INRIA
Projet SECOIA
INRIA Sophia-Antipolis
Route des Lucioles
06565 Valbonne Cedex
FRANCE

Etude du raisonnement par cas :
recherche des cas similaires et utilisation des ensembles flous

Michel JACZYNSKI

Rapport de stage
Stage ESSI 3ème année - DEA informatique
Université de Nice Sophia-Antipolis

3 septembre 1993

Responsable

Mlle. Brigitte TROUSSE

Rapporteur

Mme. Annie CAVARERO

Remerciements

Je remercie Bertrand NEVEU pour m'avoir accueilli dans son équipe.

Je remercie particulièrement Brigitte TROUSSE pour m'avoir guidé dans la réalisation de mon étude et dans la rédaction de mon rapport ainsi que pour m'avoir fait découvrir le monde passionnant de la recherche.

Je remercie toute l'équipe SECOIA et notamment Wided Lejouad pour m'avoir supporté pendant cinq mois.

Je tiens également à remercier ma famille et mes amis qui m'ont soutenu et beaucoup aidé dans la mise en forme de ce rapport.

Table des matières

Chapitre 1

Introduction

Ce stage se situe dans le cadre des recherches menées au sein du projet SECOIA[1] de l'INRIA[2] Sophia-Antipolis sur l'assistance logicielle à la réutilisation d'anciennes conceptions.

En effet, la réutilisation d'anciennes solutions fait partie intégrante de la démarche de conception qui semble suivre les étapes suivantes[Tro92] :

1. réutilisation directe de l'existant,
2. modification d'une conception existante,
3. construction de l'artefact par assemblage,
4. conception à partir de zéro.

De plus, de nombreuses études psychologiques ont montré l'intérêt d'assister les concepteurs dans cette réutilisation. En particulier, la recherche de solutions existantes intéressantes pour résoudre un problème est la phase la plus difficile à réaliser même pour un expert; c'est pourquoi cette recherche peut être considérée comme la fonction la plus importante que puisse offrir un outil d'aide à la conception [VT93].

Dans ce but et dans le domaine de l'intelligence artificielle, nous sommes amenés à étudier les apports possibles d'un système de raisonnement par cas. Effectivement, le raisonnement par cas ou CBR[3] est une méthode de résolution de problèmes basée sur l'utilisation d'expériences passées appelées *cas*. Un système de CBR gère une mémoire pour stocker ces différents cas, et pour résoudre un nouveau problème il commence par rechercher dans cette mémoire les cas les plus similaires au problème. Puis, le système adapte l'ancien cas à la nouvelle situation, de manière à déduire une solution pour le problème. Enfin, la nouvelle expérience (nouveau cas) est alors intégrée dans la mémoire de cas et pourra être utilisée dans des raisonnements futurs.

L'objectif général visé de ces recherches concerne un générateur de systèmes de CBR d'aide à la conception. Plus particulièrement, le sujet de notre stage est de concevoir un système permettant de réaliser la phase de recherche des cas similaires. Ce système doit être le plus stable possible de manière à permettre l'évolution de ses différents composants.

[1] Systèmes Experts et Conception d'Outils pour l'Intelligence Artificielle
[2] Institut National de Recherche en Informatique et en Automatique
[3] CBR : Case-Based Reasoning.

L'organisation de ce rapport est la suivante : dans le chapitre 2, nous faisons une rapide présentation du raisonnement par cas. Dans le chapitre 3, nous détaillons les différents mécanismes existants employés durant toutes les phases du raisonnement par cas. Dans le chapitre 4, nous analysons le sujet de notre étude et nous présentons les différents choix de conception d'un module de recherche des cas similaires. Dans le chapitre 5, nous détaillons la conception de notre système. Puis, dans le chapitre 6, nous présentons la réalisation et les techniques d'implantation du système. Enfin, dans le chapitre 7, nous concluons notre étude par une analyse critique du système et par ses perspectives.

Les annexes offrent des renseignements complémentaires pour mieux situer et comprendre notre travail. Ainsi, l'annexe A introduit la notion d'ensemble flou et présente les résultats de la théorie des possibilités. L'annexe B résume les différentes caractéristiques du générateur de systèmes experts en classification CLASSIC que nous avons étudié. L'annexe C présente les mécanismes de base de la couche objet du langage IlogTalk qui a été utilisé pour la réalisation. L' annexe D détaille l'interface fonctionnelle du système. L'annexe E donne un exemple de fonctionnement de la partie implantée de notre système.

Chapitre 2

Présentation du raisonnement par cas

Nous allons commencer par présenter les liens entre le raisonnement par cas en intelligence artificielle et le raisonnement analogique en psychologie. Puis nous détaillerons l'organisation des différentes étapes du raisonnement par cas ainsi que les avantages et les inconvénients de ce processus de résolution de problèmes. Nous finirons par l'introduction de quelques applications anciennes et d'autres plus récentes, et par la présentation de quelques exemples pour illustrer concrètement le raisonnement par cas.

2.1 Raisonnement par cas et raisonnement analogique

Le but du raisonnement par cas est de résoudre un problème courant grâce à l'utilisation de situations passées similaires. Faire référence à des situations passées pour résoudre un problème n'est pas original, nous utilisons tous ce raisonnement régulièrement. Par exemple, lorsqu'un patient présente des symptômes inhabituels, un docteur va essayer de se rappeler d'un cas similaire pour pouvoir s'inspirer de l'ancien diagnostic. De même, un architecte qui commence un nouveau plan, ne va pas retourner systématiquement aux principes de base et essayer toutes les possibilités. Par contre, il va chercher d'anciens plans pour en extraire des parties et il va essayer de les modifier pour satisfaire les différentes contraintes. La réutilisation d'anciennes solutions est essentielle dans l'architecture et dans tout travail de conception.

De manière générale, nous sommes plus compétents la seconde fois que la première, car nous nous rappelons de nos échecs et pouvons ainsi les éviter.

Les psychologues ont bien sûr étudié ce type de raisonnement et ils l'ont modélisé par le *raisonnement analogique*. Ces travaux ont permis de mieux spécifier les différentes étapes de ce raisonnement ainsi que la notion de similarité. En fait, le raisonnement par cas s'inspire du raisonnement analogique.

L'analogie se fonde sur la notion de similitude de termes dans un certain univers: *A est à B ce que C est à D* [Ha91a]. Plus concrètement, il existe deux approches qui permettent de modéliser le raisonnement analogique : l'approche pragmatique [Kea88] [HT89] et l'approche syntaxique [Gen89]. La première approche lie la résolution des problèmes aux buts à atteindre et au domaine considéré, alors que dans la seconde approche c'est une correspondance de structure entre le problème à résoudre et les anciennes expériences qui détermine l'analogie. Toutefois, le modèle de Wolstencroft essaye d'unifier ces deux approches pour proposer un modèle plus général [Wol89] .

S. Slade résume en cinq lignes directrices les bases psychologiques du CBR [Sla91]:

1. La mémoire contient l'expérience.
2. Les expériences sont liées entre elles par de multiples liens complexes et abstraits.
3. La mémoire est dynamique et change avec le temps.
4. L'expérience guide le raisonnement.
5. L'apprentissage se fait par l'échec.

2.2 Etapes du raisonnement par cas

Dans la figure 2.1, nous voyons les différentes étapes du raisonnement par cas. Les étapes sont

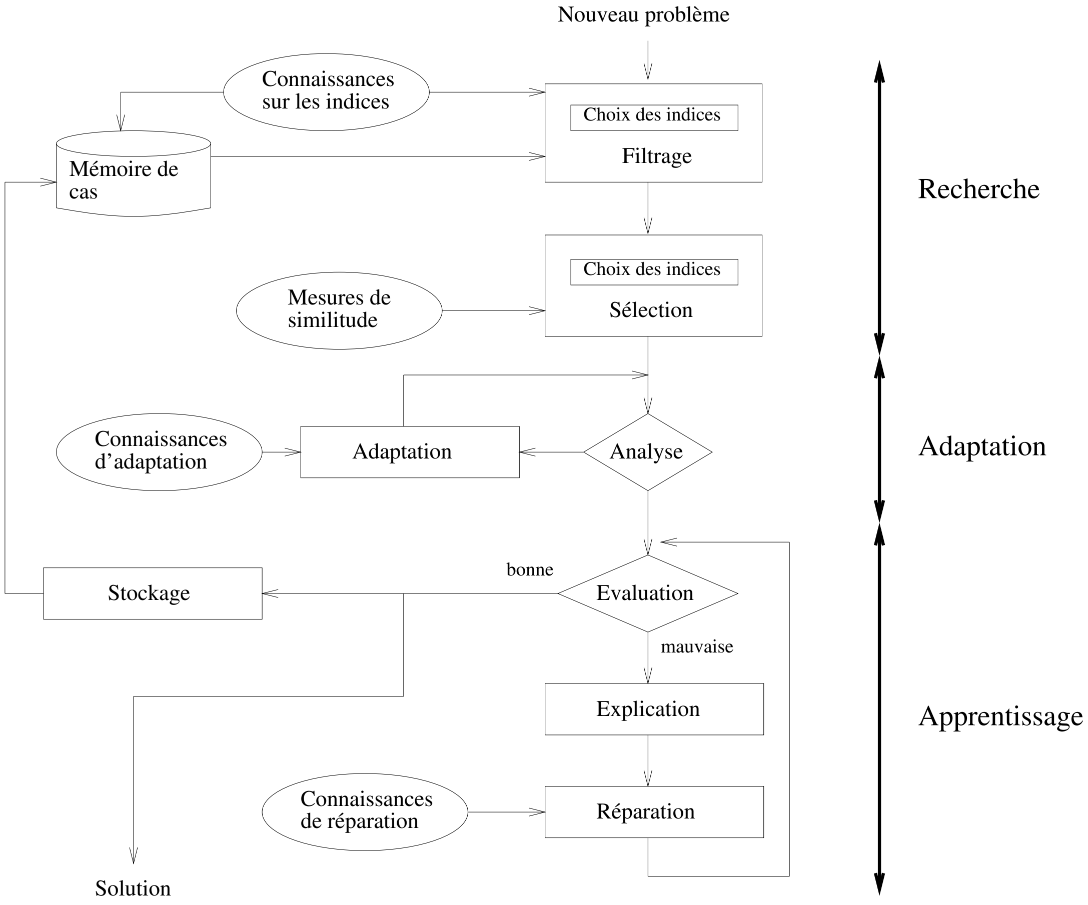

Figure 2.1 : Etapes du raisonnement par cas.

symbolisées par des rectangles. Les ellipses correspondent aux connaissances nécessaires pour ces étapes.

Ces connaissances peuvent provenir :

- de l'utilisateur grâce à un dialogue homme-machine,
- d'une description statique liée au domaine comme une base de règles,
- d'une évaluation dynamique liée au déroulement des anciens raisonnements,
- d'une combinaison des différentes méthodes.

Le raisonnement par cas se décompose en trois grandes phases qui elles-mêmes sont constituées de plusieurs étapes :

1. **Recherche des cas similaires**
   - Choix des index : il faut choisir des caractéristiques particulières du nouveau problème qui vont guider la recherche.
   - Filtrage : cette étape va rechercher et isoler le ou les cas les plus similaires potentiellement.
   - Sélection : le ou les cas les plus intéressants sont sélectionnés.
2. **Adaptation des cas**
   - Analyse : le ou les cas sont étudiés et des méthodes d'adaptation sont choisies.
   - Adaptation : le ou les cas sélectionnés sont alors transformés pour pouvoir donner une solution au problème courant.
3. **Apprentissage**
   - Evaluation, explication et réparation : la solution est évaluée et si elle ne résout pas le problème elle peut être réparée suivant les explications données pour finalement obtenir une solution correcte au problème. Ces étapes peuvent se répéter plusieurs fois.
   - Stockage : le nouveau cas constitué notamment du problème et de la solution est alors ajouté à la mémoire de cas pour servir dans les futurs raisonnements.

