :

- $I \in C'$, avec $C' \in \mathcal{SC}_P(I)$

De ce fait, l'ensemble $\mathcal{H}(I)$ des hypothèses de raffinement issu de la phase de production est l'ensemble de singletons suivant :

- $\mathcal{H}(I)= \{\{I \in C'\}, \forall C' \in \mathcal{SC}_P(I)\}$

Chaque singleton représente l'information nécessaire et suffisante pour caractériser un état de raffinement possible. Les hypothèses de raffinement ainsi représentées sont deux à deux contradictoires. Elles donneront donc lieu à la création d'autant de versions hypothétiques de l'instance I.

3.3.2.2. Hypothèses de raffinement à partir d'une classe ouverte ou fermée

La propriété d'exhaustivité constitue un cas particulier de la propriété d'exclusivité. Elle permet d'éliminer des raffinements possibles le cas où toutes les sous-classes sont déclarées impossibles. Ce cas est effectivement incohérent lorsque la propriété d'exhaustivité est soutenue.

Sans la propriété d'exhaustivité, il devient donc possible de considérer le cas où les sous-classes de la classe de rattachement courant sont toutes impossibles. Toutefois, cette hypothèse n'est pas toujours souhaitable ; elle l'est d'ailleurs rarement.

Dans l'exemple de la hiérarchie de voitures (cf. Figure VIII.9), si aucune propriété d'exhaustivité n'est imposée, deux interprétations possibles peuvent être données à la décomposition exclusive d'une classe représentant un modèle de voitures (la classe *Modèle-M*)

en plusieurs sous-classes représentant des séries de ce modèle (les classes *Série-Tourisme...*, *Série-Spéciale-Jeune*).

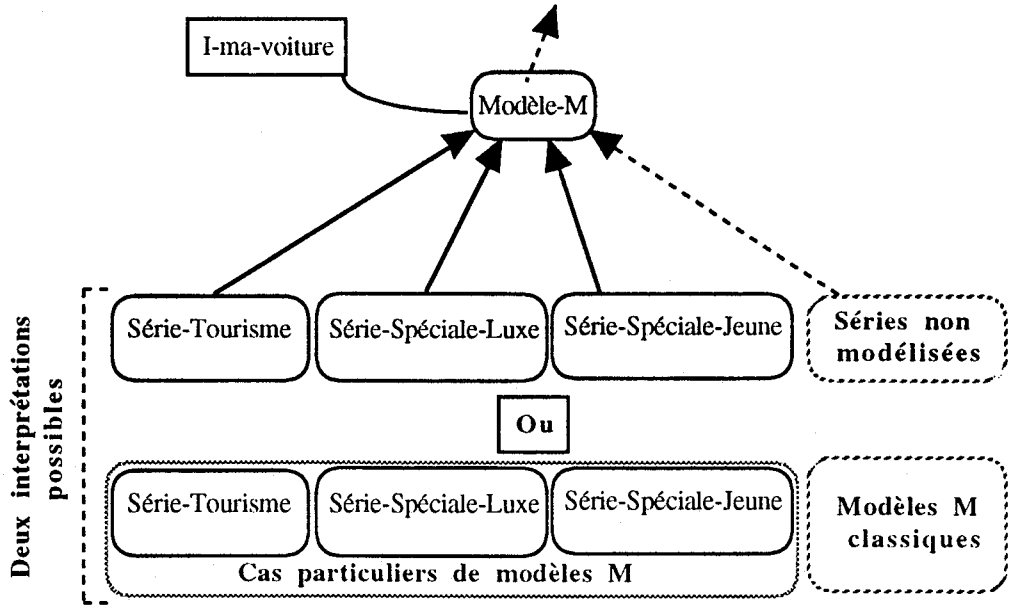

Figure VIII.9 : Deux interprétations possibles de la décomposition de la classe *Modèle-M* en ses sous-classes. Celles-ci représentent-elles toutes les séries de voiture de modèle M connues ou des cas particuliers qui se distinguent du cas classique de modèle M ?

Lorsqu'un concepteur décrit la spécialisation d'une classe en sous-classes exclusives, deux types de modélisation sont envisageables :

- la classe représente une décomposition **ouverte** (cette classe est alors dite ouverte) : l'objectif du concepteur est de décrire une décomposition exhaustive d'un ensemble d'instances à partir des sous-classes. Toutefois, cette décomposition est incomplète ; certaines sous-classes sont omises car elles ne sont pas encore connues (cas possible d'une hiérarchie de maladies) ou pas considérées pertinentes (le vendeur de voitures ne décrit que les séries d'un modèle de voiture qu'il vend).
- la classe représente une décomposition **fermée** (cette classe est alors dite fermée) : l'objectif du concepteur est de décrire une décomposition exhaustive sous la forme d'un cas général, représenté par la classe se spécialisant, et plusieurs cas spécifiques, représentés par les sous-classes. La spécialisation sert dans ce cas à modéliser des exceptions. Par exemple, les sous-classes *Pingouin* et *Autruche* de la classe *Oiseau* constituent ses exceptions.

Le cas d'une classe fermée peut être réécrit en un cas de décomposition exhaustive de classe en sous-classes. Par exemple, la classe *Modèle-M* pourrait se décomposer exhaustivement en une sous-classe de *Modèle-M-Classique* et des sous-classes de séries spéciales.

Etant donné le caractère particulier de la notion de classe fermée, une classe représentant une décomposition **fermée** doit être explicitement déclarée comme telle. En revanche, une classe est considérée **ouverte** par défaut.

Lors de la production d'hypothèses ces deux types de classes se distinguent par la prise en compte ou non d'une hypothèse de raffinement : celle représentant le cas où toutes les sous-classes sont impossibles.

Si la classe de rattachement courant de l'instance I est **ouverte**, l'ensemble $\mathfrak{H}(I)$ d'hypothèses de raffinement produit est similaire à celui d'un cas de classe possédant la propriété d'exhaustivité :

- $\mathcal{H}(I) = \{\{I \in C'\}, \forall C' \in \mathcal{SC}_P(I)\}$

Si la classe de rattachement courant de l'instance I est **fermée**, l'ensemble $\mathcal{H}(I)$ intègre, de plus, le cas où toutes les sous-classes sont déclarées impossibles :

- $\mathcal{H}(I) = \{\{I \in C'\}, \forall C' \in \mathcal{SC}_P(I)\} \cup \{H\}$

avec $H = \{h_i / \forall i, C_i \in \mathcal{SC}_P(I), h_i = (I \notin C_i)\}$

Par exemple, si on suppose que l'instance *I-ma-voiture* est dans l'état de raffinement suivant :

- **est-un**= *Modèle-M*
- $\mathcal{SC}_P$(*I-ma-voiture*)={*Série-Tourisme* ; *Série-Spéciale-Jeune*}

Suivant le type de la classe *Modèle-M*, l'ensemble $\mathcal{H}$(*I-ma-voiture*) obtenu est différent :

- *Modèle-M* est **ouverte** :

  $\mathcal{H}$(*I-ma-voiture*)={ {*I-ma-voiture* ∈ *Série-Tourisme*} ;
  {*I-ma-voiture* ∈ *Série-Spéciale-Jeune*}}

- *Modèle-M* est **fermée** :

  $\mathcal{H}$(*I-ma-voiture*)={
  {*I-ma-voiture* ∈ *Série-Tourisme*} ; {*I-ma-voiture* ∈ *Série-Spéciale-Jeune*} ;
  {*I-ma-voiture* ∉ *Série-Tourisme* ; *I-ma-voiture* ∉ *Série-Spéciale-Jeune*} }

Il faut noter une différence essentielle entre le traitement d'une classe ouverte et celui d'une classe exhaustive : il est possible que la production d'hypothèses se réduise à la donnée d'une seule hypothèse de raffinement dans le cas d'une classe ouverte alors que c'est impossible dans le cas d'une classe exhaustive. En effet, lors du traitement d'une classe exhaustive, il y au moins deux sous-classes possibles.

4. Identification/construction d'objet composite

Lorsque la configuration par assistance hypothétique est entreprise sur un objet composite, certains de ses composants peuvent subir aussi des productions d'hypothèses. Il y a dans ce cas construction de versions hypothétiques pour ces composants.

La construction de versions hypothétiques pour un composant peut provenir :

- de la répercussion des hypothèses émises sur le composite. Si les informations apportées par le composite sur le composant proviennent d'hypothèses, ces informations sont considérées au niveau du composant comme des hypothèses émisent par le composite. Elles prennent donc place dans une version du composant.
- de la nécessité pour le composite d'explorer différentes configurations de composants afin de satisfaire la contrainte de productivité.

Le premier cas est à la charge du système de gestion de versions hypothétiques. Il s'agit dans ce cas de garantir que les connaissances sont dans le contexte correspondant à leur environnement hypothétique ; c'est-à-dire l'ensemble des hypothèses dont elles dépendent.

En revanche, le second cas constitue une transposition au niveau du composant du problème posé au niveau du composite. Il s'agit de permettre l'exploration hypothétique des configurations de certains composants afin de satisfaire la contrainte de productivité du composite. Les versions hypothétiques d'un composant, pouvant ainsi être construites par ce processus, forment de nouvelles hypothèses que le composite peut explorer.

Pour mettre en place ce principe, il faut toutefois définir :

- les conditions de déclenchements de la production d'hypothèses au niveau des composants ;

- le choix des composants susceptibles d'apporter des informations nécessaires à la satisfaction de la contrainte de productivité de l'objet composite.
- les conditions permettant de considérer qu'une version d'un composant constitue une hypothèse pertinente pour l'objet composite.

La solution proposée pose des conditions qui ont pour but d'éviter une explosion combinatoire dans la recherche des configurations de l'objet composite. En contre-partie, ses conditions sont parfois restrictives et peuvent occasionnellement occulter des voies d'exploration qui pourraient être fructueuses. Par la suite ses restrictions sont présentées dans le contexte dans lequel elles apparaissent.

4.1. Problèmes de la construction hypothétique de composants

Il s'agit dans cette partie de définir clairement à partir de quel moment le processus de construction hypothétique d'un objet composite peut propager la production d'hypothèses sur ses composants.

Il convient dans un premier temps de se demander si la construction de versions des composants peut avoir lieu à n'importe quel niveau de raffinement de l'objet composite. En effet, pourquoi ne pas déclencher cette construction dès l'apparition de chaque composant dans l'objet composite ? Par exemple, pourquoi ne pas déclencher la construction de tous les moteurs possibles dès leur création dans une instance de voiture ?

Plusieurs raisons s'opposent à une telle solution :

- ne disposant pas de critère d'arrêt comme la contrainte de productivité, la construction hypothétique d'un composant doit alors considérer toutes les hypothèses possibles.
- le contexte de création d'un composant ne correspond pas nécessairement à un contexte de raffinement dans lequel composite et composant partagent des informations à travers les contraintes inter-attributs. La construction hypothétique du composant peut donc prendre effet à partir d'un composant vide d'informations.
- le composite ne peut distinguer les versions du composant qui constituent des hypothèses intéressantes de celles qui ne le sont pas. Il devra donc explorer chaque version du composant. Cette exploration donnera lieu à la création d'autant de versions du composite.
- lorsque plusieurs composants apparaissent, il faut considérer la construction hypothétique de chacun d'eux, puis explorer toutes les combinaisons de versions. Aucun critère de cohérence ne peut être appliqué à ces combinaisons. Dans l'exemple d'une instance composite voiture, cette combinaison peut avoir pour effet de construire des versions de voiture contenant des moteurs propres à une marque avec des portes provenant d'une seconde marque, une carrosserie particulière à une troisième marque, etc.

Afin d'éviter une telle explosion combinatoire deux niveaux de conditions sont adoptées pour circonscrire le déclenchement de la construction hypothétique des composants à un contexte pertinent. Ces deux niveaux décrivent, d'une part, l'étape à laquelle l'exploration des configurations possibles de l'objet composite peut requérir la construction hypothétique de ses composants et, d'autre part, le choix des composants à explorer et la définition pour chacun d'eux d'une contrainte de productivité.

4.1.1. Conditions liées au raffinement de l'objet composite

L'exemple d'une instance composite de voiture illustre le problème qu'introduit une construction prématurée de versions hypothétiques de composants. Le problème essentiel provient d'un déclenchement de ces constructions dans un contexte de raffinement de l'instance composite qui est trop général.

La première condition imposée se traduit alors par les mesures suivantes :

- la construction hypothétique des composants est un problème que le système se pose lorsqu'il traite une version de l'instance composite qui n'est pas une solution mais qui est **terminale** quant à l'exploration des hypothèses relatives à l'évaluation de ses attributs et de son raffinement.
- il ne peut avoir construction hypothétique des composants que si la classe de rattachement de l'objet composite est une classe fermée ou une feuille de la hiérarchie (considérée implicitement comme un cas particulier de classe fermée).

La première mesure a pour but d'isoler le problème de construction hypothétique des composants dans un contexte dans lequel la contrainte de productivité n'est pas encore satisfaite et ne peut plus l'être par la production d'hypothèses liées aux informations propres à l'objet composite. Cette mesure a l'avantage de garantir que la satisfaction ou l'insatisfaisabilité de la contrainte de productivité dépendent uniquement des possibilités d'exploration des composants. On évite ainsi le cas inutile d'exploration hypothétique de composants dans un contexte dans lequel un attribut de la contrainte de productivité, indépendant des composants et en attente d'évaluation, s'avère par la suite inévaluable dans tous les développements hypothétiques de l'objet composite qui succèdent.

La seconde mesure, quant à elle, a pour but d'éviter la construction de versions d'instances composites qui sortent des limites des schémas de composition connus du concept composite. Dans l'exemple des voitures, il s'agit d'éviter la construction de voitures irréelles dont la composition fait appel à un ensemble de composants provenant de marques ou de modèles différents.

Les mesures prises limitent donc l'exploration des possibilités de composants à un contexte de raffinement de l'objet composite correspondant à une classe fermée ou à une classe feuille de la hiérarchie. L'utilisateur ne peut ainsi construire d'instance composite qui ne corresponde pas un schéma connu.

Toutefois, ce fonctionnement pourrait être étendu en permettant à l'utilisateur de définir des contraintes de productivité impliquant les attributs des instances composantes. Dans ce cas, une première exploration hypothétique des composants pourraient avoir lieu en considérant les contraintes de productivité défini par l'utilisateur. Ce dernier pourrait alors étudier les possibilités d'instances composites combinant des versions de composants qui ne correspondent pas à un schéma de composition connu (une classe fermée ou feuille de la hiérarchie) du concept composite. Cette possibilité n'est pas étudiée par la suite et est considérée comme un prolongement logique du principe proposé.

4.1.2. Conditions liées à la contrainte de productivité

Lorsque la production d'hypothèses sur les composants peut être déclenchée à partir d'une version de l'instance composite, il faut alors déterminer quels sont les composants pertinents pour le problème posé. En effet, l'exploration hypothétique d'un composant n'a d'intérêt qu'à partir du moment où il a pu être établi que la contrainte de productivité de l'objet composite dépend de la configuration de ce composant.

Ce peut être le cas dès qu'un composant partage des informations avec le composite au travers de contraintes inter-attributs. En effet, si des attributs du composant sont ainsi impliqués dans la description du composite, l'évaluation d'un attribut de la contrainte de productivité définie sur le composite peut donc dépendre de l'attribut du composant.

Par exemple, l'attribut *PrixHT* d'une instance de voiture de la classe *Modèle-M* dépend de l'attribut *Prix* de son instance de moteur. De plus, puisque l'attribut *PrixTTC* de l'instance de voiture dépend de son attribut *PrixHT*, il dépend donc indirectement de l'attribut *Prix* de son instance de moteur. Si la contrainte de productivité a pour but l'évaluation de l'attribut *PrixTTC*, il est donc important d'évaluer l'attribut *Prix* du moteur.

Soient I_C une instance composite et $CP(I_C)$ la contrainte de productivité dont elle fait l'objet. Si I_C correspond à une situation où seule la production d'hypothèses sur les composants pourrait encore permettre la satisfaction de $CP(I_C)$ alors, pour savoir quels sont

parmi ses composants ceux qui constituent une voie d'exploration intéressante, il faut établir quels sont les composants impliqués dans l'évaluation des attributs de $CP(I_C)$.

Le contrôle d'évaluation des attributs reposent sur une gestion de liens de dépendance qui permet de mettre en place un mécanisme de recherche des attributs dont peut dépendre l'évaluation d'un attribut. Pour un attribut A deux types de dépendances sont envisageables. L'attribut A faisant l'objet d'un détachement procédural, il dépend de l'évaluation des attributs attendus en entrée de ce détachement procédural. L'attribut A faisant l'objet d'une inférence par réduction de domaine, il dépend des modifications que les attributs peuvent apportées au réseau de contraintes auquel il appartient. Ce deuxième type de dépendance pose certains problèmes.

4.1.2.1. Problèmes liés au fonctionnement des contraintes

Les contraintes possèdent un fonctionnement qui compliquent singulièrement la recherche des attributs qui supportent le calcul d'un autre. En effet, un attribut en attente d'inférence par réduction de domaine n'exige pas que les autres attributs du réseau de contraintes soient tous évalués. Son fonctionnement consiste à attendre que l'ensemble des domaines des attributs du réseau de contrainte soient suffisamment réduits pour permettre la réduction à un singleton du domaine de l'attribut en attente.

Par exemple, si on considère la contrainte suivante :

- *PrixHT = PrixBase + Moteur.Prix + Carrosserie.Prix*

et si, de plus, l'attribut *PrixHT* est en attente d'une inférence par réduction de domaine, cette inférence n'exige pas que les trois attributs *PrixBase*, *Moteur.Prix* et *Carrosserie.Prix* soient tous simultanément évalués. L'évaluation de *PrixHT* est uniquement dépendante des réductions de domaine de ces attributs.

Pour étudier toutes les solutions possibles, il faudrait donc explorer séparément toutes les situations de réduction de chacun de ces attributs. Puis, si des situations de réduction d'un attribut ne suffisent pas à elles seules à évaluer *PrixHT*, il faudrait alors considérer les combinaisons de situations de réduction provenant de deux attributs, puis si besoin des trois attributs.

Au niveau de l'exploration des composants, cela signifie qu'il faudra chercher toutes les situations de réduction qui peuvent entraîner l'évaluation d'un attribut du composite appartenant à la contrainte de productivité. Ensuite, si plusieurs composants sont impliqués dans l'évaluation de cet attribut, il faudra explorer les différentes combinaisons de réduction provenant de chaque composant.

4.1.2.2. Interprétation restreinte du fonctionnement des contraintes

Le fonctionnement des inférences par réduction de domaine implique dans le contexte de l'exploration hypothétique des composants un traitement complexe et pouvant provoquer une profusion de versions hypothétiques aussi bien de composants que du composite. Par la suite, afin d'éviter ces traitements, la recherche des attributs de composants pouvant supporter l'évaluation d'un attribut du composite interprète une inférence par réduction de domaine comme un détachement procédural. C'est-à-dire que, du point de vue d'un attribut en attente d'une inférence par réduction de domaine, l'évaluation de cet attribut est supposée dépendre de l'évaluation de tous les attributs impliqués dans cette contrainte.

Cette mesure restreint les possibilités inférantes des contraintes mais permet surtout de limiter le type d'informations que le système d'assistance recherche lors de l'exploration hypothétique des composants. En effet, les conditions d'évaluation d'un attribut de l'objet composite peuvent alors être exprimées en termes d'évaluation d'attributs de ses composants.

4.2. Propagation de l'exploration hypothétique sur les composants

La mesure restrictive concernant l'exploitation des inférences par réduction de domaine permet de calculer pour chaque attribut appartenant à la contrainte de productivité l'ensemble des attributs des composants dont dépend son évaluation. Pour satisfaire cette contrainte de

productivité, il faut donc trouver les versions de composants permettant l'évaluation de l'ensemble de ces attributs.

En d'autres termes, la contrainte de productivité de l'objet composite est redéfinie en termes de contraintes de productivité sur ses composants. Chaque composant auquel le composite impose une telle contrainte peut alors faire l'objet d'une configuration par assistance hypothétique. Toute version solution d'un composant constitue une nouvelle hypothèse à explorer pour le composite. L'exploration au niveau du composite consiste alors à combiner les hypothèses provenant de chaque exploration hypothétique de composants.

4.2.1. Attributs de composant supportant un attribut du composite

Soit *Support* (A, I_C) la fonction qui permet de rechercher pour un attribut A de l'instance composite I_C l'ensemble des attributs de ses composants qui, d'une part, supportent directement ou indirectement l'évaluation de A dans l'état courant de I_C et, d'autre part, ne sont ni inconnus, ni évalués. Le résultat de cette fonction est un ensemble de couples de la forme suivante :

- $(I_P, \{A_P^1..., A_P^n\})$ où I_P est une instance composante de I_C et chaque A_P^i, $1 \leq i \leq n$, est un attribut de I_P impliqué dans l'évaluation de A de I_C.

Pour obtenir ces informations, la fonction *Support* doit parcourir le réseau de dépendances formé par les inférences en cours d'exécution (état d'attente) en partant de l'inférence qui a en charge l'évaluation de A. Son fonctionnement est le suivant :

- Si A fait l'objet d'une inférence par réduction de domaine et appartient à un réseau $R(I_C)$ de contraintes :
  - tous les attributs de I_C appartenant au réseau $R(I_C)$ et qui sont en attente du résultat de leur DP font eux-même l'objet de la fonction *Support*.
  - tous les attributs appartenant à une instance composante qui apparaissent dans le réseau de contrainte et sont en état d'attente (**Demandeur**, **Indécis** ou **Bloqué**) sont retenus. C'est-à-dire qu'ils sont ajoutés à l'ensemble des attributs support d'évaluation de l'attribut A associé à l'instance composante à laquelle chacun d'eux appartient.
- Si A fait l'objet d'une inférence par DP :
  - tous les attributs de I_C appartenant aux entrées du DP et qui sont en attente du résultat d'une inférence font eux-même l'objet de la fonction *Support*.

Un marquage des attributs de I_C déjà explorés permet d'éviter les cycles dans la recherche et en assure donc la terminaison.

Si on reprend l'exemple de la voiture (cf. §VIII.4.1.2.1) dans lequel l'attribut *PrixTTC* est le but défini par la contrainte de productivité, son calcul dépend de l'attribut *PrixHT* qui dépend lui-même des attributs *Moteur.Prix* et *Carrosserie.Prix*, le calcul des attributs composants supportant *PrixTTC* donne alors :

- *Support* (*PrixTTC*, *I-Voiture*) = {(*I-Moteur*, {*Prix*}) ; (*I-Carrosserie*, {*Prix*})

où *I-Voiture* est l'instance composite, *I-Moteur* et *I-Carrosserie* deux de ses instances composantes.

A partir de la fonction *Support*, la contrainte de productivité d'un objet composite peut être réinterprétée en termes de contraintes de productivité sur ses composants.

4.2.2. Contrainte de productivité imposée à un composant

Pour une instance composite I_C qui correspond dans une situation de production d'hypothèses sur ses composants, la première étape consiste à trouver à partir de tous les attributs de $CP(I_C)$ l'ensemble des ses composants, ainsi que leurs attributs, pouvant participer à la satisfaction de $CP(I_C)$. Deux cas peuvent se présenter :

- il existe un attribut A de $CP(I_C)$ tel que *Support* (A, I_C)=Ø.

  Dans ce cas, l'évaluation de A ne dépend pas des composants de I_C. A ne pourra donc pas être évalué. L'instance I_C ne peut mener à une solution.

- sinon, les ensembles trouvés par application de la fonction *Support* à chaque attribut de $CP(I_C)$ sont réunies en un seul. Chaque couple de cet ensemble représente l'association entre une instance composante et l'ensemble de ses attributs recensés au moins une fois par la fonction *Support*.

A l'issue de cette démarche, si la production n'a pas été abandonnée, un ensemble de couples similaire à ceux rendus par la fonction *Support* est obtenu. Chacun de ces couples possède la forme $(I_P, \mathcal{A}_{CP(Ic)}(I_P))$, où I_P est une instance composante de I_C et $\mathcal{A}_{CP(Ic)}(I_P)$ est l'ensemble des attributs de I_P supportant les attributs de $CP(I_C)$.

Pour chaque couple $(I_P, \mathcal{A}_{CP(Ic)}(I_P))$, le système lance une configuration hypothétique de I_P avec la contrainte de productivité $\mathcal{A}_{CP(Ic)}(I_P)$.

4.2.3. Combinaison des versions solutions des composants

Afin que le composite ne soit pas altéré lors de la construction des versions hypothétiques de ses composants, chacune de ces productions s'effectue dans un contexte indépendant de l'objet composite. A l'issue de ces phases de productions, l'instance composite obtient pour chaque instance composante I_P son ensemble de versions solutions $\mathcal{V}_{Sol}(I_P)$. Chacune de ces versions représente une hypothèse d'instance composante I_P.