Dans la pratique toutes les étapes ne sont pas toujours réalisées; il existe plusieurs niveaux de CBR [KR89]. En effet, le système peut être uniquement intéressé dans la recherche des cas similaires, l'étape d'adaptation est alors éliminée et l'apprentissage se réduit au stockage de nouveaux cas. D'autres applications peuvent éliminer simplement l'étape de réparation.

2.3 Avantages et inconvénients du CBR

Voici les principaux avantages du CBR [RKW89] [Sla91]:

- Le CBR évite les erreurs passées par son mécanisme d'apprentissage par l'échec, contrairement au raisonnement par règles.

- L'enrichissement de la mémoire avec de nouveaux cas permet d'avoir des raisonnements qui s'améliorent et deviennent plus rapides.
- L'acquisition des connaissances est beaucoup plus simple que pour les systèmes experts car cette phase demande moins de formalisation.
- Les différentes réalisations ont montré que le temps de développement d'un système CBR est beaucoup plus court que pour un système expert sur le même sujet.
- Dans différents domaines, des bases de données de problèmes avec leurs solutions existent déjà, et le CBR permet de les réutiliser par simple intégration.
- L'explication du raisonnement est générée beaucoup plus facilement et de manière plus satisfaisante que dans les systèmes experts ou d'autres systèmes apprentis comme les réseaux de neurones. Les anciens cas qui sont utilisés servent tout simplement de justification.

Toutefois, cette technique présente quelques inconvénients :

- Le système peut ne pas proposer la solution optimale, puisque l'espace des solutions n'est pas entièrement connu.
- Un grand espace mémoire est nécessaire.

2.4 Quelques applications

Le raisonnement par cas peut être appliqué dans de multiples domaines. Toutefois selon J. Kolodner [Kol92] nous pouvons séparer deux buts distincts à l'intérieur du CBR. Nous avons tout d'abord les applications qui résolvent des problèmes (conception, planning, diagnostic), et d'un autre coté les applications qui veulent avoir des justifications, des interprétations ou des projections grâce à la connaissance d'anciens cas (justice, classification, stratégie militaire).

La figure 2.2 donne la liste de quelques applications déjà réalisées avec leur domaine respectif ainsi que leur description. Ces applications seront utilisées par la suite pour illustrer quelques techniques.

Il existe également des générateurs de systèmes CBR qui sont commercialisés. Les plus connus sont ReMind de *Cognitive Systems Inc.* [Sys92], disponible sur diverses plateformes (PC, MAC, UNIX), et CBR-Express de *Inference Corp* basé sur un outil de raisonnement par règles ART-IM. Ces deux générateurs ont permis de développer des applications notamment dans le domaine des *helpdesk* où il s'agit de faire du diagnostic de panne. D'autres applications dans l'industrie ont vu le jour [Mot93]: Digital Equipment Corp. a réalisé son propre système CBR pour diagnostiquer les pannes potentielles de réseaux, le Toronto Stock Exchange est en train de tester un système pour reconnaître des prix et des échanges anormaux.

Nom	Année	Domaine	Description
CYRUS [Kol83]	1980	Diplomatie	Suivi des déplacements d'un diplomate
IPP [RS89]	1980	Terrorisme	Compréhension de rapport d'actions terroristes
MEDIATOR [RS89]	1985	Médiation	Résolution des conflits politiques
CHEF [RS89]	1986	Cuisine	Conception de plats
JUDGE [RS89]	1986	Justice	Détermination d'une sentence pour un criminel
SWALE[RS89]	1986	Médecine	Explication de décès
HYPO [KR89]	1987	Justice	Recherche d'argumentation
CASEY [KR89]	1988	Médecine	Diagnostic des défaillances cardiaques
JULIA [KR89]	1988	Cuisine	Conception d'un menu
PERSUADER[RS89]	1988	Médiation	Résolution de conflits sociaux
PROTOS [KR89]	1988	Médecine	diagnostic des problèmes d'audition
FIRST[DHR89]	1990	Mécanique	Re-conception de poutres
DEJAVU[BZ92]	1991	Mécanique	Conception de voitures

Figure 2.2 : Liste de quelques applications.

2.5 Quelques exemples

Les différents exemples qui suivent sont tirés et adaptés de [KR89].

2.5.1 CHEF

Problème :	créer un plat avec du poulet et des petits pois.
Ancien cas :	bœuf et brocolis. hacher l'ail, couper le bœuf en morceaux, faire mariner le bœuf dans ..., couper les brocolis en morceaux, faire cuire les brocolis pendant 3 minutes, ...
Nouvelle solution :	hacher l'ail, couper le poulet en morceaux, faire mariner le poulet dans ..., faire cuire les pois pendant 2 minutes, ...

2.5.2 MEDIATOR

Problème :	deux sœurs veulent la même orange pour la manger.
Ancien cas :	deux enfants veulent manger une barre de sucre candi. l'enfant le plus âgé casse la barre en deux et laisse le plus jeune choisir un morceau.
Nouvelle solution :	la plus âgée des deux sœurs coupe l'orange en deux et laisse la plus jeune choisir un morceau.

2.5.3 SWALE

Problème :	Len Bias, un joueur de basket ball en bonne santé vient de mourir d'une attaque cardiaque. Pourquoi?
Ancien cas :	Jim Fixx, un coureur apparemment en bonne santé est mort d'une crise cardiaque après une course. Il avait des défaillances cardiaques inconnues auparavant.
Nouvelle solution :	Len bias avait des défaillances cardiaques inconnues auparavant.

Chapitre 3

Mécanismes du raisonnement par cas

Dans un premier temps, nous allons introduire le concept de cas et ses représentations, puis nous décrirons les différentes organisations de la mémoire qui permettent de stocker et de retrouver les cas. Ensuite, nous étudierons en détail chaque phase du raisonnement par cas : la recherche des cas similaires, l'adaptation des cas, et l'apprentissage.

3.1 Représentation des cas

Nous devons étudier la représentation des cas selon les deux niveaux de A. Newell [New82] : la représentation au niveau "connaissance" ("knowledge level") et la représentation pratique ("symbol level").

3.1.1 Représentation au niveau "connaissance"

Sur le plan sémantique, un cas est décrit par des caractéristiques qui peuvent représenter plusieurs types d'informations:

- la description du problème,
- la solution et/ou les étapes qui ont mené à la solution,
- le résultat (échec ou succès lors de l'évaluation),
- d'autres informations sur le déroulement du raisonnement (explication des échecs, stratégie d'adaptation,...),

Ces caractéristiques peuvent représenter le problème ou la solution à plusieurs niveaux d'abstraction. Par exemple, DEJAVU possède des caractéristiques dites de surface et des caractéristiques de comportement du cas. Mais il est possible de définir des caractéristiques encore plus abstraites [CSFC93], comme des caractéristiques expliquant comment le comportement est une conséquence des autres caractéristiques, et même des caractéristiques téléologiques pour indiquer la raison d'être du cas.

Ces caractéristiques peuvent également représenter plusieurs points de vue. Par exemple, un bâtiment peut être étudié pour son architecture, ou pour sa fonction ou encore pour son histoire.

Certaines caractéristiques considérées comme importantes et discriminantes seront utilisées comme index pour retrouver le cas. Les index doivent avoir certaines propriétés [Kol91]:

- Ils doivent être suffisamment abstraits pour qu'un nombre maximum de cas soit concerné.
- Mais ils doivent être suffisamment concrets pour être utilisables dans les raisonnements futurs.
- Ils doivent permettre de prédire rapidement un échec ou une réussite.
- Ils doivent avoir montré leur qualité de discrimination des cas.

Ces index sont des *index potentiels* car lors de la résolution d'un nouveau problème, ils pourront être choisis soit dynamiquement (*index dynamiques*) soit statiquement (*index statiques*) (cf section 3.3.1). La figure 3.1 montre l'inclusion des différents ensembles de caractéristiques et d'index.

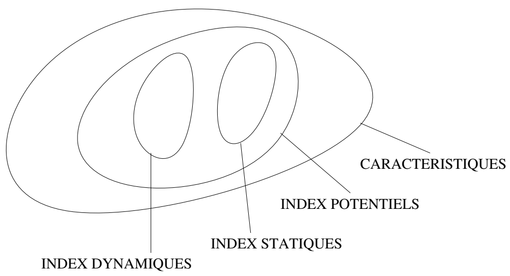

Figure 3.1 : Ensembles de caractéristiques

3.1.2 Représentation pratique

Les cas peuvent être représentés dans différents formalismes (frames, prédicats,...) et sous différentes formes (monobloc ou éclaté dans une structure plus vaste). Mais nous considérerons qu'un cas est une liste de couples attribut-valeur comme il est de coutume dans la littérature.

Les attributs peuvent avoir différents types, par exemple ReMind[Sys92], le générateur de systèmes CBR, définit les types suivants:

1. **types classiques** : texte, entier, réel, booléen, date.
2. **type symbole** : ce type est très intéressant, car non seulement il permet de définir une énumération de symboles, mais en plus ces symboles sont hiérarchisés dans un arbre. La racine de cet arbre est le symbole le plus général et les feuilles sont les symboles les plus spécifiques comme le montre l'exemple de la figure 3.2. Il est également possible de définir un ordre sur certains nœuds ou feuilles (pour plus de détails voir [Sys92] page 6-14). Cet arbre sera utilisé dans les mesures de similarité (cf. section 3.3.3) et il peut être aussi utilisé dans l'adaptation (cf. section 3.4.2).
3. **type cas** : ce type permet de référencer d'autre cas qui peuvent représenter des sous-parties du cas considéré.
4. **type formule** : la valeur de l'attribut est calculée par une formule.
5. **type liste** : ce type permet de créer une liste d'objets utilisant les types précédents.

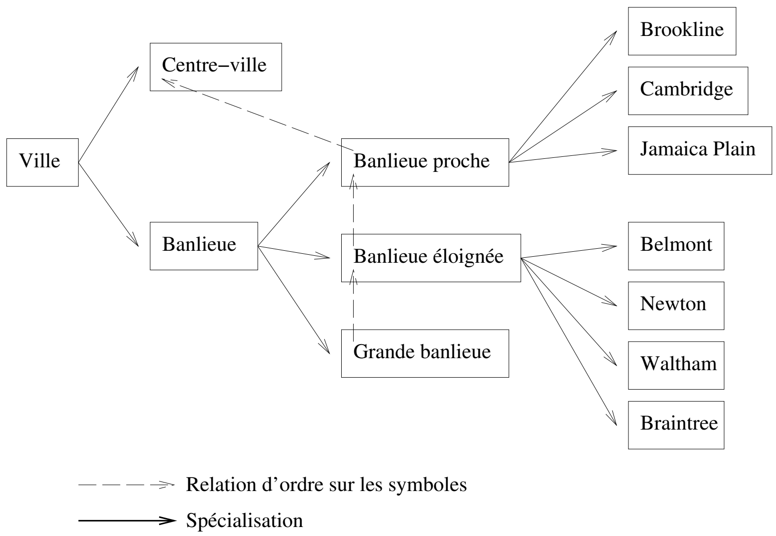

Figure 3.2 : Arbre de symboles classant les villes selon leur éloignement par rapport au centre-ville

Dans le projet FABEL[1][CDG+93], des caractéristiques représentant des tableaux de pixels (image) ou des graphes de dépendance sont également utilisées.

3.2 Organisation de la mémoire

Nous allons maintenant étudier les contraintes liées à l'organisation de la mémoire puis nous verrons les différentes solutions proposées dans la littérature et les outils CBR existants.

3.2.1 Propriétés de la mémoire

La mémoire de cas doit permettre :

- d'ajouter un nouveau fait,
- de retrouver des faits selon certains critères (index),

Les ajouts et les recherches doivent respecter certaines contraintes. Tout d'abord, l'ajout doit se faire en assurant l'accessibilité aux anciens faits. La recherche devrait également rester de complexité constante même lorsque la mémoire augmente. En effet, même si un être humain augmente sa connaissance, sa rapidité de réaction est toujours aussi bonne et même meilleure [Kol83]. De toute manière, le CBR est intéressant lorsque la base de cas est importante et c'est pourquoi il faut tout faire pour que la recherche des cas soit la plus rapide possible. Il faut donc organiser la mémoire suivant une structure qui permette de ranger les faits de manière à les retrouver facilement. La solution traditionnelle à ce problème est *l'indexation*. L'indexation est basée sur une structure de

[1] Le projet FABEL du GMD à Bonn fait des recherches sur l'intégration de techniques utilisant des cas, dans des systèmes à base de connaissances et plus particulièrement dans le domaine de l'architecture.

données avec des index potentiels et sur un algorithme de recherche avec des index dynamiques et statiques utilisant cette structure.

Dans les sections suivantes, nous allons détailler les différentes organisations de la mémoire des cas qui seront utilisées dans la recherche (cf section 3.3) à travers les algorithmes d'indexation et dans le stockage (cf section 3.5.2). On peut distinguer deux grandes catégories d'organisation : l'organisation linéaire (section 3.2.2) et les organisations hiérarchiques (sections 3.2.3, 3.2.4, 3.2.5 et 3.2.6).

3.2.2 Organisation linéaire

C'est la plus simple des organisations; il faut donc ne pas oublier de la mentionner surtout parce qu'elle sera utilisée de manière implicite en combinaison avec les différents organisations suivantes lorsqu'on parle d'ensemble de cas.

3.2.3 Arbre discriminant

Dans ce type d'organisation, nous avons un arbre dont chaque nœud correspond à une question. Cette question porte sur la valeur d'un des attributs du problème (index potentiels). Chaque fils d'un nœud correspond à une réponse possible c'est-à-dire à une valeur d'un attribut. Dans le cas d'attributs numériques la question peut poser une contrainte sur un intervalle. Les feuilles de l'arbre contiennent tous les cas qui répondent aux questions suivant le chemin qui mène de la racine de l'arbre à la feuille. Un tel arbre est appelé un *arbre de classement* ou aussi un *arbre de décision* , dans le cas où l'arbre est binaire on parle d'un *arbre de régression.*

Le but de cet arbre est d'arriver le plus vite possible à classer un cas dans une des feuilles. L'arbre doit donc avoir certaines qualités. Tout d'abord, il doit être le plus pertinent possible, c'est-à-dire que les questions doivent être posées dans un bon ordre. De plus, la hauteur de l'arbre doit être la plus petite possible.

Cet arbre doit être construit de manière automatique, car s'il était défini statiquement, il serait un cas particulier d'une hiérarchie de prototypes (cf section 3.2.4). Nous pouvons construire cet arbre par deux méthodes :

- Premièrement, nous pouvons définir un ordre d'importance a priori des caractéristiques (index potentiels) d'un cas. Ainsi les questions sont posées de manière séquentielle, c'est-à-dire que quelles que soient les branches de l'arbre, à chaque niveau on pose la même question. La question la plus importante – la plus pertinente – est posée en premier. La figure 3.3 donne un exemple où l'ordre de classement est (couleur, taille, figure). Dans le cas où il n'y a qu'une seule réponse possible, la question n'est pas posée, nous obtenons ainsi un arbre plus compact.

  Mais cette organisation pose deux problèmes. Tout d'abord l'importance des caractéristiques dépend généralement du problème posé et il est difficile de donner un ordre a priori. De plus, lors de la recherche, il se peut qu'une réponse soit impossible à donner et le système sera alors obligé de descendre dans les différents sous-arbres. Une solution à ces problèmes est de gérer plusieurs arbres différents avec des ordres de classements différents. Nous pouvons par exemple enrichir la classification faite à la figure 3.3 par l'arbre de la figure 3.4 où l'ordre de classement est (figure, taille, couleur). Mais il est bien sûr impossible de gérer toutes les possibilités d'arrangement des attributs à cause de la saturation de l'espace mémoire. Cependant, nous pouvons garder certains ordres qui ont montré leur pertinence.

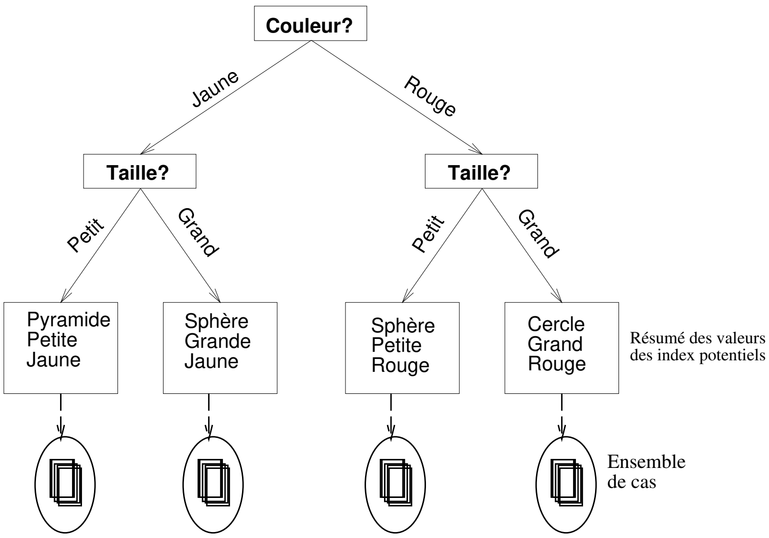

Figure 3.3 : Exemple d'arbre discriminant

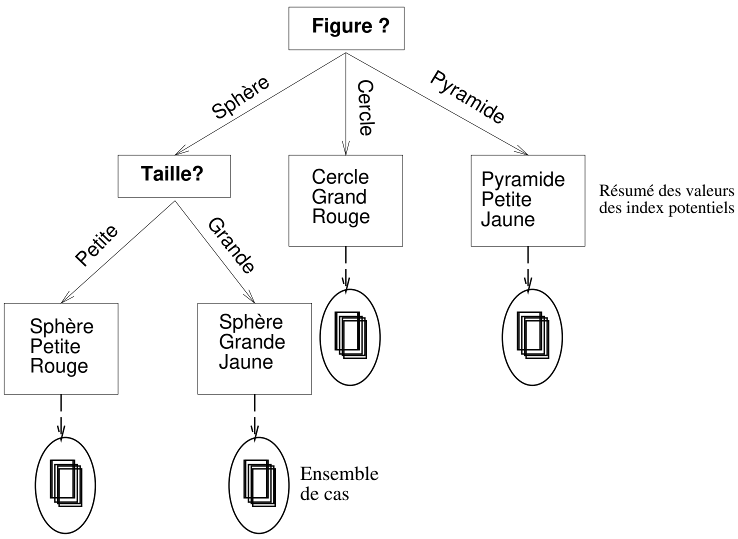

Figure 3.4 : Exemple d'arbre discriminant

Un autre problème de cette organisation est que souvent l'intérêt d'une question peut dépendre grandement de la réponse à la question précédente. Nous allons voir que la seconde méthode de construction permet de résoudre ce problème.

- Nous pouvons également utiliser des algorithmes d'induction (algorithme ID3[2] [Qui79] ou CART [BFOS84]), dans le cas où le but (valeur d'une caractéristique) est bien défini. Ces algorithmes étudient à chaque nœud quelle est la question à poser pour optimiser le gain d'informations, en prenant en compte une base de cas déjà existante ainsi que des poids d'importance sur les caractéristiques, des modèles qualitatifs du domaine et les questions déjà posées. Lorsque ce type de méthode est utilisé, on parle *d'arbre d'induction*. Cette technique permet donc de résoudre le deuxième problème de la méthode précédente, mais il reste nécessaire d'avoir plusieurs arbres dans le cas ou il existe plusieurs buts possibles (ReMind qui utilise cette organisation de la mémoire gère ce problème grâce à la création de *vues* pour chaque but). Ces algorithmes sont coûteux (23 minutes pour 29 cas de 22 caractéristiques avec ReMind) et demande une base de cas importante pour que la construction soit la plus efficace possible.

3.2.4 Hiérarchie de prototypes

Un prototype permet de décrire des conditions sur les caractéristiques des cas et tous les cas qui vérifient ces conditions sont associés à ce prototype. Ces prototypes sont organisés dans une hiérarchie d'héritage qui permet de spécifier des prototypes généraux et d'autres plus spécifiques. Ce type d'organisation est proposé dans ReMind.

La figure 3.5 donne un exemple d'organisation prototypique.

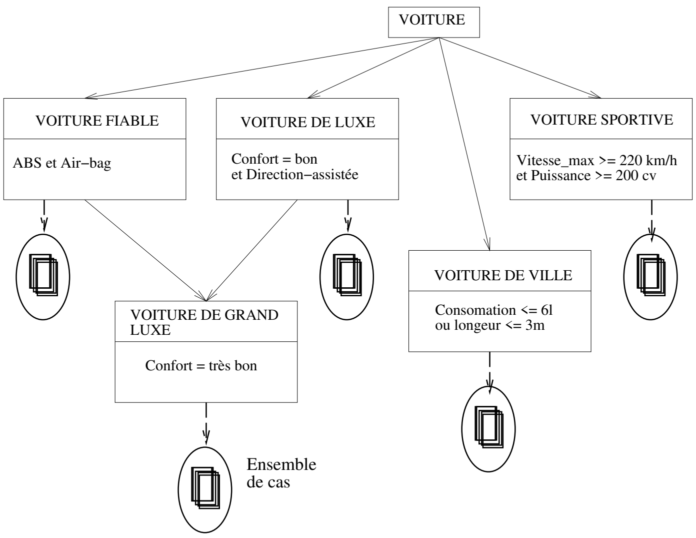

Figure 3.5 : Hiérarchie de prototypes

[2]Iterative Dichotomizer Three

3.2.5 Hiérarchie de prototypes et arbre d'induction

Dans cette combinaison d'organisation, un arbre d'induction peut être associé à chaque prototype. Cette organisation est utilisée dans ReMind et présente deux avantages : elle permet d'améliorer l'organisation par arbre d'induction grâce à une connaissance a priori du domaine, qui est modélisée par cette hiérarchie statique de prototypes; de plus, la construction de chaque arbre d'induction est plus efficace car elle ne prend en compte que les cas associés au prototype et non l'ensemble des cas tout entier.

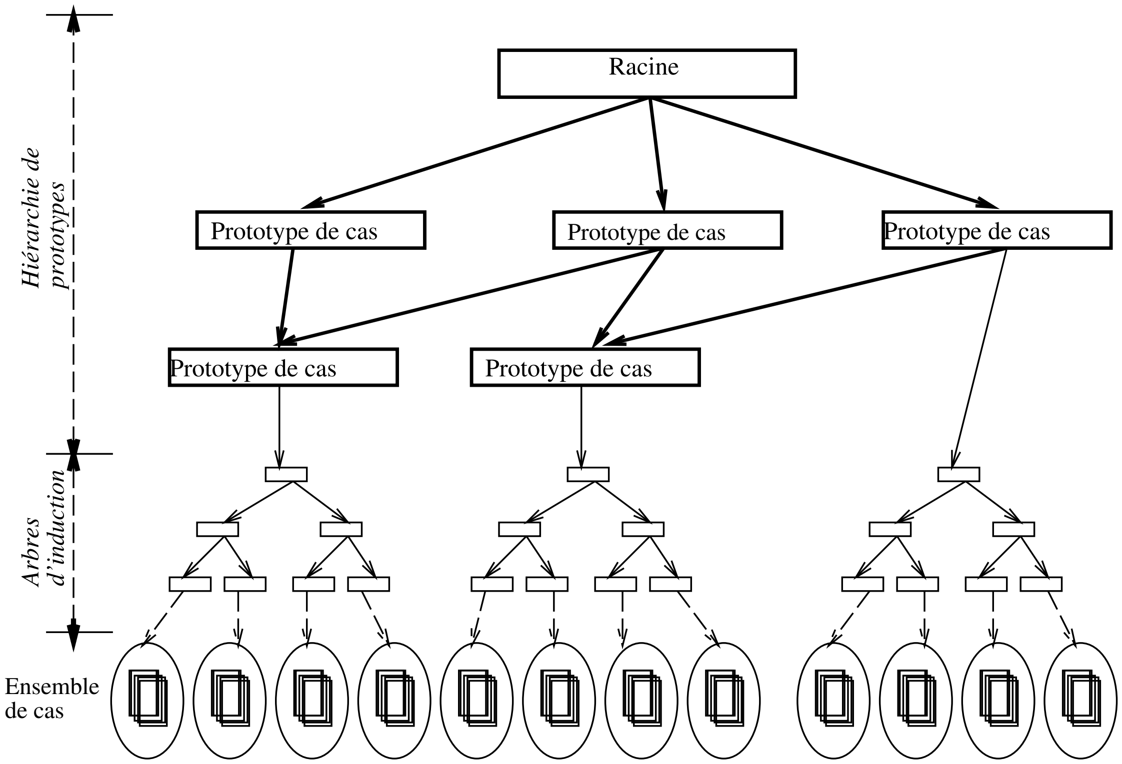

Figure 3.6 : Hiérarchie de prototypes et arbre d'induction

3.2.6 Hiérarchie de MOPs

L'organisation en MOPs[3] est basée sur les travaux de de R. C. Shank [Sch82]. Un MOP peut être défini comme une entité qui représente un cas (instance ou événement) ou une abstraction de cas. Les MOPs sont reliés entre eux par différents liens [RS89]:

- liens d'abstraction pour former une hiérarchie d'abstraction,
- liens avec les scènes qui composent un MOP, pour former une hiérarchie de composants,
- liens avec des exemples particuliers,

[3] MOP : Memory Organization Package.