Le système communique au système de gestion de version ces hypothèses qui les combinent entre elles pour former de nouvelles versions de l'instance composite. Selon le principe habituel, le système d'assistance peut explorer ces nouvelles versions de l'instance composite pour vérifier qu'elles sont cohérentes et qu'elles satisfont la contrainte de productivité.

5. Conclusion

Ce chapitre constitue la dernière étape d'extension du modèle TROPES dans laquelle prennent place l'ensemble des propositions formulées dans le chapitre V.

Globalement, cette dernière extension vise à dépasser le cadre du raisonnement déductif, jusqu'alors proposé par le système, pour permettre l'exploration des différentes possibilités offertes lors d'une situation de connaissances incomplètes. L'indécision qu'introduit une telle situation, bloquante pour un raisonnement déductif, peut être résolue si des hypothèses peuvent être produites, introduites dans la base de connaissances, et être exploitées par le raisonnement. Cette constatation justifie l'effort d'intégration d'une composante hypothétique dans l'architecture du modèle TROPES. Le résultat global enrichit le système d'un niveau d'assistance hypothétique.

Dans ce contexte, le problème de l'identification/construction d'instances peut être traité de façon générale comme un problème de configuration hypothétique d'instance. Un tel problème est formulé par un utilisateur par la pose d'une *contrainte de productivité* sur l'instance à configurer. Cette contrainte permet à l'utilisateur de préciser en termes d'évaluation d'attributs les informations qu'il requiert dans son instance. La résolution du problème consiste à trouver des versions de l'instance qui sont cohérentes et satisfont la contrainte de productivité.

Le schéma de résolution adopté est celui d'un processus d'exploration des différentes évolutions possibles de l'instance permises par la description des objets. Chacune de ces évolutions se caractérise par un ensemble différent de propositions de raffinement de l'instance et de choix d'inférence d'attributs à déclencher. L'ensemble de ce processus est mis en œuvre par le mécanisme de configuration hypothétique d'instance proposé dans le système d'assistance hypothétique. Ce mécanisme permet de construire l'ensemble des versions hypothétiques d'une instance qui satisfont une contrainte de productivité fixée par l'utilisateur.

Pour ce mécanisme, la contrainte de productivité représente une contrainte posée sur l'état d'évaluation des attributs de l'instance. A chaque étape d'exploration, l'ensemble des

informations fournies par le contrôle d'évaluation des attributs est confronté à cette contrainte. Cette opération permet d'établir si une voie d'exploration reste prometteuse, devient inintéressante ou fournit déjà une solution au problème posé.

La démarche d'exploration mise en place s'inspire de la démarche incrémentale du raffinement de l'instance. A chaque étape, si l'état de l'instance à traiter ne satisfait pas déjà la contrainte de productivité, l'ensemble des informations possibles pour poursuivre le raffinement de l'instance sont recensées. Ces informations sont proposées comme hypothèses et combinées pour construire des versions de l'instance qui seront traitées lors d'une prochaine étape du mécanisme.

Le raisonnement hypothétique mené lors de la résolution est donc dirigé par les données et contrôlé grâce à la contrainte de productivité. Toutefois, lorsque ce raisonnement est appliqué à un objet composite, il est inadapté lorsque la satisfaction de la contrainte de productivité requiert l'exploration hypothétique des composants.

Afin d'éviter une explosion combinatoire provoquée par une exploration systématique et aveugle des différentes possibilités de configuration de composants, cette exploration est différée à la fin du traitement de l'instance composite. L'exploration sur les composants peut donc avoir lieu à partir de versions du composite pour lesquelles l'exploration des composants reste la seule solution possible pour satisfaire la contrainte de productivité posée.

La propagation du processus d'exploration d'un objet composite sur ses composants correspond à un schéma de décomposition d'un problème en sous-problèmes. En effet, à partir de l'état de contrôle des attributs, le système d'assistance établit quels sont les composants et, plus précisément, leurs attributs qui peuvent intervenir dans l'évaluation des attributs de la contrainte de productivité. Ces informations obtenues, la contrainte de productivité imposée à l'objet composite est reformulée en contraintes de productivité sur les composants.

L'exploration sur les composants consistent donc à trouver l'ensemble des versions d'instances composantes satisfaisant la contrainte de productivité qui leur a été associée à travers le composite. Ces versions de composants constituent les nouvelles hypothèses émisent sur le composite. Elles sont combinées pour former de nouvelles versions du composite qui sont ensuite explorées pour vérifier si la contrainte de productivité est satisfaite.

Pour les besoins du traitement des objets composites, une interprétation approximative du comportement des contraintes inter-attributs a été employée afin de réduire les voies d'exploration possibles. En effet, en toute théorie, un attribut qui fait l'objet d'une inférence par réduction de domaine ne dépend pas de l'évaluation de tous les attributs impliqués dans le même réseau de contraintes que lui. L'évaluation de cet attribut ne dépend que des réductions de domaine subies par les autres attributs. Non seulement ce fonctionnement introduit des disjonctions dans les dépendances entre attributs mais, de plus, ces dépendances ne traduisent pas un besoin d'évaluation d'attributs. En pratique, considérer que les contraintes introduites entre composite et composants expriment avant tout des propagation de valeurs est une approximation de leur comportement qui nous semble convenable.

Conclusion

CONCLUSION

1. Bilan

Le thème central étudié dans ce mémoire est celui de la dualité entre l'identification et la construction d'objets. De nombreux travaux soulignent l'intérêt d'exploiter la représentation des connaissances par objets selon chacun de ces deux aspects.

Cependant, l'étude menée fait apparaître une partition des modèles à objets en deux catégories. Les modèles de l'une tirent parti de la description des objets pour la construction et la gestion d'objets, tandis que les modèles de l'autre s'appuient sur une description définitionnelle des objets pour mettre en place un raisonnement classificatoire.

Les premiers proposent une description riche des objets dans laquelle de nombreuses et diverses notions contribuent à associer à chaque type d'entités, introduit par un domaine d'application particulier, son comportement propre et complexe. Le foisonnement de concepts de cette approche s'explique par l'intégration d'idées provenant aussi bien des travaux sur la représentation des connaissances, notamment des travaux autour de la notion de *frames,* que des travaux sur les langages de programmation. La description des objets en une hiérarchie de classes est fortement motivée par la possibilité d'exprimer le partage de connaissances entre différents types d'objets. Les possibilités d'héritage de connaissances d'une classe vers ses sous-classes constituent l'élément moteur de cette approche. Le choix d'une classe pour la création d'une instance permet à celle-ci de bénéficier des informations provenant aussi bien de la description de la classe que de celles de ses sur-classes.

Les seconds proposent une description restreinte et stricte des objets dont l'objectif est avant tout de dégager les propriétés des objets sur lesquelles un raisonnement classificatoire peut s'appuyer. Contrairement aux modèles précédents, l'organisation hiérarchique des objets n'est plus motivée par son rôle de réservoir d'héritage de connaissances mais par les relations d'inclusion ensembliste entre classes d'objets qu'elle met en évidence. L'intérêt de cette interprétation est d'exploiter la description des objets pour trouver en fonction de la structure et des valeurs d'attributs d'une instance sa ou ses classes d'appartenance. Le mécanisme qui met en place ce raisonnement est la classification d'instances. Dans les modèles, dits classificatoires, la construction d'une instance est considérée comme un problème externe à la description des objets. L'expression des connaissances est donc dédiée à la description de la structure d'instances et du domaine de valeurs des attributs.

L'analyse d'applications, dans lesquelles la résolution de problèmes est ramenée à un raisonnement cherchant à la fois à identifier et à construire des objets, montre l'intérêt de concilier ces deux aspects au sein du même modèle à objets. Par exemple, dans un domaine comme l'aide à la conception d'objets, la démarche de conception d'un objet peut être décomposée en deux phases. La première phase consiste à identifier, au vu des caractéristiques particulières du problème, les schémas de conception les plus appropriés. La seconde phase vérifie l'adéquation des schémas sélectionnés en essayant de construire une solution. Une démarche similaire peut être constatée dans un raisonnement d'aide au diagnostic.

Différents travaux sur la représentation par objets se sont intéressés à cette démarche. Leur approche consiste à adapter un modèle classificatoire de telle façon que le mécanisme de classification puisse exploiter des connaissances introduite dans la description pour construire l'instance, ou plus précisément la compléter.

Toutefois, une étude critique de cette approche permet de constater que l'importance donnée au raisonnement classificatoire limite l'exploitation de la description des objets et, plus généralement, tend à orienter la description des connaissances en fonction du type de problèmes à résoudre. Ce constat sous-entend donc que la représentation des connaissances est dépendante de son utilisation. Cette solution est acceptable dans une base de connaissances dédiée à une application précise dans lequel le schéma de raisonnement est connu et figé. Mais elle devient

restrictive dans le cas de grandes bases de connaissances qui peuvent être partagées par plusieurs utilisateurs-concepteurs et dont l'exploitation peut changer en fonction des connaissances que possède et requiert un utilisateur particulier.

Les enseignements tirés de l'étude des différents types de modèles à objets et, plus particulièrement, de ceux proposant une solution mixte, permettent de définir une nouvelle approche de l'identification/construction d'instances. Cette nouvelle approche se caractérise par un ensemble de propositions concernant les deux aspects indissociables d'un modèle :

- le modèle à objets doit favoriser une représentation se limitant à la description des propriétés des objets.

  Concrètement, la proposition suivante est adoptée : une classe est un ensemble de contraintes. Si une instance satisfait ces contraintes elle appartient à la classe ; si son appartenance à la classe est affirmée, elle se conforme aux contraintes imposées par cette classe. Les différents aspects de la description des objets sont ainsi interprétés en termes de contraintes : les relations qu'une classe entretient avec les autres classes de la hiérarchie, la structure d'une instance, le domaine de valeurs des attributs, et la stratégie d'évaluation des attributs.

- le raisonnement lié au problème de l'identification/construction d'instances n'est pas un cas particulier de raisonnement classificatoire. Au contraire, l'exploitation de la classification dans la résolution de ce type de problème n'est pertinente que dans certains cas particuliers. Un tel cas apparaît quand les connaissances fournies pour décrire le problème permettent aussi de l'identifier complètement et donc, de le résoudre.

  Dans le cas général, un problème d'identification/construction d'instance se caractérise par la mise en évidence de connaissances incomplètes. Pour remédier au problème des connaissances incomplètes, le mécanisme de résolution doit développer un raisonnement hypothétique. Ce raisonnement permet de prolonger l'exploration de la description des objets au delà des limites atteintes par la classification.

Ces propositions sont mises en application dans le modèle TROPES. La démarche suivie est ascendante puisqu'elle consiste

1) à étendre le modèle classificatoire en intégrant des mécanismes d'inférence d'attributs ;
2) à mettre en place un contrôle d'évaluation d'attributs permettant d'exprimer et de développer la stratégie d'évaluation des attributs en fonction du raffinement d'une instance ;
3) à étendre l'architecture du modèle pour mettre en place le raisonnement hypothétique nécessaire à la résolution d'un problème d'identification/construction d'instances.

Dans ce contexte, un problème d'identification/construction d'instances est résolu par application d'un mécanisme de configuration hypothétique d'instances. A partir de ce mécanisme, un problème est exprimé en indiquant l'instance à construire et en précisant les attributs de l'instance que l'utilisateur souhaiterait connaître.

Le résultat fourni par ce mécanisme est un ensemble des versions de l'instance qui offrent une réponse cohérente à la demande de l'utilisateur. Chacune de ces versions est le résultat d'un ensemble d'hypothèses que le mécanisme de configuration hypothétique émet lorsque le raffinement de l'instance ne peut plus être assuré par la classification. Une hypothèse représente un choix répondant à une situation d'indécision soulevée par le mécanisme de classification : choix d'un mécanisme d'évaluation pour un attribut ou choix d'une classe possible pour le rattachement.

L'intérêt de cette approche est double puisqu'elle permet à la fois de considérer des problèmes à solutions multiples et différents types de problèmes à partir d'une même description d'objets.