- liens avec d'autres MOPs grâce à des index potentiels (on a alors un arbre de discrimination),
- liens avec des instances qui ont mené à un échec.

La figure 3.7 donne un exemple qui a été réalisé dans CYRUS [Kol83] qui montre bien les liens d'index potentiels. D'autres applications comme CHEF, JUDGE [RS89], utilisent des hiérarchies de MOPs simplifiés.

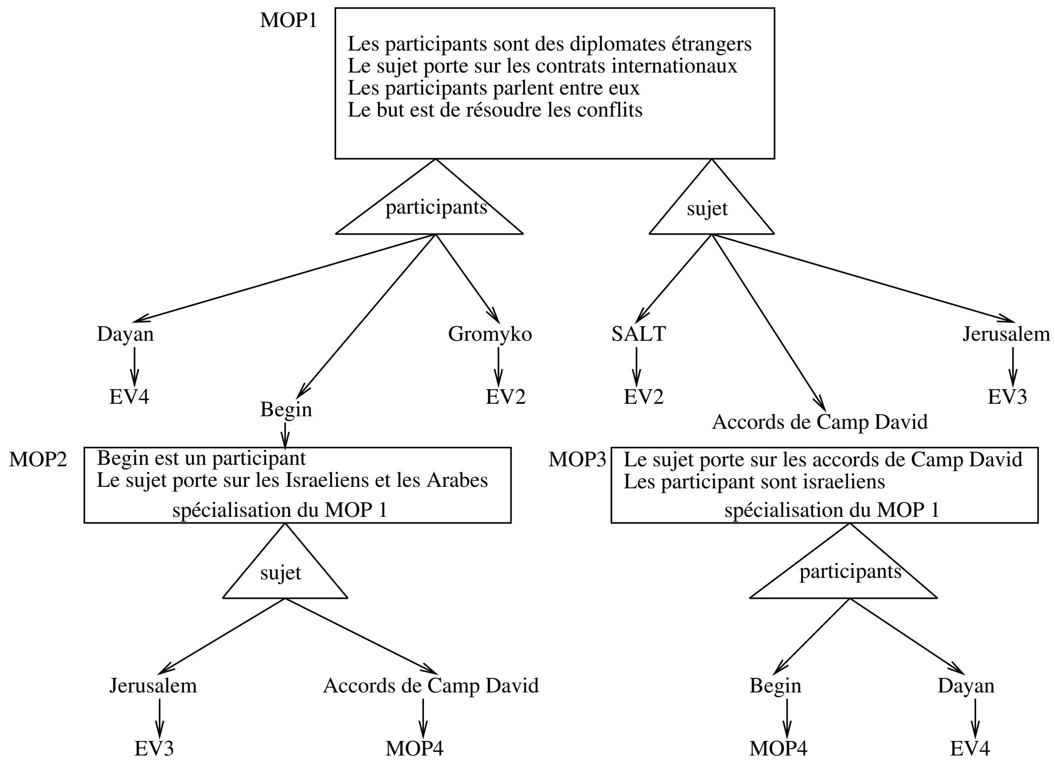

Figure 3.7 : Exemple de hiérarchie de MOPs représentant des réunions diplomatiques.

3.3 Recherche des cas similaires

La recherche se décompose en deux grandes étapes : le filtrage et la sélection. Ces deux étapes utilisent des index statiques et/ou des index dynamiques qui vont effectivement guider la recherche. Nous allons donc décrire les différentes techniques de choix des index puis nous détaillerons l'étape de filtrage et enfin l'étape de sélection.

3.3.1 Choix des index

Nous rappelons que des index sont des caractéristiques particulières d'un problème, qui vont être utilisés par les algorithmes de recherche. Nous pouvons citer plusieurs méthodes de choix [KR89], ainsi on peut choisir :

- toutes les caractéristiques,
- certaines caractéristiques,
- les caractéristiques qui ont été les plus déterminantes dans le passé,
- les caractéristiques qui sont les plus différentes de celles des cas qui sont déjà en mémoire.

Ces index doivent toutefois appartenir à l'ensemble des index potentiels. Certains index peuvent être choisis lors de la conception, on parle alors d'*index statiques*, sinon le choix des index peut se faire dynamiquement de manière automatique ou grâce à un dialogue homme-machine ou bien par un combinaison des deux méthodes, il s'agit alors d'*index dynamiques*.

Il est parfois possible de préciser l'importance relative des différents index grâce à des coefficients, des formules ou des modèles qualitatifs. Mais, il n'est pas toujours facile de connaître et d'évaluer l'importance d'une caractéristique par rapport aux autres. De plus cette importance dépend du domaine d'application, du but à atteindre lors du raisonnement, et également des valeurs des autres caractéristiques du problème.

3.3.2 Filtrage

Dans cette étape, grâce à un parcours de la mémoire, il faut isoler un certain nombre de cas qui seront utilisés dans la suite du raisonnement. Le filtrage n'est pas toujours utilisé et il est possible de passer directement à l'étape de sélection. Il existe plusieurs algorithmes d'indexation basés sur des organisations différentes de la mémoire et quelquefois sur le choix d'index dynamiques :

- **indexation par arbre discriminant** : on utilise bien sûr un arbre discriminant. On part de la racine et on répond aux différentes questions jusqu'à arriver à une feuille. Si la valeur d'une caractéristique est inconnue, l'algorithme est obligé d'examiner les différentes alternatives possibles. Dans ReMind, ce dialogue question-réponse peut être interactif ou se faire à partir d'une description préalable du problème. Toutefois si une valeur est inconnue, ReMind peut poser la question à l'opérateur. Les caractéristiques mises en jeu dans les questions sont considérées comme des index statiques.

- **indexation par induction dynamique** : dans ce cas une structure linéaire est utilisée, mais un algorithme d'induction permet de construire dynamiquement certaines branches de l'arbre d'induction. En effet, en prenant en compte les index dynamiques, l'algorithme évalue à chaque nœud le gain d'information pour choisir dynamiquement la caractéristique discriminante suivante. Cette méthode permet de s'adapter au problème (but non fixé) et de mieux prendre en compte les valeurs inconnues. Le seul inconvénient de cette méthode est son coût puisqu'il faut analyser à chaque nœud l'ensemble des cas qui ont répondu aux conditions des nœuds précédents. Toutefois, les concepteurs du système PATDEX[MA93] basé sur cette méthode, montrent que les ensembles de cas se réduisent très vite et que l'efficacité reste bonne.

- **indexation prototypique** : on part toujours de la racine et on descend dans les fils si les conditions sont vérifiées. Ces conditions portent sur des index statiques. Toute condition faisant intervenir une valeur inconnue est supposée vraie. On récupère alors l'ensemble des prototypes les plus spécifiques et grâce à l'association prototype-ensemble de cas, on obtient un ensemble de cas.

- **indexation prototypique et inductive** : un algorithme d'indexation prototypique est utilisé en premier, puis au lieu de s'arrêter, il faut continuer avec une indexation inductive sur les arbres d'induction associés aux prototypes les plus spécifiques.
- **indexation par MOP** : les valeurs des index dynamiques agissent comme des clés qui permettent de passer d'un MOP à l'autre jusqu'à arriver au MOP le plus spécifique compatible avec le problème.

3.3.3 Sélection

Il s'agit maintenant de classer les cas obtenus par le filtrage, dans l'ordre de leur similitude ou de leur intérêt par rapport au problème posé. Il suffira alors de choisir le meilleur cas, ou un ensemble de cas qui sont en tête du classement pour poursuivre le raisonnement. Si l'étape de filtrage donne déjà le nombre de cas voulu, l'étape de sélection ne sera pas exécutée. Cette étape correspond à l'*indexation par plus proches voisins*. Cette méthode d'indexation nécessite le choix d'index dynamiques avec leur coefficient d'importance. Il s'agit alors de mesurer la similarité des valeurs de chaque index puis d'agréger ces résultats pour obtenir la mesure de similarité entre le problème et un ancien cas. Ces mesures permettront alors de classer les différents cas.

Mesures de similarité entre deux index

Chaque type d'index doit posséder une fonction qui doit donner le degré de similitude entre deux valeurs appartenant au domaine de valeur. Cette fonction sera appelée *sim* et elle peut être définie ainsi :

$$sim : \quad D \times D \quad \longrightarrow \quad [a, b]$$

Normalement, la similitude est d'autant plus grande que la valeur de cette fonction se rapproche de b.

Nous pouvons maintenant présenter des exemples de fonction de mesure de similarité. Dans le cas de ReMind, nous avons $a = 0$ et $b = 100$, et suivant le type de l'attribut nous avons[4] :

- type texte, booléen, cas :

$$sim(x, y) = 100 \text{ si } x = y \text{ , 0 sinon}$$

- type symbole :

$$sim(x, v) = 100 - 100 * \frac{\psi(MSCA(x, y), x) + \psi(MSCA(x, y), y)}{\psi(racine, x) + \psi(racine, y)}$$

Avec ψ :

$$\begin{array}{llll} \psi : & D \times D & \longrightarrow & I\!N \\ & (x, y) & \longmapsto & \text{nombre d'arêtes reliant } x \text{ à } y \\ & y \subseteq x & & \end{array}$$

[4] Les notations utilisées ont été modifiées par rapport au manuel de ReMind.

Où la relation $\subseteq$ signifie "plus spécifique que ", et la fonction $MSCA$[5] :

$$\begin{array}{rlll} MSCA: & D \times D & \longrightarrow & D \\ & (x,y) & \longmapsto & \text{Abstraction la plus spécifique} \\ & & & \text{entre } x \text{ et } y \end{array}$$

Par exemple, si nous prenons la hiérarchie de la figure 3.2, et si nous voulons calculer la similarité entre Brookline et Cambridge, nous avons :

$$MSCA(\text{Brookline}, \text{Cambridge}) = \text{Banlieue-proche}$$

Et :

$$\begin{array}{rl} sim(\text{Brookline}, & \text{Cambridge}) \\ & = 100 - 100 * \frac{\psi(\text{Banlieue-proche,Brookline}) + \psi(\text{Banlieue-proche,Cambridge})}{\psi(\text{Ville,Brookline}) + \psi(\text{Ville,Cambridge})} \\ & = 100 - 100 * \frac{1+1}{3+3} \\ & = 66,7 \end{array}$$

De même pour la similarité entre Brookline et Belmont :

$$MSCA(\text{Brookline}, \text{Belmont}) = \text{Banlieue}$$

Et :

$$\begin{array}{rl} sim(\text{Brookline}, & \text{Belmont}) \\ & = 100 - 100 * \frac{\psi(\text{Banlieue,Brookline}) + \psi(\text{Banlieue,Belmont})}{\psi(\text{Ville,Brookline}) + \psi(\text{Ville,Belmont})} \\ & = 100 - 100 * \frac{2+2}{3+3} \\ & = 33,3 \end{array}$$

Ainsi Brookline est plus proche de Cambridge que de Belmont. D'autres formules du même type permettent de prendre en compte les relations d'ordre sur les symboles.

- type numérique (entier, réel, date) :

$$sim(x,y) = 100 - 100 * \left|\frac{x-y}{max - min}\right|$$

Les nombres min et max sont calculés dynamiquement à partir de la moyenne, du minimum et du maximum des valeurs de l'attribut dans la base de cas.

- type liste multi-valuée:

$$sim(x,y) = 100 * \frac{|x \cap y|}{|x \cup y|}$$

Dans le projet FABEL[CDG$^+$93], nous avons $a = 0$ et $b = 1$, et nous pouvons également trouver des formules de similarité qui dépendent du type de l'attribut :

[5]MSCA : Most Specific Common Abstraction.

- type booléen :
$$sim(x, y) = 1 \text{ si } x = y \text{ , } 0 \text{ sinon}$$

- type numérique :
$$sim(x, y) = 1 - \frac{|x - y|}{\text{borne-sup} - \text{borne-inf}}$$

- type liste non ordonnée :
$$sim(x, y) = 1 \text{ si } x = y \text{ , } 0 \text{ sinon}$$

- type liste ordonnée :
$$sim(x, y) = 1 - \frac{|position(x) - position(y)|}{\text{cardinal-de-la-liste}}$$

- type ensemble multi-valué :
$$sim(x, y) = \frac{|x \cap y|}{|x \cup y|}$$

Sur des domaines simples, ces formules sont parfois triviales à trouver, mais sur des domaines plus complexes (graphes de dépendances, images), il est difficile de trouver des formules efficaces et valides.

Mesures de similarité entre le problème et un cas

Il est donc possible de comparer chaque index de deux cas, il faut maintenant évaluer le degré de similarité global entre deux cas en combinant les différentes mesures. Mais la représentation d'un problème et d'un ancien cas ne fait pas toujours intervenir le même ensemble d'attributs. Des index du problème peuvent ne pas exister dans un ancien cas ou être inconnus. Notons alors A l'ensemble des attributs communs et connus entre le problème P et un ancien cas C. Soit $p(a)$ le poids de l'index a, $pv(a)$ la valeur de l'index dans le problème et $cv(a)$ la valeur de l'index dans un ancien cas. Le degré de similitude est alors obtenu par la formule suivante [RS89] :

$$\frac{\sum_{a \in A} p(a) * sim(pv(a), sv(a))}{\sum_{a \in A} p(a)}$$

D'autres systèmes utilisent des poids différents, dans le cas où la caractéristique est spécifiée ou non dans le problème à résoudre [Har91].

Heuristiques de sélection

La méthode que nous venons de voir constitue la technique de base mais nous pouvons l'enrichir avec des heuristiques de sélection. J. L. Kolodner [Kol89] souligne le fait que la sélection ne doit pas forcement trouver le ou les cas les plus similaires mais plutôt les plus utiles. Elle définit également des heuristiques de préférence. Ainsi, il vaut mieux sélectionner en priorité les cas :

1. qui résolvent déjà une partie des buts du problème,

2. qui partagent le plus de caractéristiques importantes,
3. les plus spécifiques,
4. les plus utilisés,
5. les plus récents,
6. les plus faciles à adapter.

3.4 Adaptation

L'adaptation permet, à partir de la description du problème à résoudre et du (des) plus proche(s) cas sélectionné(s) dans la mémoire, de donner une solution. Cette adaptation est généralement indispensable car il est très improbable qu'il existe un cas déjà rencontré qui soit exactement identique au problème à résoudre.

En fait, il y a quatre étapes dans l'adaptation:

1. identifier ce qu'il faut adapter,
2. choisir des méthodes d'adaptation,
3. ordonner les méthodes,
4. faire l'adaptation.

Les trois premières étapes constituent la partie d'analyse. L'adaptation doit se faire en respectant les contraintes du problème. Ces étapes sont généralement réalisées par des mécanismes qui sont très liés au domaine d'application. Un point important est que les différentes méthodes que nous allons détailler, effectuent des modifications qui peuvent entraîner à leur tour de nouvelles adaptations.

3.4.1 Identifier ce qu'il faut adapter

Il existe plusieurs méthodes [KR89] qui sont basées sur les points suivants :

- après une comparaison de structure entre le cas retrouvé dans la mémoire et le problème à résoudre, le système met en évidence des différences à corriger;
- le système peut vérifier une liste de caractéristiques bien définies mais dépendant du type de problème à résoudre;
- si la solution ne satisfait pas les buts et les contraintes définis dans le problème, le système doit les corriger;
- le système peut être guidé par les échecs passés ou les explications données dans la phase d'évaluation.

3.4.2 Méthodes d'adaptation

Dans le cas où le système doit exécuter plusieurs méthodes d'adaptation, il commencera d'abord par les méthodes les plus spécifiques, puis les plus simples et en dernier les plus complexes. Nous présentons maintenant quelques techniques d'adaptation [RS89] [KR89].

Transformation

La transformation d'un cas correspond à des actions effectuées sur les caractéristiques qui décrivent le cas. Ces actions peuvent être: l'effacement d'une caractéristique, l'ajout d'une caractéristique, la substitution de la valeur d'une caractéristique par une autre.

Dérivation

Cette méthode s'applique lorsqu'un cas contient non seulement la solution au problème mais aussi les étapes du raisonnement qui ont mené à cette solution. Ces étapes sont alors vérifiées (du point de vue de la validité et de la nécessité des actions menées sur les différents objets) et transformées (ajout ou effacement). Dans certains cas, cette méthode peut se ramener à de simples substitutions, ou bien elle fait appel à d'autres méthodes (ajustement de valeurs, abstraction et spécialisation).

Ajustement de valeurs

Dans certains cas la valeur d'une caractéristique dépend des autres caractéristiques, il faut donc ajuster cette valeur dans le contexte du problème à résoudre. Nous pouvons envisager d'utiliser plusieurs techniques pour ajuster une valeur : une formule mathématique ou statistique qui fait les liens entre le ou les cas qui ont été retrouvés par le système et le problème courant, ou bien un raisonnement par règles, ou encore d'autres heuristiques.

Par exemple, JUDGE va essayer d'ajuster la durée de l'emprisonnement suivant la gravité du crime: si le cas retrouvé dans la mémoire de cas donne une durée de 25 ans d'emprisonnement mais que le cas courant est moins grave, JUDGE réduira la peine à 15 ans.

Abstraction et spécialisation

Grâce à l'utilisation d'une hiérarchie de classification, il est possible dans un premier temps d'abstraire la valeur d'une caractéristique puis de la re-spécialiser.

Par exemple, on pose à CHEF le problème de construire un menu pour végétarien et de cuisine française. Après recherche dans la mémoire de cas, CHEF propose une bouillabaisse comme plat principal, ce qui ne résout pas le problème . Les plats sont organisés dans une hiérarchie comme le montre la figure 3.8 [RS89]. Le système va alors abstraire puis spécialiser pour enfin choisir la ratatouille.

Dans ce cas, la recherche est locale, mais d'autres systèmes comme SWALE utilisent de plus grandes abstractions dans le but de faire des adaptations plus créatives. Si l'idée d'abstraire puis de spécialiser s'applique à la description de cas tout entier, on parle alors de ré-instanciation. Le problème central de cette méthode est de savoir jusqu'où il faut abstraire et jusqu'où il faut spécialiser.

Interrogation de la mémoire

Lorsqu'un cas est composé de plusieurs parties et que l'une d'entre elles ne répond pas aux contraintes du problème, le système peut faire une recherche dans la base de cas pour retrouver une sous-partie mieux adaptée. Cette méthode peut être appliquée dans le cas où la méthode d'abstraction puis de spécialisation n'a pas fonctionné.

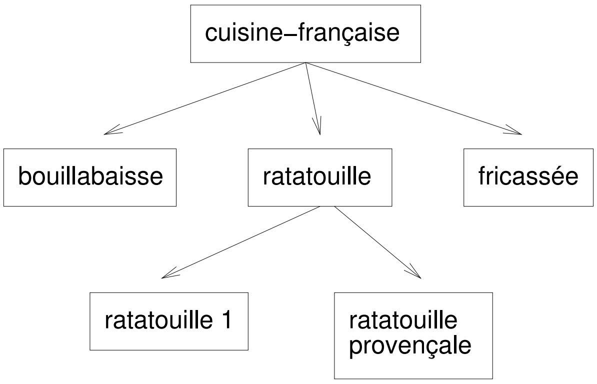

Figure 3.8 : Arbre d'abstraction

Connaissances du domaine

Le système reconnaît une certaine configuration de caractéristiques du cas et déclenche une ou plusieurs des méthodes précédentes. Ce type d'adaptation convient bien à une représentation par une base de règles. Bien sûr les règles qui expriment ces connaissances sont beaucoup plus réduites que pour un système expert car il s'agit simplement de modifier un ancien cas et non pas de construire de toute pièce la solution.

Par exemple, le système CHEF sait que lorsqu'on doit cuire un canard, on doit d'abord enlever la graisse, il pourra donc ajouter cette étape dans la recette.

3.5 Apprentissage

3.5.1 Evaluation, explication et réparation

L'évaluation est la confrontation entre la solution et le monde réel. Il peut s'agir d'une vérification de la solution par un expert, par une simulation, ou même par une exécution réelle. C'est une étape importante puisque c'est à ce moment que le système est informé de la qualité de la solution. Lors de l'évaluation, deux types d'échecs peuvent être constatés [RS89]:

- les buts spécifiés du problème ne sont pas atteints,
- les contraintes implicites ne sont pas respectées.

Si la solution est mauvaise, des explications doivent être données pour que le système comprenne les raisons de l'échec; des mécanismes de sanctions et de récompenses peuvent même être utilisés. Ensuite, l'étape d'amélioration aussi appelée réparation, est exécutée. L'amélioration est en fait une nouvelle adaptation, la seule différence est que l'adaptation se fait à partir des cas retrouvés dans la mémoire alors que l'amélioration est basée sur les explications de l'échec de la solution.

Dans ce processus, toutes les méthodes d'adaptation peuvent être de nouveau exécutées.

3.5.2 Stockage

Dans cette étape, le système doit choisir des index pour le stockage de manière à ce que le nouveau cas soit accessible pour un nouveau raisonnement. L'importance de ce cas pour les futurs raisonnements doit donc être évaluée. Il faut également tenir compte du fait que la solution proposée par le système a été un échec ou non. Les contraintes implicites mises en évidence lors de l'évaluation doivent également être reliées à la solution pour qu'elles soient connues du système. Cette étape peut aussi impliquer la réorganisation d'une partie de la mémoire de cas. En fait, cette étape doit faire la synthèse du raisonnement qui a été suivi de manière à sauvegarder les parties importantes. Ainsi lors d'un prochain problème, le raisonnement ne sera pas à refaire.

Plus concrètement, les mécanismes dépendent de l'organisation de la mémoire :

- Organisation linéaire : la méthode est triviale.
- Arbre discriminant : l'arbre est à reconstruire en prenant en compte le nouveau cas.
- Hiérarchie de prototypes (ReMind): la hiérarchie est statique donc il n'y a pas de modification.
- Hiérarchie de MOPs : la hiérarchie peut être modifiée par trois mécanismes [RS89] :
  - création de nouvelles instances sous le MOP le plus spécifique,
  - création de nouvelles abstractions lorsque le nombre d'instances d'un MOP qui ont une caractéristique commune, dépasse un certain seuil (normalement $\frac{2}{3}$),
  - création de nouveaux index potentiels.

Chapitre 4

Analyse du sujet

Notre sujet est de concevoir et de réaliser un système de CBR réduit à la phase de recherche des cas similaires dans le cadre de la réutilisation de conception. Nous allons tout d'abord détailler les objectifs et les limites de notre étude puis nous présenterons les problématiques de conception et de réalisation ainsi que les principaux choix effectués.