2. Perspectives

Cette nouvelle approche de la représentation des connaissances par objets suggère de nombreuses perspectives à explorer :

- **Classifications multiples** :

  L'étude proposée ne prend pas en compte la possibilité offerte par le modèle TROPES d'exprimer différents points de vue sur un même concept. La notion de point de vue permet de décrire un même concept d'objets selon différentes taxonomies de classes. Chaque taxonomie propose sa propre organisation du concept et fournit un support de classification d'instance indépendant. La notion de passerelle permet à la classification de déduire l'appartenance d'une instance à des classes d'un point de vue en se basant sur les résultats obtenus sur d'autres points de vue.

  Dans le cadre de la résolution de problème d'identification/construction d'instances, la modélisation d'un concept selon plusieurs points de vue favorise la réduction du nombre d'hypothèses de rattachement à émettre. En effet, la multiplicité des taxinomies augmente les possibilités d'identification d'une instance incomplète. Selon le point de vue considéré, l'état incomplet d'une instance peut avoir moins d'incidence et permettre une descente conséquente de l'instance dans la hiérarchie. Grâce aux passerelles, la classification peut donc établir l'appartenance de l'instance à des classes bien qu'elle soit incomplète relativement à la description de ces classes.

  Le problème essentiel que soulève la prise en compte de la notion de point de vue, réside dans le contrôle d'évaluation d'attributs. Il est effectivement nécessaire d'adapter le modèle de contrôle d'évaluation proposé pour prendre en compte non seulement les possibilités d'évaluation d'un attribut selon les différents points de vue mais aussi une stratégie d'évaluation globale capable de s'adapter en fonction du point de vue. Pour préserver l'esprit de la notion de point de vue, la solution consistant à décrire la stratégie d'évaluation d'un attribut en fonction de chaque point de vue semble une voie de recherche intéressante. Dans ce cas, la principale difficulté réside dans la coexistence possible de plusieurs stratégies pour un même attribut.

- **Hiérarchie de classes et propriétés particulières de classes :**

  La version du modèle TROPES, qui a fait l'objet d'extensions dans ce mémoire, impose des contraintes strictes sur la hiérarchie des classes. Les classes sont définitionnelles, elles décrivent des conditions nécessaires et suffisantes d'appartenance, la décomposition d'une classe en sous-classes est exclusive et la spécialisation multiple est impossible. Il serait intéressant d'étudier quand et comment lever ces contraintes.

  La notion de classes descriptives, présente dans FROME ou dans les langages terminologiques, peut être prise en compte facilement par le mécanisme de configuration hypothétique d'instances. En effet, l'appariement d'une instance à une telle classe ne pouvant rendre un statut sûr, le mécanisme émettra une hypothèse de rattachement si elle est possible.

  La levée de la contrainte d'exclusivité augmente sensiblement le nombre de combinaisons d'hypothèses à prendre en compte durant l'exploration de la hiérarchie. Pour éviter une profusion d'hypothèses de rattachement, il serait toutefois bon de bénéficier d'informations concernant les cas connus d'exclusion de classes, comme dans FROME ou SHOOD, par exemple.

  Du point de vue du contrôle d'évaluation d'attributs, la levée de la contrainte d'exclusivité introduit des cas d'indécision particuliers puisqu'une instance peut alors être rattachée simultanément à deux classes incomparables par la relation de spécialisation. Cette situation correspond à celle des conflits d'inférences d'attributs de SHOOD. Deux solutions peuvent être envisagées : contraindre les deux inférences à rendre un même résultat, ou gérer deux hypothèses.

La spécialisation multiple de classes implique que l'exclusivité de la décomposition de classes en sous-classes ne soit plus assurée. Elle renvoie donc aux problèmes précédents.

En règle générale, les propriétés restrictives des hiérarchies de classes proposées par TROPES nous semblent acceptables, étant données les possibilités de descriptions multiples introduites par la notion de point de vue. Elles possèdent l'avantage d'imposer une rigueur méthodologique dans la constitution des hiérarchies de classes. La levée de l'une de ces contraintes ne devrait être envisagée que ponctuellement, provenant d'un acte volontaire et réfléchi du concepteur d'une hiérarchie.

- **Stratégie d'évaluation des attributs :**

  Les règles permettant d'exprimer les priorités entre les différentes situations d'évaluation d'attribut étendent le principe du masquage en permettant de prendre en compte les possibilités d'évaluation qui n'ont pu ni être accédées, ni être écartées. Des outils de visualisation du résultat de l'application d'une priorité et de vérification de cohérence d'un ensemble de règles constituent une perspective intéressante lors de la phase de conception ou de phases de modification de la description des connaissances.

  La prise en compte de comportement hypothétique d'un mécanisme d'évaluation est une perspective à court terme. Le chapitre VIII apporte à ce sujet de premiers éléments de réponse.

  L'impact de la révision d'une instance sur la stratégie d'évaluation des attributs est un problème qui doit faire l'objet d'une étude complète. Notamment, il est important de cerner le problème de l'impact que peut avoir la pose dynamique de contraintes sur la stratégie d'évaluation.

- **mécanisme de configuration hypothétique d'instances :**

  A travers ce mécanisme, un problème est exprimé par la donnée d'une contrainte de productivité. Cette contrainte se réduit à la donnée des attributs à évaluer. Une perspective intéressante est d'étendre les possibilités d'expression de cette contrainte pour en faire un véritable langage de requête. Notamment, il peut être utile d'introduire des disjonctions dans la requête (l'évaluation de cet attribut *ou* de cet autre), d'introduire des choix de valeurs hypothétiques pour certains attributs, restreindre les sources de connaissances possibles d'un attribut, etc.

  Pour résoudre un problème d'identification/construction d'instance, un utilisateur doit pouvoir l'exprimer. Il est donc nécessaire qu'il connaisse les attributs qu'il cherche à évaluer. Mettre en place une interface d'interaction permettant à l'utilisateur de construire sa requête en fonction de la progression de l'exploration de la hiérarchie est une extension souhaitable. Cette extension constitue une adaptation du principe d'interaction entre le mécanisme de classification et l'utilisateur.

Une perspective d'application, qui s'avère à la source de cette étude, est l'exploitation du mécanisme d'identification/construction d'instance dans le modèle de tâches proposé dans TROPES [Gens&92] [Gens&94]. Une tâche étant représentée comme un objet composé d'objets sous-tâches, son exécution peut être ramenée à un problème d'identification/construction d'instance dans lequel les attributs représentant les sorties de la tâche forment la contrainte de productivité. Le résultat de cette exécution est dans ce cas décrit par l'ensemble des versions de la tâche dont l'exécution a abouti.

Le passage de l'application MYOSIS du modèle SHIRKA au modèle TROPES est une perspective qui devient réaliste. Elle serait d'autant plus intéressante que l'expérience acquise avec SHIRKA permet de bénéficier d'une première modélisation du domaine sous forme de hiérarchies de classes et que la mise en place d'une session de diagnostic repose sur la classification d'instances. Enfin, l'étude d'autres domaines d'application comme l'aide à la conception, la gestion de configuration, etc., constitue une perspective à plus long terme.

Annexe

Annexe
GESTION DE LA STRATEGIE D'EVALUATION D'UN ATTRIBUT

Le développement de la stratégie d'un attribut prend place lors de son apparition dans l'instance et évolue en fonction du raffinement de l'instance et des échecs d'inférence (cf. §VII.4). La mise à jour continuelle de l'état de la stratégie d'un attribut A permet au contrôle de répercuter au niveau de chaque phase l'impact d'un nouvel événement :

- tant que le contrôle reste dans la phase de sélection, la mise à jour de Max(A, I) après chaque raffinement de l'instance I permet l'obtention de App(A, I) l'ensemble des déclencheurs applicables.
- tant que le contrôle reste dans la phase d'exécution, d'une part, la mise à jour de Max(A, I) est assurée dans l'éventualité d'une nouvelle transition vers la phase de sélection (échec) et, d'autre part, la gestion des échecs stratégiques des inférences par réduction de domaine (reprise) est mise en place à chaque nouvelle étape de raffinement.

Si un état de la stratégie d'un attribut A est conceptuellement décrit par la partition de $\mathfrak{DT}(A)$ en Desact(A, I), Act(A, I) et Inact(A, I), les propriétés de la structure de $\mathfrak{DT}(A)$ permettent de réduire sensiblement les informations de contrôle nécessaires à sa gestion. En effet, l'approche adoptée vise à ramener cette gestion à celle des mises à jour de l'ensemble Max(A, I). La gestion de cet ensemble s'articule autour des deux opérations de base que peut entreprendre le contrôle d'évaluation d'un attribut : la **désactivation** et l'**application** de déclencheur.

La première partie présente comment la gestion de l'état peut être ramenée à celle des **désactivations** et des **applications** de déclencheurs. La seconde partie décrit la contribution de chaque type d'événements (apparition d'un attribut, raffinement de l'instance, échec d'inférence) au développement de la stratégie d'un attribut.

1. Principe de désactivation et d'application de déclencheurs

La prise en considération des priorités permet de réduire le problème de la gestion des états de la stratégie d'un attribut à celui de la gestion des applications et des désactivations de déclencheurs de l'ensemble Max(A, I).

1.1. Vers une gestion de Max(A, I) par désactivation de déclencheurs

Le principe adopté s'appuie sur le constat que tout déclencheur qui est supérieur à au moins un déclencheur de Max(A, I), selon l'ordre $\rightarrow$, est un déclencheur désactivé. En effet, par définition de Max(A, I), il ne peut exister un déclencheur activé ou inactivé qui soit supérieur à un déclencheur de Max(A, I).

Soit $\mathbf{Sup_{Max}(A, I)}$ l'ensemble des déclencheurs qui sont supérieurs à ceux de Max(A, I), on a donc :

- $\mathbf{Sup_{Max}(A, I)} \subseteq$ Desact(A, I)

Soit $\mathbf{Inf_{Max}(A, I)}$ son complémentaire par rapport à $\mathfrak{DT}(A)$:

- $\mathbf{Inf_{Max}(A, I)} = \mathfrak{DT}(A) \setminus \mathbf{Sup_{Max}(A, I)}$

Etant donné que l'ensemble des déclencheurs désactivés croît en fonction du développement de la stratégie d'évaluation de A, l'évolution de l'ensemble $\mathbf{Sup_{Max}(A, I)}$ est croissante. Par définition, lorsque $\mathbf{Sup_{Max}(A, I)}$ croît, $\mathbf{Inf_{Max}(A, I)}$ décroît en conséquence.

L'opération par laquelle un déclencheur passe de l'ensemble $\mathbf{Inf_{Max}(A, I)}$ à $\mathbf{Sup_{Max}(A, I)}$ est appelée **désactivation** du déclencheur.

L'ensemble $\mathbf{Inf_{Max}(A, I)}$ possède une première propriété importante :

- L'ensemble des déclencheurs maximaux de $\mathbf{Inf_{Max}(A, I)}$ pour l'ordre $\rightarrow$ est exactement l'ensemble **Max(A, I)**.

Cette première propriété permet de mettre en évidence que toute **désactivation** de déclencheur de $\mathbf{Inf_{Max}(A, I)}$ provient d'un déclencheur appartenant à Max(A, I).

Enfin, l'ensemble $\mathbf{Inf_{Max}(A, I)}$ possède une deuxième propriété importante pour le contrôle d'évaluation :

- $Sel(A, I) \subseteq \mathbf{Inf_{Max}(A, I)}$; sachant que $Sel(A, I) = Act(A, I) \cup Inact(A, I)$.

Cette seconde propriété indique que la gestion de l'état de la stratégie de l'attribut A peut se limiter aux déclencheurs de $\mathbf{Inf_{Max}(A, I)}$ puisque tous ceux susceptibles d'être impliqués dans les phases de sélection et d'exécution lui appartiennent. Notamment, la sélection d'un déclencheur **applicable** peut y être assurée puisque Act(A, I) et Max(A, I) sont tous deux inclus dans $\mathbf{Inf_{Max}(A, I)}$.

Par conséquent, le problème de la gestion de l'état de développement de la stratégie d'un attribut A peut être ramené à celui de la gestion à la fois de la **désactivation** et de l'**application** de déclencheurs. Cette gestion s'articule autour des mises à jour de l'ensemble Max(A, I) qui devient donc le support principal du contrôle. Pour mettre en place cette gestion, le contrôle doit de plus maintenir à jour l'ensemble des déclencheurs en attente de désactivation et l'ensemble des déclencheurs en attente d'application. Ces deux ensembles sont inclus dans $\mathbf{Inf_{Max}(A, I)}$ et se caractérisent par le fait que leurs mises à jour proviennent des événements apparaissant dans les phases de sélection ou d'exécution d'inférences.