4.1 Objectifs et limites

Nous allons commencer par mettre en évidence les différents points qui nous semblent importants dans la résolution de problèmes en conception et qui seront la base de notre problématique de conception et de réalisation. Nous pourrons alors situer ces différents points par rapport aux systèmes CBR existants. Nous définirons enfin les limites de notre étude.

4.1.1 Caractéristiques des problèmes de conception

Dans notre étude nous voulons prendre en compte certaines caractéristiques des problèmes de conception :

- Représentation des problèmes et des solutions : les problèmes de conception peuvent être définis comme un ensemble de contraintes sur des objets. Or les objets qui sont mis en jeu sont souvent composites par nature. Par exemple pour concevoir une voiture, il faut concevoir entre autre la carrosserie et le moteur, la conception d'un moteur à son tour se décompose également en sous-problèmes et ainsi de suite.

  La solution d'un problème est également décrite par des objets composites. Une voiture existante est décrite par les caractéristiques de la carrosserie et du moteur, et le moteur est décrit par les caractéristiques de ses sous-parties, etc. Une solution peut également représenter les étapes de la conception, dans ce cas également les étapes de conception peuvent se décomposer en sous-étapes et une étape de conception peut être considérée comme un objet.

- Description des problèmes : lors de la description d'un problème, il est nécessaire d'exprimer des préférences, l'imprécis, l'inconnu et des points de vue.

  - Les préférences : il existe deux sortes de préférences. Tout d'abord, il faut pouvoir exprimer les préférences sur les valeurs d'une caractéristique du problème. Par exemple, il faut concevoir une voiture qui pèse 1,5 tonnes plutôt que 2 tonnes. Les préférences peuvent faire intervenir plusieurs caractéristiques; par exemple, il faut concevoir une voiture qui respecte les contraintes de poids plutôt que les contraintes de confort de l'habitacle.

- L'imprécis : l'imprécis peut prendre plusieurs formes, par exemple, il faut concevoir une voiture *rapide*, ou bien une voiture dont le poids doit être *approximativement* de 1,5 tonnes.
- L'inconnu : les contraintes sur certaines caractéristiques du problème à résoudre peuvent être inconnues au départ.
- Les points de vue : la description d'un problème peut porter sur un ou plusieurs points de vue d'un même objet. Par exemple, une voiture peut être conçue dans l'optique de son esthétique ou bien de sa robustesse ou encore de sa consommation.

• Démarche de conception : le problème peut se résoudre par assemblage, c'est-à-dire que l'on peut résoudre des sous-problèmes séparément puis les assembler. Par exemple, une voiture peut être conçue par assemblage de la résolution des problèmes liés au châssis, au moteur et à la carrosserie.

4.1.2 Etude de l'existant

Nous pouvons trouver une bonne comparaison des différents systèmes CBR en conception dans [TV93]. De notre côté, nous voulons situer plus particulièrement les systèmes CBR spécifiques ou non à la conception, par rapport aux différentes caractéristiques des problèmes de conception que nous venons d'énoncer.

Parmi les outils de CBR, les systèmes basés sur une organisation de hiérarchie de MOPs permettent de gérer des objets composites (par exemple CHEF et JUDGE [RS89]). Dans le domaine de la conception, seul le système CADET [SNC93] utilise des objets composites. D'un autre côté, ReMind [Sys92] ne gère pas une structure composite des cas et toutes les caractéristiques sont représentées de manière linéaire. Cette méthode simplifie les algorithmes mais engendre un manque de clarté qui peut rendre la maintenance difficile. De plus, cette linéarisation représente une perte d'information par rapport à la structuration souvent naturelle des problèmes. Par exemple, la SITA[1] a développé une application CBR en utilisant ReMind pour une aide au choix d'un système de messagerie électronique. Cette réalisation fait intervenir des cas décrits par 200 caractéristiques différentes. Une structuration des cas aurait permis, à notre avis, une meilleure modélisation, et à plus long terme elle aurait amené plus de possibilités d'évolution du système.

Les systèmes gèrent souvent des poids d'importance entre les diverses caractéristiques du problème ce qui permet de représenter les préférences entre les caractéristiques. Par contre, aucun système ne gère des préférences de valeurs pour une caractéristique.

Nous n'avons jamais rencontré, dans la littérature et dans les outils de CBR, un véritable moyen permettant la représentation de l'imprécis. Bien sûr, il est généralement possible de représenter une pseudo-imprécision grâce à des symboles, mais il s'agit en fait d'une transformation de l'imprécis en une représentation exacte qualitative.

Pour les points de vue, les systèmes permettant le choix d'index dynamiques (par exemple ReMind lors de l'indexation par plus proches voisins) donnent un moyen de sélectionner un ensemble

[1]SITA : Société Internationale de Télécommunications Aéronautiques - établissement de Sophia Antipolis.

de caractéristiques représentant un point du vue. Mais étant donné que ces systèmes ne gèrent pas une représentation composite des points de vue, il y a un manque de clarté dans cette représentation.

Seul le système CADET [SNC93] permet de faire de l'assemblage dans le domaine de la conception. Par ailleurs, les autres outils ne sont pas satisfaisants car il faudrait créer une base de cas pour chaque partie. Ici encore, une modélisation composite des cas serait intéressante.

4.1.3 Limites de l'étude

Dans le cadre de la spécification et de la réalisation d'un générateur de systèmes de CBR d'aide à la conception, nous nous limiterons pendant ce stage à l'étude de tout ce qui est nécessaire à la phase de recherche : représentation des cas, organisation de la mémoire et mécanismes de recherche.

4.2 Problématique de la conception de la phase de recherche

Nous allons maintenant étudier les différents problèmes et les solutions adoptées pour la phase de recherche. La recherche repose sur deux grands problèmes : la représentation des cas, et le choix de méthodes d'indexation basées sur une organisation de la mémoire et sur un algorithme d'indexation.

4.2.1 Représentation des cas

Un cas doit permettre la représentation du problème et de la solution associée. Nous avons vu que les problèmes et les solutions se représentent de manière composite, donc un cas doit pouvoir également être composite. Nous pensons donc naturellement à une représentation multi-hiérarchique de prototypes permettant de définir le plus facilement possible les objets décrivant le problème et la solution (cf figure 4.1). Chaque hiérarchie permet de décrire une partie d'un cas grâce à la définition d'attributs qui possèdent un domaine de valeurs et une restriction du domaine. Ces attributs sont créés au niveau d'un prototype et sont hérités dans les prototypes plus spécifiques. De plus, les restrictions peuvent être redéfinies pour mieux décrire les prototypes. Certains attributs peuvent avoir une hiérarchie de prototypes comme domaine de valeurs, ce qui permet de gérer les sous-parties. Cette représentation est grandement inspirée de SMECI [ILO92a] (même si nous avons réuni les concepts de *catégories* et de *prototypes* dans celui de prototype).

Pour la représentation de l'imprécis et des préférences dans la description du problème, nous avons choisi des ensembles flous car la théorie des possibilités qui les manipule est bien formalisée (cf annexe A).

Grâce à cette représentation hiérarchique, nous pouvons également gérer de manière simple les points de vue. En effet, il suffit de décrire plusieurs hiérarchies pour la même sous-partie logique suivant différents points de vue. L'inconvénient majeur est que nous obtenons ainsi plusieurs sous-parties physiques. Il serait donc intéressant d'intégrer des mécanismes plus complexes pour une gestion plus réaliste des points de vue comme nous pouvons en trouver dans le système TROPES : instanciation unique et existence de passerelles entre les hiérarchies définissant les points de vue. TROPES est un système en cours de développement dans le projet SHERPA de L'INRIA Rhones-Alpes dirigé par F. Rechenman, et il n'existe pas encore de version diffusable. Ne voulant développer de tels mécanismes dans le cadre de ce stage, nous gardons notre première modélisation.

Pour pouvoir résoudre n'importe quel problème, il faut donc que la hiérarchie mère de la description d'un problème ne soit pas fixée une fois pour toute, dans le but de résoudre des problèmes

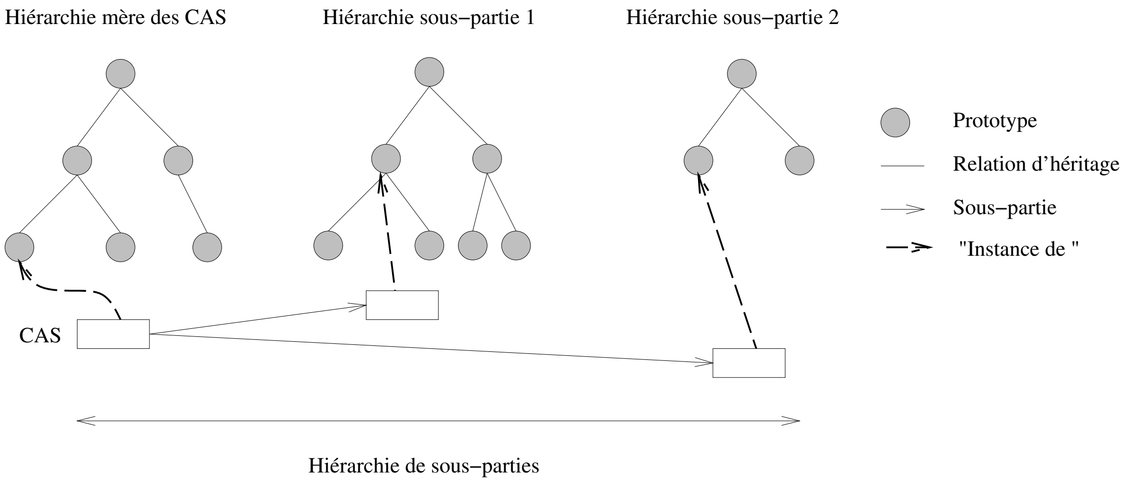

Figure 4.1 : Structure de représentation d'un cas

par assemblage.

4.2.2 Choix des méthodes d'indexation

Nous allons nous baser sur l'étude comparative des méthodes d'indexation existantes (cf figure 4.2), tout en sachant qu'aucune méthode n'est applicable directement à notre système à cause de la spécificité de notre représentation des cas. Pour pallier aux problèmes de manque d'efficacité lors de l'utilisation d'une base de cas importante, nous choisissons de manière classique de décomposer la phase de recherche en une étape de filtrage et une étape de sélection.

Le filtrage

Pour le filtrage, nous pouvons nous inspirer : de l'indexation par arbre discriminant avec un ordre statique sur les caractéristiques, de l'indexation inductive statique ou dynamique, de l'indexation prototypique de ReMind, ou d'une indexation par MOP. La première solution nous paraît trop simpliste, et l'indexation par induction statique ne nous convient pas à cause de sa mauvaise gestion de l'inconnu et de la nécessité d'avoir un but bien figé. Il existe pourtant des algorithmes d'induction gérant une représentation composite des objets comme celui présenté dans le système de classification KATE [Man89]. Nous pourrions nous baser sur les travaux très récents sur l'indexation par induction dynamique implantée dans le système PADTEX [MA93], mais notre idée est d'utiliser directement les hiérarchies construites pour la représentation des cas.

De plus, nous remarquons que le principe de classification est présent dans la plupart des algorithmes d'indexation : par exemple dans l'indexation prototypique ou l'indexation par MOP,

	Indexation par arbre discriminant avec ordre statique	Indexation inductive	Indexation inductive dynamique	Indexation prototypique (ReMind)	Indexation par plus proches voisins	Indexation par MOP
Organisation de la mémoire	arbre discrininant	arbre discrininant	structure linéaire	hiérarchie de prototypes	structure linéaire	hiérarchie de MOP
Cout de la recherche	–	–	++	–	++	–
But figé	non	oui	non	grossièrement	non	non
Avantages	–arbre simple à construire.	– l'arbre de décision synthétise les connaissances de la base de cas.	– prise en compte optimale des informations données par la description du problème.	– permet de modéliser des connaissances a priori du domaine.	– permet de classer les cas suivant leur degré de similitude.	– indexation dynamique grâce à des mécanismes automatiques de modification de la hiérarchie de MOP.
Inconvénients	– l'ordre des caractéristiques est statique.			– la hiérarchie est statique.	– validité des métriques de similarité difficile à prouver. – coefficients d'importance pas toujours faciles à déterminer.	– validité des mécanismes automatiques.

Figure 4.2 : Comparaison des méthodes d'indexation

le but est de descendre dans la hiérarchie jusqu'à trouver les prototypes ou les MOP les plus spécifiques correspondant à la description du problème. Nous nous sommes alors intéressés aux autres travaux sur la classification hiérarchique. Le générateur de systèmes expert en classification CLASSIC[ILO87] nous apparaît comme un outil de référence dans ce domaine. Après l'étude détaillée de la version interactive de CLASSIC, dont nous présentons le résumé dans l'annexe B, nous concluons que le principe de classification convient très bien à notre problème de filtrage. En effet, si nous classons un problème donné dans un ou plusieurs prototypes de problèmes, nous sommes alors capables de récupérer l'ensemble des cas appartenant à ce prototype. Ainsi, nous proposons d'utiliser les hiérarchies de prototypes grâce à un algorithme de classification basé sur certains points intéressants issus de l'étude de CLASSIC.

- L'expression du flou dans la description des prototypes et dans l'objet à classer permet une classification floue (l'appartenance à un prototype n'est pas un booléen) qui est plus réaliste. Nous avions déjà choisi d'exprimer du flou dans la description des problèmes, il suffit alors d'introduire du flou dans les restrictions de valeurs des attributs, ce qui est d'ailleurs très intéressant puisque nous savons qu'il est souvent difficile d'exprimer ces restrictions de manière précise. Dans ce but, nous utiliserons des ensembles flous.
- Le calcul de coefficients de compatibilité et d'incompatibilité utilisant la théorie des possibilités permet de réaliser la classification floue.
- Le classement et la description interactive de l'objet nous semblent importants lorsque les objets à décrire sont très complexes et qu'il est impossible de les décrire avant la classification.

Nous devons également étendre l'algorithme de classification de manière à l'adapter à un contexte de CBR, ce qui nous amène à :

- ajouter la possibilité de choisir des index dynamiques et d'indiquer leur importance (préférences) lors de la classification, la notion d'index n'existant pas en classification;
- ajouter la possibilité de lier les prototypes avec les objets classés, de manière à pouvoir récupérer les anciens cas;
- pouvoir effectuer la classification de manière non interactive en consultant une description des problèmes faite avant le classement.

Ainsi nous proposons une nouvelle technique d'indexation *l'indexation par classification floue* dans laquelle l'organisation de la mémoire correspond aux hiérarchies de description des prototypes, les cas étant associés aux prototypes les plus spécifiques avec lesquels ils sont compatibles. Pour la partie de classification et par rapport à CLASSIC, nous voulons offrir une représentation plus riche du flou et des caractéristiques des prototypes.

Notre méthode d'indexation se distingue de toutes les autres par la gestion du flou. De plus, par rapport à l'indexation par MOP, nous gardons les liens d'abstraction, de scènes, et d'une manière plus réduite des liens d'index grâce aux restrictions. Cependant notre hiérarchie est statique[2]. Par rapport à l'indexation prototypique de ReMind, nous apportons la gestion de cas composites et des index dynamiques tout en gardant le principe d'association entre un prototype et les cas compatibles, et l'idée d'une modélisation statique du domaine. Par contre nous perdons les possibilités d'avoir de l'héritage multiple et d'exprimer des contraintes disjonctives.

[2] Toutefois, on pourrait envisager de mettre à jour la totalité ou une partie des hiérarchies lors du stockage.

La sélection

La sélection doit forcément se faire via une indexation par plus proches voisins, mais il faut modifier tous les mécanismes pour que cette technique soit applicable à notre représentation des cas (cas composites avec expression du flou).

Nous avons pensé dans un premier temps que notre problème était proche de celui des requêtes floues dans une base de données floues. Effectivement, dans ce domaine, il s'agit de répondre à une requête floue par un ensemble de tuples vérifiant plus ou moins la requête. Ces tuples peuvent alors être ordonnés suivant leur satisfaction à la requête. De notre côté, il s'agit de trouver et d'ordonner des cas composés de données floues suivant leur similarité avec la description floue d'un problème. Dans les deux cas, il y a bien un objet de référence (requête ou problème) et un ensemble d'entités (tuples ou cas). Il faut pourtant se garder de faire une analogie hâtive entre "similarité" et "satisfaction". En effet, dans la similarité, les deux objets à comparer sont de même nature, alors que dans la satisfaction il y a un objet générique et un objet particulier. Ce problème se retrouve dans les techniques employées : la requête et la donnée sont représentées par deux ensembles flous R et D, et on cherche à avoir R qui recouvre D, ce qui montre bien la différence de nature des deux entités. De plus, si l'intersection entre R et D est nulle, D est éliminé, alors que pour calculer une similarité, on aimerait avoir, quelque soit la configuration, une évaluation de la distance entre les deux entités. De toute manière, même pour l'interrogation floue, le problème du tri des tuples reste présent [DDT88]. En effet, la satisfaction est évaluée grâce aux coefficients classiques de possibilité et de nécessité, et il faut trier ces couples de valeurs. Dans ce but, on peut utiliser l'ordre de Pareto[3], mais il peut rester des indéterminations et dans ce cas il faut encore calculer d'autres coefficients (par exemple la "mesure d'intersection relative" [And92]).

Ainsi, pour proposer une solution à l'étape de sélection, il faut définir de nouvelles métriques de similarité. Le problème des métriques de similarité est encore très ouvert même sur une représentation simple des cas, et à notre connaissance, il n'existe aucun travail traitant de cette méthode d'indexation avec des cas composites ayant des caractéristiques floues.

4.3 Problématique de réalisation

La réalisation doit vérifier certaines contraintes, car elle doit permettre :

- la représentation des hiérarchies d'héritage des prototypes,
- la représentation des cas composites dont chaque sous-partie est instance d'un prototype,
- des modifications aisées des différents mécanismes mis en jeu car ils peuvent évoluer,
- le contrôle par l'utilisateur du déroulement des différents mécanismes (paramétrage du filtrage, de la sélection et de la recherche de manière globale, choix parmi des métriques de similarité prédéfinies...).

Ces différentes contraintes nous amènent naturellement à choisir une programmation par objet. Le projet SECOIA possède une version bêta de IlogTalk de ILOG [ILO93] qui est une version du langage LE_LISP intégrant une couche objet puissante inspirée de CLOS[4] : métaclasses, classes,

[3] $(a, b) > (c, d) \Leftrightarrow (a > c \text{ et } b \geq d) ou (a \geq c \text{ et } b > d)$

[4] Common Lisp Object System

méthodes génériques, protocole d'instanciation, protocole d'héritage. La version commerciale de IlogTalk n'a été disponible que fin août 1993 et en dehors de l'intérêt propre de la réalisation, il nous paraît important de tester le comportement et les possibilités offertes par ce langage.

Nous avons également à notre disposition le générateur de systèmes experts SMECI [ILO92a] et un module additionnel pour gérer les valeurs floues [ILO92b]. Cependant, selon M.F. Borderes [Bor90] SMECI n'est pas adapté pour faire de la classification floue surtout parce qu'il ne permet pas de définir de nouveaux types de données. Après avoir étudié ce que pourrait apporter le module de gestion du flou, nous nous alignons sur cette opinion. En effet, pour classer de manière floue un objet, il faut qu'il puisse appartenir à un prototype sans qu'il vérifie de manière stricte toutes les contraintes de typage. Comme il n'est pas possible de définir des types flous (même avec le module flou) nous sommes obligés de débrayer les vérifications de typage perdant ainsi un atout du modèle objet de SMECI. De plus, la création de facettes additionnelles qui puissent être redéfinissables dans des prototypes plus spécifiques n'est pas réalisable de manière élégante. Or nous avons besoin de ces facettes pour représenter les informations nécessaires à la classification.

Nous pouvons aussi utiliser la version de développement de CLASSIC et modifier certains mécanismes pour réaliser un outil de CBR. Cependant, la quantité de travail à fournir pour faire l'adaptation est difficile à évaluer. De plus, CLASSIC est disponible uniquement en Le_Lisp version 15, ce qui ne correspond pas à notre souhait de tester la version 16 (i.e. IlogTalk).

Nous choisissons donc d'effectuer la réalisation avec le langage IlogTalk. Ainsi, nous pourrons utiliser les classes et les métaclasses pour construire notre propre modèle objet de représentation des hiérarchies de prototypes et des cas, et les algorithmes seront implantés par des méthodes génériques.

4.4 Conclusion

Ainsi, nous avons vu que les différentes caractéristiques des problèmes de conception nous ont amenés à utiliser des hiérarchies de prototypes pour représenter des cas composites. Notre modélisation permet également de gérer d'une certaine manière les points de vue et de proposer des solutions sur différentes parties d'un problème de manière à le résoudre par assemblage. La gestion du flou dans les caractéristiques des cas permet de représenter les préférences et l'imprécis.

Pour la phase de recherche, nous avons ébauché les grandes lignes de l'indexation par classification floue qui est une nouvelle méthode d'indexation que nous proposons pour le filtrage. Pour cette indexation, l'organisation de la mémoire correspond aux hiérarchies de prototypes. Nous avons également décidé de réaliser la sélection par une indexation par plus proches voisins avec des métriques de similarité adaptées à la nature composite et floue des cas.

Nous avons enfin choisi d'utiliser le langage IlogTalk pour la réalisation.

Chapitre 5

Conception du module de recherche

Nous allons commencer par la conception de la représentation des cas et de l'organisation de la mémoire car nous avons vu que dans notre modèle ces deux parties étaient très liées. Puis nous aborderons la conception de la phase de recherche composée de l'indexation par classification floue (filtrage) et de l'indexation par plus proches voisins (sélection).

5.1 Représentation des cas et organisation de la mémoire

Nous avons choisi une représentation utilisant des hiérarchies de prototypes, et nous allons voir dans un premier temps comment les cas sont associés à ces différentes hiérarchies. Puis nous définirons exactement quelles sont les informations qu'un prototype doit fournir pour la représentation des cas et pour l'indexation par classification floue.

5.1.1 Hiérarchies de prototypes et cas

Un cas est représenté de manière composite dont chaque composant est instance d'un prototype qui lui-même est défini dans une hiérarchie de prototypes (cf figure 4.1). Chaque composant doit être associé à tous les prototypes les plus spécifiques avec lesquels il est compatible dans le sens de la classification floue. Cette association devrait être faite par l'étape de stockage de la phase d'apprentissage, mais pour pouvoir continuer la conception de notre système, nous supposerons que les cas sont correctement liés aux prototypes.

5.1.2 Description d'un prototype

Chaque prototype possède un ensemble de caractéristiques décrivant une famille de cas. Notre but est de définir différentes sortes de caractéristiques pour obtenir le meilleur système de représentation possible en nous basant sur l'étude de ReMind et de CLASSIC. Ces caractéristiques (ou attributs) sont identifiées par leur nom, et sont héritées dans les prototypes plus spécifiques. Ces caractéristiques sont décrites par des propriétés statiques et dynamiques. Les propriétés statiques sont définies dans les prototypes et sont communes à toutes les instances. Les propriétés dynamiques sont liées à chaque instance et peuvent donc avoir des valeurs différentes dans chacune d'entre-elles.