1.2. Déclencheurs en attente de désactivation ou d'application

La **désactivation** résulte d'un événement qui permet de mettre en évidence de nouveaux déclencheurs désactivés. Elle dépend donc des évolutions que peuvent faire subir le raffinement de l'instance ou l'échec d'une inférence à l'ensemble Desact(A, I).

Quant à elle, l'**application** provient d'un événement qui permet soit la mise en évidence d'un nouveau déclencheur activé, soit d'une mise à jour de l'ensemble Max(A, I) qui rend prioritaire un déclencheur activé. Elle dépend donc soit des évolutions que peut faire subir le raffinement à l'ensemble Act(A, I), soit des évolutions que peut faire subir à l'ensemble Max(A, I) la **désactivation** des déclencheurs.

Aussi, techniquement, le contrôle gère l'état de la stratégie d'un attribut A en lui associant et en maintenant à jour les quatres sous-ensembles de déclencheurs suivants :

- **Max(A, I)**, l'ensemble des déclencheurs non désactivés prioritaires, à partir duquel application et désactivation de déclencheurs sont mises en place.
- **App(A, I)**, l'ensemble des déclencheurs applicables obtenu à chaque phase de sélection. Lors de la phase d'exécution d'inférence, cet ensemble dénote donc le déclencheur qui est en cours d'application.
- $\mathbf{Att_{Des}(A, I)}$, l'ensemble des déclencheurs de $\mathbf{Inf_{Max}(A, I)}$ en attente de **désactivation**. C'est-à-dire l'ensemble des déclencheurs de Desact(A, I) qui ne sont pas prioritaires :

$$\mathbf{Att_{Des}(A, I) = Desact(A, I) \setminus Sup_{Max}(A, I)}$$

- **$Att_{App}(A, I)$**, l'ensemble des déclencheurs de **$Inf_{Max}(A, I)$** en attente d'**application**. C'est-à-dire l'ensemble des déclencheurs activés qui ne sont pas prioritaires :

$$Att_{App}(A, I) = Act(A, I) \setminus App(A, I)$$

La **désactivation** peut intervenir quelle que soit la phase de contrôle, sélection ou exécution. Elle résulte d'un événement qui permet de mettre en évidence de nouveaux déclencheurs désactivés ; c'est-à-dire dès que de nouveaux déclencheurs sont ajoutés à $Att_{Des}(A, I)$. Si parmi ces déclencheurs, certains appartiennent à Max(A, I), ce dernier connaît alors un processus de mise à jour par désactivation.

En revanche, l'**application** ne peut intervenir que dans la phase de sélection d'une nouvelle inférence à déclencher. Durant cette plase, chaque mise à jour de $Att_{App}(A, I)$ ou de Max(A, I) entraîne simplement le recalcul de App(A, I). L'application est décidée en fonction du résultat de ce calcul.

Comme le montre le schéma suivant (cf. Figure A.1), les situations d'application ou de désactivation de déclencheurs sont déterminées en fonction de la configuration des trois ensembles Max(A, I), $Att_{Des}(A, I)$ et $Att_{App}(A, I)$.

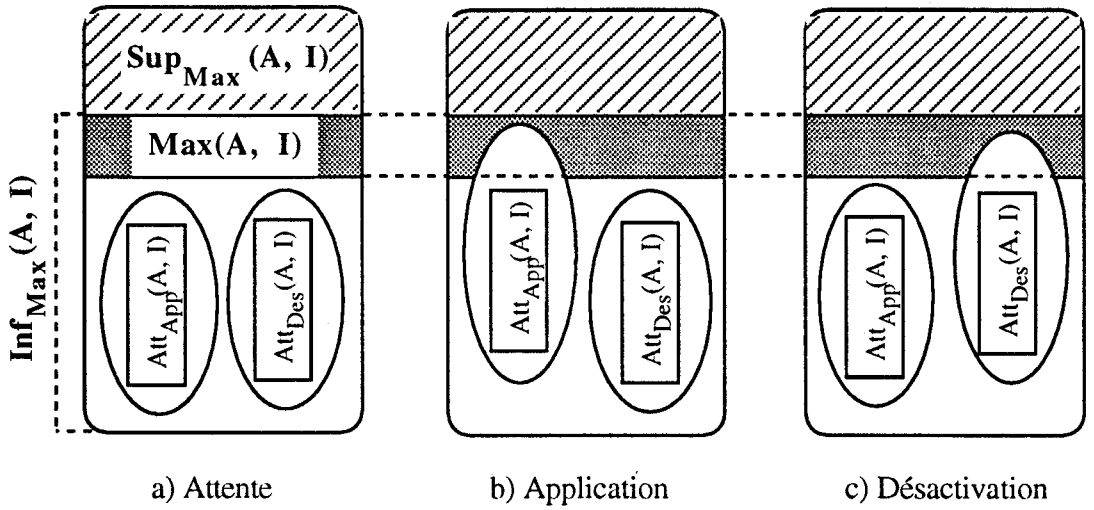

Figure A.1 : La gestion des trois ensembles Max(A, I), $Att_{App}(A, I)$ et $Att_{Des}(A, I)$, permet au contrôle de gérer les applications ou désactivations de déclencheurs. La situation a) décrit une situation d'attente du contrôle car aucun déclencheur à appliquer ou à désactiver n'est prioritaire. La situation b) (resp. c)) décrit une situation d'application (resp. désactivation) ; le déclencheur traité étant celui appartenant à l'intersection de Max(A, I) et $Att_{App}(A, I)$ (resp. $Att_{Des}(A, I)$).

La gestion de $Att_{App}(A, I)$ ne se résume pas simplement à gérer les nouveaux déclencheurs candidats à l'application. En effet, cet ensemble constitue le support utilisé par le contrôle pour gérer aussi les cas d'échec des inférences par réduction de domaine. Principalement, cette gestion vise à assurer que deux déclencheurs de source domaine ne peuvent être simultanément activés ; l'un d'eux étant alors nécessairement dans une situation d'échec. Un traitement particulier est donc mis en place lors de l'**activation** (opération d'ajout du déclencheur à $Att_{App}(A, I)$) d'un déclencheur de **source domaine**.

Le principe de la mise à jour de l'ensemble Max(A, I) par désactivation de déclencheurs et le principe des mises à jour de $Att_{App}(A, I)$ lors de l'activation d'un déclencheur de source domaine sont successivement décrits dans les deux parties suivantes.

1.3. Gestion des désactivations de déclencheurs

Pour un attribut A d'une instance I, l'ajout de nouveaux déclencheurs à $Att_{Des}(A, I)$ provoque la mise à jour de Max(A, I) par l'appel à la procédure de désactivation suivante :

1) Obtenir l'ensemble ΔDesact(A, I) des déclencheurs dont la désactivation devient effective :

   • $\Delta Desact(A, I) = Max(A, I) \cap Att_{Des}(A, I)$.

2) Propagation des désactivations et obtention des nouveaux déclencheurs maximaux.

   Pour chaque déclencheur *d* de ΔDesact(A, I), obtenir l'ensemble Succ(*d*) de ces successeurs selon l'ordre →.

   Pour chaque *d'* de Succ(*d*), faire

   - si *d'* est élément de $Att_{Des}(A, I)$, il doit être aussi désactivé. Il est donc ajouté à ΔDesact(A, I).
   - sinon, si *d'* est maximal (tous ses prédécesseurs, selon l'ordre →, sont déjà désactivés ou en cours de l'être), il est ajouté à Max(A, I).

3) Mise à jour de $Att_{Des}(A, I)$ et Max(A, I) avec les désactivation réalisées :

   • $Att_{Des}(A, I) = Att_{Des}(A, I) \setminus \Delta Desact(A, I)$
   • $Max(A, I) = Max(A, I) \setminus \Delta Desact(A, I)$

1.4. Gestion de l'activation d'un déclencheur de source Domaine

L'activation d'un déclencheur intervient quand une étape de raffinement de l'instance permet l'accès à ce déclencheur. En général l'activation signifie donc la mise en attente d'application du déclencheur (l'ajout de déclencheur à $Att_{App}(A, I)$). Cependant, lorsque l'activation concerne un déclencheur de source domaine, elle signifie aussi l'aboutissement de la phase de réductions couvert par tout autre déclencheur qui serait déjà activé (cf. §VII.4.3.1.3). S'il existe effectivement un tel déclencheur déjà activé, l'activation du nouveau déclencheur pose donc un problème au contrôle : l'un des deux est nécessairement en échec. Pour savoir lequel, le contrôle se base sur les priorités établies entre ces déclencheurs et, si besoin, l'état du domaine de valeurs de l'attribut. Il garantit ainsi que deux déclencheurs ne peuvent être simultanément dans l'état activé.

Soit *d* le déclencheur déjà activé, $d \in Act(A, I)$, tel que *d*=(C, Dom) et soit *d'*=(C', Dom) le déclencheur en cours d'activation. Puisque l'activation de *d'* intervient après celle de *d*, la classe C est nécessairement une sur-classe de C'. Entre l'activation de *d* et de *d'*, le domaine de valeurs de l'attribut A a connu les réductions de domaine imposées par toutes les étapes de raffinement allant de la classe C inclue à la classe C' exclue, son état est donc celui de la dernière réduction de la phase associée à *d*.

Lors de l'activation de *d'*, le traitement dépend du cas de figure :

• ($d' \rightarrow d$), *d'* est prioritaire sur *d*.

  L'application de *d'* primant sur celle de *d*, l'activation de *d'* entraîne dans tous les cas la désactivation de *d*.

  Le déclencheur *d* est donc supprimé de l'ensemble $Att_{App}(A, I)$, ou de l'ensemble App(A, I) s'il était en cours d'application. Il est ensuite ajouté à $Att_{Des}(A, I)$.

  Le déclencheur *d'* est quant à lui ajouté à l'ensemble $Att_{App}(A, I)$.

• ($d \rightarrow d'$), *d* est prioritaire sur *d'*.

  Deux cas sont alors possibles :

  - le domaine de valeurs de l'attribut A est déjà réduit à un singleton lors de l'activation de *d'*.

    Le déclencheur *d* décrit donc le cas d'une inférence qui est déjà en situation de réussite. Il est donc maintenu activé. L'activation du déclencheur *d'* est donc abandonnée, il est ajouté à $Att_{Des}(A, I)$.

- le domaine de valeurs de l'attribut A n'est pas réduit à un singleton lors de l'activation de *d'*.

  Le déclencheur *d* décrit le cas d'une inférence qui est déjà en situation d'échec. Le traitement est celui du cas où *d'* est prioritaire : $d \in Att_{Des}(A, I)$ et $d' \in Att_{App}(A, I)$.

Cette gestion assure donc l'échec stratégique d'une inférence par réduction de domaine en cours d'exécution et permet d'assurer que, malgré les décalages pouvant exister entre l'activation et l'application d'un déclencheur, l'état du domaine de valeurs de l'attribut est toujours cohérent avec le déclenchement d'une inférence par réduction de domaine.

2. Contribution des événements au développement de la stratégie

La mise à jour s'organise autour des opérations de désactivation et d'activation des déclencheurs. Elle correspond à la prise en compte d'un événement pouvant intervenir lors de la phase de sélection ou de la phase d'exécution. Leur contribution en termes de désactivation ou d'activation est présentée en fonction du type de l'événement.

2.1. L'événement Apparition de A dans I

Cet événement correspond à la mise en place initiale du contrôle de l'attribut A. L'ensemble Max(A, I) est donc initialisé avec $Max_0(A, I)$, l'ensemble des déclencheurs maximaux de $\mathfrak{Dl}(A)$. Les deux ensembles, $Att_{Act}(A, I)$ et $Att_{Des}(A, I)$, ont pour valeur initiale l'ensemble vide.

Si l'instance I est déjà rattachée à une classe de la hiérarchie (son lien **est-un** est évalué), le traitement de cet événement est complété par celui de l'événement de raffinement "I appartient à la classe racine". Dans le cas contraire, l'instance I étant dans sa phase de création, cet événement de raffinement particulier sera traité lorsque le rattachement de l'instance à la classe racine sera effectif.

2.2. Evénements de raffinement de l'instance I

Les deux événements de raffinement possibles sont la déclaration de non appartenance de I à l'une des classes possibles, ou la déclaration d'appartenance de I à l'une des sous-classes **directes** et possibles de la classe de rattachement courant. Toute déclaration d'appartenance provenant du processus de raffinement, elle se limite donc effectivement à la spécialisation de I dans l'une des sous-classes directes de la classe de rattachement courant.

Le traitement de chacun de ces événements est décrit dans la suite en adoptant les notations suivantes :

- $\mathfrak{Dl}_C(A)$ dénotant l'ensemble des déclencheurs de $\mathfrak{Dl}(A)$ qui ont C pour classe d'accès.
- $\mathfrak{Imp}(I \notin C)$ dénotant l'ensemble des nouvelles classes impossibles obtenu par propagation de l'événement $I \notin C$. La propagation dans ce cas consiste à déclarer impossible toutes les sous-classes de C qui restaient possibles.
- $\mathfrak{Imp}(I \in C)$ dénotant l'ensemble des nouvelles classes impossibles obtenu par propagation de l'événement $I \in C$. Dans TROPES, la propagation consiste à déclarer impossibles toutes les classes sœurs de C.

2.2.1. L'événement $I \notin C$: la classe C est déclarée impossible pour I

Le traitement de cet événement est celui de la tentative de désactivation de tous les déclencheurs dont la classe d'accès devient impossible :

1) Pour toute classe C' de $\mathfrak{Imp}(I \notin C)$, l'ensemble $\mathfrak{Dl}_{C'}(A)$ est ajouté à $Att_{Des}(A, I)$.

2) L'ensemble Max(A, I) est mis à jour en traitant les éventuelles désactivations que permet le nouvel ensemble $Att_{Des}(A, I)$.

Si cet événement intervient alors que le contrôle est dans la phase de sélection, la mise à jour de Max(A, I) entraîne ensuite le calcul de l'ensemble App(A, I) des déclencheurs applicables :

- $App(A, I) = Max(A, I) \cap Att_{Act}(A, I)$

Si un seul déclencheur est sélectionné, l'inférence correspondante est exécutée.

2.2.2. L'événement I∈C : la classe C est déclarée sûre pour I

Le traitement de cet événement donne lieu à la séquence d'étapes suivantes :

1) mise à jour de l'ensemble $Att_{Des}(A, I)$ avec les déclencheurs que l'événement permet de désactiver :

   - Pour toute classe C' de $\mathfrak{Imp}(I \in C)$, l'ensemble $\mathfrak{Dl}_{C'}(A)$ est ajouté à $Att_{Des}(A, I)$

2) L'ensemble Max(A, I) est mis à jour en traitant les éventuelles désactivations que permet le nouvel ensemble $Att_{Des}(A, I)$.

3) mise à jour de l'ensemble $Att_{Act}(A, I)$ par **activation** de l'ensemble $\mathfrak{Dl}_C(A)$ des déclencheurs dont la classe C définit l'accès :

   - Pour tout *d* de $\mathfrak{Dl}_C(A)$, traiter l'activation de *d* s'il est de source domaine ; sinon l'ajouter simplement à $Att_{Act}(A, I)$.

Lorsque l'activation intervient dans la phase de sélection ou provoque l'échec d'une inférence en cours d'exécution, App(A, I) est calculé à partir des nouveaux Max(A, I) et $Att_{App}(A, I)$.

2.3. L'événement Echec : l'application d'un déclencheur est en échec

Le déclencheur, dont l'application est en échec, est ajouté à $Att_{Des}(A, I)$. L'ensemble Max(A, I) est mis à jour par la procédure de désactivation et l'ensemble App(A, I) est recalculé.

Bibliographie

BIBLIOGRAPHIE

[Atta&85] G. Attardi, M. Simi, **Metalanguage and Reasoning Across Viewpoints**, Advances in artificial intelligence, Tim O'Shea (Ed.), pp. 413-422, North-Holland, Amsterdam, 1985.

[Atta&86] G. Attardi, M. Simi, **A Description-Oriented Logic for Building Knowledge Bases**, IEEE, Vol. 74, N° 10, pp. 1335-1344, October 1986.

[Bisso94] G. Bisson, **Définition de la notion de similarité dans les modèles à objets**, Langages et modèles à objets (LMO), pp. 93-96 , Grenoble, octobre 1994.

[Blak&87] E. Blake, S. Cook, **On including Part Hierarchies in Object-Oriented Languages, with an implementation in Smalltalk**, European Conference on Object-Oriented Programming, pp. 45-54, Paris, France, 1987.

[Bobr&77] D.G Bobrow, T. Winograd, **An Overview of KRL, a Knowledge Representation Language**, Cognitive Science, Vol. 1, N° 1, pp. 3-46, 1977.

[Borg91] A. Borgida, **Terminologic Frames as Types: Inference Rules and Prospective Applications**, Technical Report DCS-TR-280, Department of Computer Science, Rutgers University, New Jersey, May 1991.

[Borg92] A. Borgida, **From type systems to knowledge representation: natural semantics specifications for description logics**, International Journal of Intelligent and Cooperative Information Systems, Vol. 1, N°1, pp. 93-126, april 1992.

[Born81] A. Borning, **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.

[Brac83] R.J. Brachman, **What IS_A is and isn't: An analysis of Taxonomic Links in Semantic Networks**, IEEE Computer, Vol. 16, N° 10, pp.30-37, 1983.

[Brac85] R.J. Brachman, **"I lied about the Trees" or, Defaults and Definitions in Knowledge Representation**, The AI Magazine, Vol. 6, N° 3, pp. 80-93, fall 1985.

[Brac92] R.J. Brachman, **"Reducing" CLASSIC to Practice: Knowledge Representation Theory Meets Reality**, 3rd International Conference on Principles of Knowledge Representation and Reasoning KR'92, pp. 247-258, Cambridge, MA, october 25-29, 1992.

[Brac&83] R.J. Brachman, R.E. Fikes, H.J. Levesque, **KRYPTON: A Functional Approach to Knowledge Representation**, Special Issue on Knowledge Representation, IEEE Computer, Vol. 16, N° 10, pp. 67-73, october, 1983.

[Brac&85] R.J. Brachman, J.G. Schmolze, **An overview of the KL-ONE Knowledge Representation System**, Cognitive Science, Vol. 9, N° 2, pp. 171-216, 1985.

[Brac&91] R.J. Brachman, D.L. McGuinness, P.F. Patel-Schneider, L.A. Resnick, A. Borgida, **Living with CLASSIC: When and to use a KL-ONE-like Language**, J.F. Sowa (Ed.), Principles of Semantic Networks, pp. 401-456, Morgan Kaufman, 1991.

[Capp93] C. Capponi, **Classification des classes par les types**, 2ndes Journées Représentation Par Objets (RPO), pp. 215-224, La Grande Motte, 17-18 juin 1993.

[Capp94] C. Capponi, **Exploitation des types dans un modele de representation des connaissances par objets**, 9ième RFIA, Paris, 11-14 janvier 1994.

[Carr89] B. Carré, **Méthodologie orientée objet pour la représentation des connaissances**, Thèse, Laboratoire Informatique de Lille, 1989.

[Carr&87] B. Carré, G. Comyn, **Instanciation et représentation**, Rapport ERA N° 94, LIFL, Lille, 1987.

[Carr&88] B. Carré, G. Comyn, **On multiple classification, points of view and object evolution**, Artificial Intelligence and Cognitive Sciences, J. Demongeot, T. Hervé, V. Rialle, C. Roche (Eds.), Manchester University Press, chap. 4, pp. 49-62, 1988.

[Carr&90] B. Carré, L. Dekker, J-M. Geib, **Multiple and Evolutive Representation in the ROME Language, Towards an Integrated Corporate Information System**, Rapport ERA N° 80, LIFL, Lille, 1990.

[Carr&91] B. Carré, L. Dekker, **Inheriting Object-Oriented Features through Meta-Programming, A Frame Extension to ROME**, Rapport ERA N° 93, LIFL, Lille, 1991.

[Chail&86] J. Chailloux, M. Devin, F. Dupont, J-M. Hullot, B. Serpette, J. Vuillemin, **LE-LISP version 15.2: the Reference manual**, INRIA, Publication, mai 1986.

[Chan86] B. Chandrasekaran, **Generic Tasks in knowledge-Based Reasoning : High-Level Building Blocks for Expert System Design**, IEEE Expert, pp. 23-30, Fall 86.

[Clan83] W.J. Clancey, **The Epistemology of a Rule-Based Expert System — a Framework for Explanation**, Artificial Intelligence, N° 20, pp. 215-251, 1983.

[Clan85] W.J. Clancey, **Heuristic Classification**, Artificial Intelligence, Vol. 27, N° 4, pp. 289-350, 1985.

[Clay86] B. Clayton, **ART Programming Tutorial 2: a First Look at Viewpoints**, inférence, Los Angeles, 1986.

[Colm&83] A. Colmerauer, H. Kanoui, M. Van Caneghem, **PROLOG, bases théoriques et développements actuels**, TSI, Vol. 2, N° 4, pp. 255-292, 1983.

[Cord86] M-O. Cordier, **Connaissances incomplètes: raisonnement hypothétique et raisonnement par défaut**, 1ères Journées Nationales du PRC-GRECO Intelligence Artificielle, pp. 171-186, Aix-les-bains, 1986.

[Cruy&92] F. Cruypenninck, D. Ziébelin, **Classification d'objets complexes dans les bases de connaissances**, Représentation Par Objets (RPO), EC2, pp. 59-72 , La Grande-Motte, France, Juin 1992.

[Davi80] R. Davis, **Applications of meta-level knowledge to the construction, maintenance, and use of large knowledge bases**, Knowledge-Based Systems in Artificial Intelligence, McGraw-Hill, R. Davis & D. Lenat (Eds.), New York, 1980.

[Dekk94] L. Dekker, **FROME: Représentation multiple et classification d'objets avec points de vue**, Thèse de l'université des sciences et technologies de Lille, 1994.

[Dekk&92] L. Dekker, B. Carré, **Multiple an Dynamic Representation of Frames with Points of Viewin FROME**, Représentation Par Objets (RPO), EC2, pp. 97-111, La Grande-Motte, France, Juin 1992.

[DeKl86] J. De Kleer, **An Assumption-based TMS**, Artificial Intelligence, Vol. 28, N° 1, pp. 127-162, 1986.

[Deva&91] P.T. Devanbu, D.J. Litman, **Plan-Based Terminological Reasoning**, Second international Conference of Principles of Knowledge Representation and Reasoning (KR'91), pp 128-138, Cambridge, april 22-25, 1991.

[Doyl79] J. Doyle, **A Truth Maintenance System**, Artificial Intelligence, Vol. 12, N° 3, pp. 231-272, 1979.

[Doyl&91] J. Doyle, R.S. Patil, **Two theses on knowledge representation: language restriction, taxonomic classification, and the utility of representation services**, Artificial Intelligence, Vol. 48, N° 3, pp. 261-297, 1991.

[Duco90] R. Ducournau, **YAFOOL, Langage à objets, Version 3.3**, Manuel de référence, Y3, SEMA GROUP, Avril 1990.

[Duco93] R. Ducournau, **Héritages et Représentations**, Rapport d'habilitation à diriger des recherches, Université de Montpellier II, Mai 1993.

[Duco95] R. Ducournau, **Les systèmes classificatoires**, RR. LIRMM, N° 95-038, Montpellier, 1995.

[Duco&86] R. Ducournau, J. Quinqueton, **YAFOOL: encore un langage objet à base de frames!**, RT. INRIA, N° 72, Le Chesnay, août 1986.

[Duco&89] R. Ducournau, M. Habib, **La multiplicité de l'héritage multiple**, TSI, Vol. 8, N° 1, pp. 41-62, 1989.

[Duco&95] R. Ducournau, M. Habib, M. Huchard, M-L. Mugnier, A. Napoli, **Le point sur l'héritage multiple**, TSI, Vol. 14, N° 3, pp. 309-345, 1995.

[Esca93] J. Escamilla, **Shood: un modèle méta-circulaire de représentation des connaissances**, Thèse de l'institut national polytechnique de Grenoble, 1993.

[Esca&90] J. Escamilla, P. Jean, **Relationships in an object knowledge representation model**, 2nd conference on Tools for Artificial Intelligence, IEEE, pp. 632-638, Washington D.C., 1990.

[Euze87] J. Euzenat, **Un système de maintenance de la vérité pour une représentation de connaissance centrée-objet**, Mémoire de DEA, INPG-UJF, Grenoble , juin 1987.

[Euze88] J. Euzenat, **Un nouvel algorithme de maintenance de la vérité**, Rapport Technique, Cognitech, Paris, 1988.

[Euze90] J. Euzenat, **Un système de maintenance de la vérité à propagation de contextes**, Thèse de l'Université Joseph Fourier, 1990.

[Euze93a] J. Euzenat, **Définition abstraite de la classification et son application aux taxonomies d'objets**, Représentation Par Objets (RPO), EC2, pp. 235-246, La Grande-Motte, Juin 1993.

[Euze93b] J. Euzenat, **On a purely taxonomic and descriptive meaning for classes**, Proceeding of the IJCAI'93 Workshop on Object-Based Representation Systems (technical report CRIN 93-R-156, Nancy, 1993), pp. 81-92, Chambéry, 1993.

[Ferb86] J. Ferber, **Systèmes Experts et Approches Orientées Objets**, AVIGNON 86, pp. 525-542, Avignon, 28-30 avril 1986.

[Fike&71] R.E. Fikes, N.J. Nilsson, **STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving**, Artificial Intelligence, Vol. 2, pp. 189-208, 1971.

[Film88] R.E. Filman, **Reasoning with Worlds and Truth Maintenance in a Knowledge-based Programming Environment**, Communication of the ACM, Vol. 31, N° 4, pp. 382-401, April 1988.

[Fisc90] G. Fischer, **communication requirements for cooperative problem solving systems**, Information Systems, Vol. 15, N° 1, pp 21-36, 1990.

[Forg82] C.L. Forgy, **Rete: A Fast Algorithm for the many Pattern/ Many Object Pattern Match Problem**, Artificial Intelligence, N° 19, pp. 17-37, September 1982.

[Forn&89] M. Fornarino, A-M. Pinna, B. Trousse, **Approche orientée objet pour la mise en œuvre des relations dans un langage de schémas**, Reconnaissance des Formes et Intelligence Artificielle (RFIA), pp. 885-894, Paris, 1989.

[Forn&90] M. Fornarino, A-M. Pinna, **Expression des relations et maintien de la cohérence: le concept de lien**, RR. INRIA, N° 1346, Sophia-Antipolis, décembre 1990.

[Fox&86] M.S. Fox, J.M. Wright, D. Adam, **Experiences with SRL: An Analysis of a Frame-based Knowledge Representation**, Expert Database Systems, pp. 161-172, 1986.

[Gabr&91] R.P. Gabriel & al, **CLOS: integrating object-oriented and functional programming**, communication of ACM, Vol. 34, N° 9, september 1991.

[Ghal88] M. Ghallab, **Compilation de bases de connaissances**, Actes des journées nationales "Intelligence Artificielle", PRC-GRECO, Teknea, pp. 231-253, Toulouse, 14-15 mars 1988.

[Gens90] J. Gensel, **Gestion des dépendances et des hypothèses dans un modèle de connaissances à objets**, Mémoire de DEA, UJF-INPG, Grenoble, juin 1990.

[Gens93] J. Gensel, **Expression et satisfaction de contraintes dans TROPES, un modèle de représentation de connaissances par objets**, Représentation Par Objets (RPO), EC2, pp. 51-62, La Grande-Motte, France, Juin 1993.

[Gens95] J. Gensel, **Contraintes et représentation des connaissances par objets: application au modèle TROPES**, Thèse de l'université Joseph Fourier, Grenoble, 1995.

[Gens&92] J. Gensel, P. Girard, **Expression d'un modèle de tâches à l'aide d'une représentation par objets**, Représentation Par Objets (RPO), EC2, pp. 225-236, La Grande-Motte, France, Juin 1992.

[Gens&93] J. Gensel, P. Girard, O. Schmeltzer, **Integrating Constraints, Composite Objects and Tasks in a Knowledge Representation System**, 5th IEEE International Conference on Tools with Artificial Intelligence, pp. 127-130, Boston, november 8-11, 1993.

[Gens&94] J. Gensel, P. Girard, O. Schmeltzer, **Intégration de contraintes, d'objets composites et de tâches dans un modèle de représentation de connaissances par objets**, Congrès Reconnaissance des Formes et Intelligence Artificielle (RFIA), pp. 281-292, Paris, 11-14 janvier 1994.

[Gero&88] J.S Gero, M.L. Maher, W. Zhang, **Chunking Structural Design Knowledge as Prototypes**, RR. EDRC-12-25-88, Carnegie-Mellon University, Pittsburgh, 1988.

[Gold82] I.P. Goldstein, **The genetic graph : a representation for the evolution of procedural knowledge**, D.H. Sleeman & J.S. Brown (Eds.), Intelligent Tutoring Systems, Academic Press, London, 1982.

[Gold&83] A. Goldberg, D. Robson, **SMALLTALK-80, The language and its implementation**, Addison Wesley, 1983.

[Grei&80] R. Greiner, D.B. Lenat, **A Representation Language Language**, AAAI-80, pp. 165-169, Stanford, 1980.

[Griv&92] S. Grivaud, F. Rechenmann, **Navigation dans les bases de connaissances associant objets et hypertexte**, Représentations Par Objets (RPO), pp. 262-280, La Grande Motte, 22-23 juin 1992.

[Hart&91] R.T. Hartley, M.J. Coombs, **Reasoning with graph operations**, Principles of Semantic Network, Explorations in the representation of knowledge, J.F. Sowa (Ed.), Morgan Kaufmann (Publ.), Chapter 16, pp. 401-456, 1991.

[Hasl&84] D.W. Hasling, W.J. Clancey, G. Rennels, **Strategic explanations for a diagnostic consultation system**, International Journal of Man-Machine Studies, Vol. 20, pp 3-19, 1984.

[Hato&91] J.P. Haton, N. Bouzid, F. Charpillet, M-C Haton, H. Lâasri, P. Marquis, T. Mondot, A. Napoli, **Le raisonnement en intelligence artificielle**, InterEditions, Paris, 1991.

[Hend79] G.G. Hendrix, **Encoding Knowledge in Partitioned Networks**, Associative Networks: Representation and Use of Knowledge, N.V. Findler (Ed.), Academic Press, pp. 51-92, New York, 1979.

[Horn91] W. Horn, **The challenge of deep models, inference structures and abstract tasks**, Applied Artificial Intelligence, Vol. 5, pp 87-96, 1991.

[ILOG92] ILOG S.A., **PECOS Version 1.1, User's Reference Manual**, Gentilly, France, 1992.

[Inte86] IntelliCorp, **Knowledge Engineering Environment (KEE) System**, IntelliCorp, Technical Report, Update, august, 1986.

[Kacz&86] T.S. Kaczmarek, R. Bates, G. Robins, **Recent developments in NIKL**, AAAI 86, pp. 978-987, Philadelphia, 1986.

[Kaut&86] H.A. Kautz, J.F. Allen, **Generalized Plan Recognition**, Fifth National Conference on Artificial Intelligence, Philadelphia, August 1986.

[Laur87] J.L. Laurière, **Intelligence artificielle: résolution de problèmes par l'homme et la machine**, Eyrolles, Paris, 1987.

[Lema93] F. Lemaire, **Étude des protocoles d'interaction dans un environnement d'aide à la recherche collaborative**, Mémoire de DEA science cognitive, INPG, Grenoble, juillet 1993.

[Lin&91] D. Lin, R. Goebel, **A Message Passing Algorithm for Plan Recognition**, Twelveth International Joint Conference on Artificial Conference IJCAI'91, pp. 280-285, Sydney, 1991.

[Liot93] M-P. Liotard, **Mécanisme de classification pour un système de représentation de connaissances**, Mémoire d'ingénieur CNAM, Grenoble, 1993.

[MacG&91] R.M. MacGregor, M.H. Burstein, **Using a Description Classifier to Enhance Knowledge Representation**, IEEE Expert Intelligent Systems and Applications, juin 1991.

[MacG&92] R.B. MacGregor, D. Brill, **Recognition Algorithms for the Loom Classifier**, AAAI, pp. 774-779, San Jose, California, July 1992.

[Maes87] P. Maes, **Introspection in knowledge representation**, Advances in Artificial Intelligence - II, B. Du Boulay, D. Hogg and L. Steels (Eds.), Elsevier Science Publishers B.V. (North-Holland), pp. 249-262, 1987.

[Magn94] M. Magnan, **Réutilisation des composants: les exceptions dans les objets composites**, Thèse de l'Université de Montpellier II, septembre 1994.

[Marc86] K. Van Marcke, **A parallel Algorithm for Consistency Maintenance in Knowledge Representation**, 7th ECAI, Vol. 1, pp. 278-290, Brigthon, 1986.

[Mari89] O. Mariño, **Classification dans un modèle multi-points de vue**, Mémoire de DEA, UJF-INPG, Grenoble, septembre 1989.

[Mari91] O. Mariño, **Classification d'objets composites dans un système de représentation de connaissances multi-points de vue**, Congrès Reconnaissance des Formes et Intelligence Artificielle (RFIA), pp. 233-242, Lyon-Villeurbanne, 1991.

[Mari93] O. Mariño, **Raisonnement classificatoire dans une représentation à objets multi-points de vue**, Thèse de l'Université Joseph Fourier, Grenoble, 1993.

[Mari&90] O. Mariño, F. Rechenmann, P. Uvietta, **Multiple Perspectives and Classification Mechanisms in Object-Oriented Representation**, L.C. Aiello editor, Ninth European Conference on Artificial on Artificial Intelligence ECAI'90, pp. 425-430, Stockholm, Sweden, August 1990.

[Masi&89] G. Masini, A. Napoli, D. Colnet, D. Léonard, K. Tombre, **Les langages à objets**, InterEditions, Paris, 1989.

[Meye90] B. Meyer, **Conception et programmation par objets, pour du logiciel de qualité**, InterEditions, Paris, 1990.

[Mins75] M. Minsky, **A Framework for Representing Knowledge**, The Psychology of Computer Vision, P. Winston (Ed.), McGraw-Hill, pp. 211-281, New York, 1975.

[Napo92] A. Napoli, **Représentations à objets et raisonnement par classification en intelligence artificielle**, Thèse de Doctorat d'Etat, Université de Nancy 1, Janvier 1992.

[Napo&92] A. Napoli, R. Ducournau, **Subsomption in Objected-Based Representations**, ERCIM Workshop on Theoretical and Experimental Aspects of Knowledge Representation, pp. 1-9, Pisa, May 21-22, 1992.

[Naul&91] J.P. Nault, O. Schmeltzer, **La reconnaissance d'intentions par appariement d'objets**, Rapport de projet (Année spéciale Intelligence Artificielle), Grenoble, 1991.

[Nebe88] B. Nebel, **Computational Complexity of Terminological Reasoning in BACK**, Artificial Intelligence, Vol. 34, N° 3, pp. 371-383, 1988.

[Newe81] A. Newell, **The Knowledge Level**, Artificial Intelligence, Vol. 2, N°2, pp. 1-20, 1981.

[Newe&63a] A. Newell, J.C. Shaw, H.A. Simon, **Empirical Explorations with the Logic Theory Machine: A case study in Heuristics**, Computers and thought, E.A. Feigenbaum & J. Feldman (Eds.), McGraw-Hill, New York, 1963.

[Newe&63b] A. Newell, H.A. Simon, **GPS: A Program That Stimulates Human Thought**, Computers and thought, E.A. Feigenbaum & J. Feldman (Eds.), McGraw-Hill, New York, 1963.

[Nguy&92] G.T. Nguyen, D. Rieu, J. Escamilla, **An object model for engineering design**, RR. INRIA, N° 1653, Rocquencourt, Avril 1992.

[Orsi90] B. Orsier, **Evolution de l'attachement procédural: intégration de tâches, méthodes et procédures dans une représentation de connaissances par objets**, Mémoire de DEA, UJF-INPG, Grenoble, juin 1990.

[Pate&90] P.F. Patel-Schneider, et al., **Term subsumption Language in Knowledge Representation**, AI Magazine, Vol. 11, N° 2, pp. 16-23, summer 1990.

[Pier91] C. Pierret-Goldbreich, **Vers une nouvelle architecture pour l'explication du raisonnement au niveau connaissances : TASK**, Congrès Reconnaissance des Formes et Intelligence Artificielle (RFIA), pp 515-524, Lyon, 1991.

[Pier&91] C. Pierret-Goldbreich, I. Delouis, **Task centered representation for expert systems at the knowledge level**, 8th Conference of the society for the study of the Artificial Intelligence and simulation of behavior, University of Leeds, april 1991.

[Pitr90] J. Pitrat, **Métaconnaissances**, Hermès, Paris, 1990.

[Ponc91] T. Poncabaré, **SCAI : un environnement de développement de systèmes à base de connaissances en calcul scientifique et technique**, Mémoire d'ingénieur CNAM, Grenoble, 1991.

[Ponc&91] T. Poncabaré, F. Rechenmann, **SCAI : un environnement de développement de systèmes à base de connaissances en calcul scientifique et technique**, Convention IA, Hermes, pp 491-509, Paris, 1991.

[Puge92] J.F. Puget, **Programmation par contraintes orientée objet**, AVIGNON 92, pp. 129-138, Avignon, 1992.

[Quil68] M.R. Quillian, **Semantic Memory**, Semantic Information Processing, M. Minsky (Ed.), MIT Press Cambridge, pp. 227-270, MA, 1968.

[Quin93] J.A. Quintero Garcia, **Parallélisation de la classification d'objets dans un modèle de connaissances multi-points de vue**, Thèse, Université Joseph Fourier, Grenoble, 1993.

[Rech85] F. Rechenmann, **SHIRKA: mécanismes d'inférence sur une base de connaissances centrée-objet**, Congrès Reconnaissance des Formes et Intelligence Artificielle (RFIA), pp. 1243-1254, Grenoble, 1985.

[Rech92] F. Rechenmann, **Method inheritance and selection in class-based knowledge models**, ERCIM Workshop on Theoretical and Experimental Aspects of Knowledge Representation, pp. 205-208, Pise, Italie, 21-22 mai 1992.

[Rech93] F. Rechenmann, **Building and sharing large knowledge bases in molecular genetics**, Workshop KB & KS'93 (International Conference on Building and Sharing of Very Large-Scale Knowledge Bases), Tokyo, Japon, 1-4 december 1993.

[Rech&89] F. Rechenmann, P. Uvietta, P. Fontanille, **SHIRKA, manuel d'utilisation**, Rapport interne, IMAG, Grenoble (FR), 1989.

[Rech&91] F. Rechenmann, P. Uvietta, **SHIRKA - An Object-Centered Knowledge Base Management System**, Artificial Intelligence in Numerical and Symbolic Simulation, ALEAS, Lyon, France, 1991.

[Reit78] R. Reiter, **On Closed-World Data Base**, Logic and Data Bases, Gallaire et Minker (Eds.), Plenum Press, pp. 55-76, New-York, 1978.

[Rich87] E. Rich, **Intelligence Artificielle**, Masson, Paris, 1987.

[Rieu&92] D. Rieu, G.T. Nguyen, J. Escamilla, **Méthodes et représentation de connaissances**, Représentation Par Objets (RPO), EC2, pp. 17-29, La Grande-Motte, France, Juin 1992.

[Robe&77] R.B. Roberts, I.P. Goldstein, **The FRL Manual**, AI Memo 409, Artificial Intelligence Laboratory, MIT, september 1977.

[Robs&81] D. Robson, A. Goldberg, **The Smalltalk-80 System**, Byte 6-VIII, pp. 36-48, august 1981.

[Rous88] B. Rousseau, **Vers un environnement de résolution de problèmes en biométrie: apport des techniques de l'intelligence artificielle et de l'interaction graphique**, Thèse, Université Claude Bernard Lyon 1, 1988.

[Scha&77] R.C. Schank, R.P. Abelson, **Scripts, Plans, Goals and Understanding**, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1977.

[Shor76] E. Shortliffe, **Computer-Based Medical Consultation: MYCIN**, Elsevier, New York, 1976.

[Smit&84] R.G. Smith, G.E. Lafue, E. Schoen, S.C. Vestal, **Declarative Task Description as a User-Interface Structuring Mechanism**, IEEE Computer, September 1984.

[Somb88] L. Sombé, **Inférences non-classiques en Intelligence Artificielle - Ebauche de comparaisons sur un exemple**, Actes des journées nationales "Intelligence Artificielle", PRC-GRECO, Teknea, pp. 137-229, Toulouse, 14-15 mars 1988.

[Stee87] L. Steels, **The Deepening of Expert Systems**, AI Communications, N°1, pp. 9-17, 1987.

[Stee90] L. Steels, **components of expertise**, AI Magazine, Summer 1990.

[Stef&85] M.Stefik, D. Bobrow, **Object-Oriented Programming: Themes and Variations**, The AI Magazine, Vol. 6, N° 4, pp. 40-62, 1986.

[Stro91] B. Stroustrup, **The C++ Programming Language**, Second Edition, Addison-Wesley, 1991.

[Suss&80] G.J. Sussman, G.L. Steele Jr., **CONSTRAINTS—A Language for Expressing Almost-Hierarchical Descriptions**, Artificial Intelligence, Vol. 14, pp. 1-39, 1980.

[Swar83] W.R. Swartout, **XPLAIN: a system for creating and explaining expert consulting systems**, Artificial Intelligence, Vol. 3, N° 21, pp. 285-325, September 1983.

[Taya93] N. Tayar, **A model for developing large shared knowledge bases**, 2nd International Conference on Information and Knowledge Management, pp. 717-719, Washington, 1-5 novembre 1993.

[Wein80] J.L. Weiner, **BLAH, a system which explains its reasoning**, Artificial Intelligence, Vol. 15, N° 1-2, pp. 19-48, 1980.

[Will84] C. Williams, **ART: The Advanced Reasoning Tool, Conceptual Overview**, Inference Corp., Los Angeles, 1984.

[Wino75] T. Winograd, **Frame Representation and the Declarative/Procedural Controversy**, Representation and Understanding: Studies in Cognitive Science, D.G. Bobrow and A.M. Collins (Eds.), Academic Press, pp. 185-210, 1975.

[Wood75] W.A. Woods, **What's in a Link: Foundations for Semantic Networks**, Representation and Understanding: Studies in Cognitive Science, D.G. Bobrow and A.M. Collins (Eds.), Academic Press, pp. 35-82, New York, 1975.

[Wood91] W.A. Woods, **Understanding Subsumption and Taxonomy: A Framework for Progress**, Principles of Semantics Networks, Explorations in the representation of knowledge, J.F. Sowa (Ed.), Morgan Kaufmann (Publ.), pp. 45-94, 1991.

[Yen&91] J. Yen, R. Neches, R. MacGregor, **CLASP: Integrating Term Subsumption Systems and Production Systems**, IEEE Transactions on Knowledge and Data Engineering, Vol. 3, N° 1, pp 25-31, March 1991.

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

- **Doctorat de L'Université Joseph Fourier - Grenoble 1**
- **Doctorat d'Etat**
- **Diplôme Supérieur de Recherches**

(Rayer les mentions inutiles)

Nom GIRARD Prénom Pierre N° Etudiant 1|5|0|1|1|6|4|1

Titre de la Thèse Construction hypothétique d'objets complexes

Président du Jury : Jean-François BRANDET, Professeur Université Joseph Fourier (Grenoble I)

Membres du Jury :

M. Carré Bernard, Maître de Conférence, Université Lille I

M. Ducourneau Roland, Professeur, Université de Montpellier II

M. NGUYEN Gia Toan, Directeur de Recherche, INRIA, Grenoble

M. Rechenmann François, Directeur de Recherche, INRIA, Grenoble

Date de la soutenance : 26 octobre 1995

Reproduction de la thèse soutenue :

a - [X] Thèse pouvant être reproduite en l'état

b - [ ] Thèse ne pouvant être reproduite

c - [ ] Thèse pouvant être reproduite APRES CORRECTIONS SUGGEREES au cours de la soutenance.

Signature du Président du Jury

J.F. Brandet

Cas C : Après la soutenance, **DANS UN DELAI DE 3 MOIS,** le Directeur de thèse veille à l'exécution des corrections demandées au candidat par le jury.

UNIVERSITE
JOSEPH FOURIER
SCIENCES. TECHNOLOGIE. MEDECINE

ATTESTATION

Le Président de l'Université Joseph FOURIER - Grenoble I -, soussigné certifie que

Monsieur GIRARD Pierre

né(e) le 7 mai 1965 à ROUEN (76)

inscrit(e) à l'Université Joseph Fourier - Grenoble I -, sous le numéro 8501164

a soutenu le **26 octobre 1995** conformément aux règlements, la thèse:

Construction hypothétique d'objets complexes

devant le jury composé de :

Président

Madame BRUANDET Marie-France, Professeur

Membres

M. CARRE B., Maître de Conférences
M. DUCOURNAU R., Professeur
M. NGUYEN G.T., Directeur de Recherches
M. RECHENMANN F., Directeur de Recherches

Le jury a accordé à l'intéressé(e) le grade de Docteur de l'Université Joseph Fourier - Grenoble I - spécialité **INFORMATIQUE**

avec la mention **TRES HONORABLE**

pour en jouir avec les droits et prérogatives qui y sont attachés par les lois, décrets, et règlements.

Délivrée à Grenoble, le 18 Décembre 1995

Le Président de l'Université

Daniel BLOCH

AVIS TRES IMPORTANT
- L'intéressé(e) ne devra en aucun cas se dessaisir de la présente attestation car il ne lui en sera pas délivré un second exemplaire.Pour justifier de ses capacités, l'impétrant doit faire des copies de cette attestation, sur papier libre, et les faire certifier conformes à l'original par le Maire ou le Commissaire de Police.

UJF - GRENOBLE 1 - B.P. 53 - 38041 GRENOBLE CEDEX 9 - FRANCE - TELEPHONE : (33) 76 51 46 00 - TELEX : 980 134 F - TELECOPIE : (33) 76 51 48 48

RESUME

Dans les modèles à objets distinguant la notion de classe (ensemble d'individus) de celle d'instance (individu particulier), la classe peut jouer deux rôles fondamentalement différents. Les modèles qui mettent en œuvre des mécanismes de classification d'instances, présentent la classe comme une unité d'identification caractérisant les propriétés que doivent posséder toutes ses instances. Ces propriétés sont alors utilisées pour établir l'appartenance d'une instance à une classe. D'autres modèles utilisent la classe comme unité de construction, ou de gestion, d'instances. Ils laissent alors toutes libertés dans la description de la classe pour introduire des informations procédurales qui sont utilisées pour modifier ou compléter une instance qui lui a été explicitement rattachée.

Notre travail consiste à mettre en place un mécanisme capable de trouver les différentes solutions que peut proposer une hiérarchie de classes à un problème de construction d'instance. Schématiquement, un tel problème se caractérise par la donnée d'une instance pour laquelle subsistent des possibilités de raffinement dans la hiérarchie de classes et dont la valeur de certains attributs reste inconnue.

La solution proposée prend place dans un modèle à objets, appelé TROPES, défini pour accueillir un raisonnement classificatoire. La mise en place du mécanisme de construction d'instance par exploration d'une hiérarchie est assurée par un système, dit d'assistance hypothétique, couplé au système TROPES. Ce système d'assistance est chargé de produire et de gérer les hypothèses permettant de prolonger l'exploration d'une hiérarchie de classes au delà des limites atteintes par la classification d'instances. Les différentes combinaisons d'hypothèses pouvant ainsi être formées sont validées par TROPES sur des versions différentes de l'instance. Lorsque le mécanisme est appliqué à un objet composite, le partage de valeurs d'attributs entre composants et composite permet la propagation du processus de construction du composite vers les composants.

MOTS-CLES : système à base de connaissances, représentation par objets, raisonnement classificatoire, raisonnement hypothétique, objet composite, version d'objet.

ABSTRACT

Among object-based knowledge models distinguishing the class notion (as a set of individuals) from the instance notion (as a specific individual), a class may play two different roles. For models that support instance classification mechanism, a class is an identification unit that describes the properties its instances must satisfy. Therefore, a class description is used for establishing whether an instance can belong to the class or not. Other models make use of class as an instance construction unit. In these models, the class description is more permissive since procedural information may be introduced and used for modifying or filling in an instance being explicitly attached to the class.

The purpose of our work is to define a mechanism able to find the different solutions a class hierarchy can propose for an instance construction problem. Such a problem is characterized by an instance for which some refinements into the class hierarchy are still possible and some attribute values are still unknown.

Our proposal takes place into an object-based knowledge model, namely TROPES, which supports a classification-based reasoning. The instance construction mechanism, based on class hierarchy exploration, is controlled by a system, namely *hypothetical assistance system*, coupled with the TROPES system. This assistance system manages all the hypotheses made on the instance being constructed in order to pursue the hierarchy class exploration out of the limits reached by the instance classification. The different possibilities of hypothesis combinations are validated by TROPES on different instance versions. When the mechanism is applied to a composite object, shared properties between components and composite are used to propagate the construction process from composite to components.

KEYWORDS : knowledge base system, object representation, classification-based reasoning, hypothetical reasoning, composite object, object version.

ISBN - 2 - 7261 - 0960 - 8

TU_370