- Propriétés statiques :
  - **type de caractéristique** : il existe deux types de caractéristiques, celles qui sont des index potentiels (et qui pourront être utilisées comme index dynamiques ou statiques) et celles qui servent à compléter les informations sur les cas et qui ne seront jamais utilisées comme index. Nous avons fait cette distinction, car il est parfois possible de savoir a priori que certaines caractéristiques ne seront jamais utilisées comme index et que leur intérêt se limite à l'expression de la solution. Cette possibilité permet également de représenter une caractéristique qui ne peut pas être utilisée comme index soit parce qu'il n'existe pas de domaine correspondant, soit parce qu'il n'existe pas de métrique de similitude pour son domaine de valeurs. Par exemple, on déclarera le chemin d'accès à un fichier représentant une image comme une caractéristique qui ne peut pas servir d'index. Pourtant, cette image peut constituer un élément important pour la description de la solution.
  - **domaine**[1] : domaine de valeurs de la caractéristique (cf section 5.1.2).
  - **restriction**[1] : restriction floue du domaine (cf section 5.1.2). Cette propriété peut être spécialisée (i.e. redéfinie lors de l'héritage).
  - **valeur par défaut** : expression ou méthode capable de calculer la valeur (floue pour des index) de la caractéristique.
  - **index par défaut**[1] : expression booléenne ou méthode capable d'indiquer si la caractéristique doit être choisie comme index. Cette propriété est surtout utile pour indiquer les index statiques et pour sélectionner automatiquement des index dynamiques. En effet, si les index ne sont fixés que par la description du problème, il est possible que des caractéristiques représentant des buts implicites ne soient pas choisies comme index. Par exemple, si le problème consiste à concevoir une voiture à 3 portes, il ne faut pas oublier le but implicite que la voiture doit pouvoir rouler, démarrer...
  - **importance par défaut**[1] : expression ou méthode capable de calculer l'importance de la caractéristique (nombre entre 0 et 1). Cette propriété permet donc un calcul automatique de l'importance. Cette valeur n'a de sens que si la caractéristique est choisie comme index.
  - **valeurs symboliques possibles**[1] : permet de représenter des valeurs symboliques pour des caractéristiques définies sur un domaine numérique (cf section 5.1.2).
  - **valeur à demander** : expression booléenne ou méthode capable d'indiquer si la valeur est à demander à l'opérateur. Cette propriété sera utilisée lors de l'indexation par classification interactive. Cette valeur peut être calculée dynamiquement car il est possible que la demande de la valeur d'une caractéristique dépende des valeurs des autres caractéristiques.
  - **numéro de précédence** : donne l'ordre dans lequel les caractéristiques doivent être examinées au niveau de chaque prototype.
  - **texte de la question** : texte qui accompagne la demande de la valeur à l'opérateur.
  - **commentaires** : commentaires sur la sémantique du champ. Cette propriété peut être spécialisée.

[1] Cette propriété n'est présente que pour les index potentiels.

- Propriétés dynamiques :
  - **index**[1] : indique effectivement si la caractéristique a été choisie comme index.
  - **importance**[1] : importance effective.
  - **valeur** : la valeur effective de la caractéristique.

Par rapport à CLASSIC, nous avons enrichi les informations spécifiques à la classification : domaines plus nombreux (cf section 5.1.2), restriction plus générales (cf section 5.1.2 et 5.1.2), l'importance d'une caractéristique et la valeur de la propriété 'à demander' peuvent être calculées dynamiquement, l'ordre des questions peut être contrôlé par des numéros et le texte de la question peut être spécifié. Par contre, nous ne gérons pas les règles de production.

Dans ReMind, la possibilité d'avoir des valeurs par défaut calculées n'existe pas, ce qui a obligé les concepteurs des systèmes CBR de la SITA à créer des caractéristiques supplémentaires de type formule pour représenter chaque valeur par défaut. Par contre, ReMind utilise des modèles qualitatifs pour déterminer les importances des index. Nous avons prévu de gérer des importances calculées et il serait intéressant d'offrir les mêmes mécanismes (dans une extension).

Voyons maintenant plus en détail certaines propriétés : les domaines, les restrictions et les valeurs symboliques.

Domaines

Un domaine D représente un ensemble de valeurs. Nous avons défini des types de domaines de base :

- **réel** : $D = (D_{min}, D_{max}) \in I\!R \times I\!R$
- **entier** : $D = (D_{min}, D_{max}) \in I\!N \times I\!N$
- **liste mono-valuée** : $D = (\alpha | \alpha \text{ est une expression})$
  Nous noterons $|D|$ le cardinal de la liste. Ce domaine est muni également de la soustraction permettant de calculer le nombre d'éléments séparant deux éléments donnés. Ce type de domaine permet de définir un domaine symbolique ordonné, par exemple `(très-petit, petit, moyen, grand, très-grand)`.
- **ensemble mono-valué** : $D = \{\alpha | \alpha \text{ est une expression}\}$
  $|D|$ représente toujours le cardinal de l'ensemble. Nous pouvons ainsi définir le domaine booléen sur l'ensemble de valeurs $\{$`vrai,faux`$\}$.
- **hiérarchie de symboles** : $D = D_{racine}$
  Sur ce domaine, nous pouvons définir la fonction $pere$ qui donne le père d'un élément du domaine s'il existe, et $fils$ qui donne l'ensemble des fils d'un élément du domaine. La fonction $MSCA$ qui a déjà été définie (cf 3.3.3), est utilisable sur ce type de domaine.

  Par exemple, la figure 5.1 représente le domaine "cuisine française ".
- **hiérarchie de prototypes** : $D = D_{racine}$
  Nous utilisons ce type de domaine pour représenter le domaine d'une sous-partie. Les fonctions $pere$, $fils$ et $MSCA$ sont également valables sur ce type de domaine.

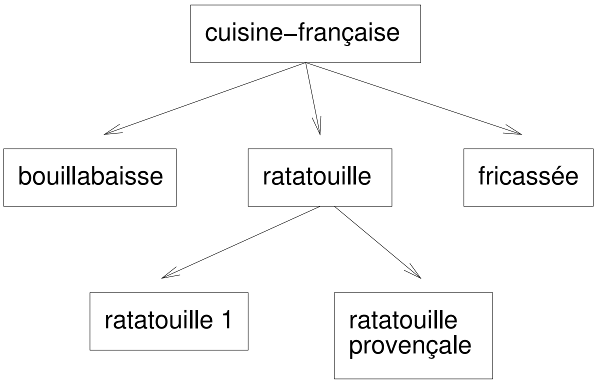

Figure 5.1 : Domaine "cuisine française "

Nous nous sommes limités à des domaines mono-valués, ainsi nous ne gérons pas un domaine équivalent au type *liste* de ReMind. Le domaine *hiérarchie de symboles* s'inspire de ReMind, toutefois, nous ne gérons pas les liens d'ordre entre les symboles. La notion de liste ordonnée n'est présente ni dans ReMind ni dans CLASSIC, et pourtant le domaine *liste* permet de ne pas perdre l'information que représente l'ordre entre les symboles.

Restriction

Une restriction permet de réduire l'ensemble des valeurs d'un domaine de manière floue grâce à un ensemble flou sur le domaine considéré (cf annexe A page 69 pour la définition d'un ensemble flou). Ces ensembles flous peuvent être vus comme des distributions de possibilités (cf annexe A page 73). L'idée de restriction floue n'existe pas dans ReMind et dans CLASSIC le flou est présent mais de manière limitée (cf annexe B page 78). Notre but est donc d'améliorer la représentation du flou.

Dans la suite, nous noterons : R_D l'ensemble des restrictions sur le domaine D, r une restriction particulière, μ_r la fonction caractéristique de la restriction r définie par :

$$\mu_r : D \longrightarrow [0,1]$$

et ω_r le support de la restriction r défini par :

$$\omega_r = \{x \in D | \mu_r(x) \neq 0\}$$

Pour une meilleure représentation, nous avons défini plusieurs types de restriction qui peuvent s'appliquer sur différents domaines. Il faut bien entendu qu'il existe au moins un type de restriction par type de domaine. Nous allons étudier chaque type de restriction, ensuite nous donnerons les domaines où les restrictions peuvent s'appliquer et nous détaillerons le procédé de calcul de la fonction μ_r.

- **trapèze linéaire** (figure 5.2) : cet ensemble flou est défini par deux bornes min et max, et les tolérances $tmin$ et $tmax$.

  Nous avons donc :

$$r = (r_{min}, r_{max}, r_{tmin}, r_{tmax}) \in D^4$$

$$\mu_r(v) = \begin{cases} 1 & \text{si } r_{min} \leq v \leq r_{max} \\ 1 - \frac{v - r_{max}}{r_{tmax} - r_{max}} & \text{si } r_{max} < v \leq r_{tmax} \\ \frac{v - rtmin}{r_{min} - r_{tmin}} & \text{si } r_{tmin} \leq v < r_{min} \\ 0 & \text{sinon} \end{cases}$$

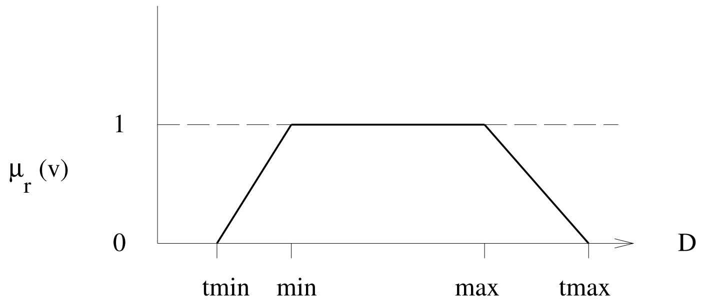

Figure 5.2 : Trapèze linéaire

Ce type de restriction ne peut être utilisé que dans des domaines ordonnés (entier, réel, liste). Normalement, les tolérances des ensembles flous représentent la distance admise à partir d'une borne intérieure du trapèze. Dans notre représentation, les tolérances sont les bornes extérieures elles-mêmes. Cette modification ne change rien à la généralité du modèle puisqu'il est possible de passer d'une représentation à l'autre. Par contre, il nous semble plus aisé de spécifier les bornes, surtout sur un domaine *liste de symboles.*

- **trapèze non linéaire** (figure 5.4) : nous proposons de généraliser les ensembles flous de type trapèze en utilisant la fonction f^λ avec $\lambda \in]-1, \infty[$ [ILO92b]:

$$f^\lambda(x) = \begin{cases} \frac{(1+\lambda)x}{1+2\lambda x} & 0 \leq x \leq \frac{1}{2} \\ \frac{1-(1-\lambda)(1-x)}{1+2\lambda(1-x)} & \frac{1}{2} \leq x \leq 1 \end{cases}$$

La figure 5.3 montre comment se comporte f^λ suivant les valeurs de λ ($\lambda \to -1$, $\lambda = 0$, $\lambda \to \infty$).

Ainsi ce type d'ensemble flou est défini par les paramètres de la fonction de tolérance λ_{min} et λ_{max} et par les caractéristiques des ensembles flous de type trapèze. L'ensemble flou de type trapèze linéaire est alors un cas particulier ($\lambda = 0$).

Nous avons donc :

$$r = (r_{min}, r_{max}, r_{tmin}, r_{tmax}, r_{\lambda_{min}}, r_{\lambda_{max}}) \in D^4 \times]-1, \infty[^2$$

$$\mu_r(v) = \begin{cases} 1 & \text{si } r_{min} \leq v \leq r_{max} \\ 1 - f^{\lambda_{max}}(\frac{v - r_{max}}{r_{tmax} - r_{max}}) & \text{si } r_{max} < v \leq r_{tmax} \\ f^{\lambda_{min}}(\frac{v - r_{min}}{r_{min} - r_{tmin}}) & \text{si } r_{tmin} \leq v < r_{min} \\ 0 & \text{sinon} \end{cases}$$

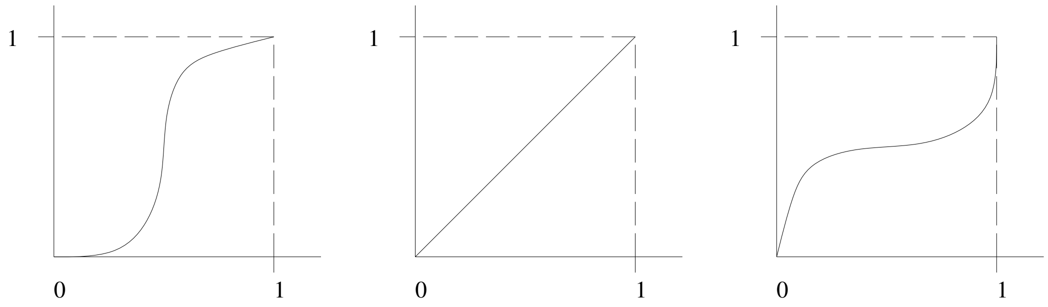

Figure 5.3 : Différentes formes de la fonction f^{λ}

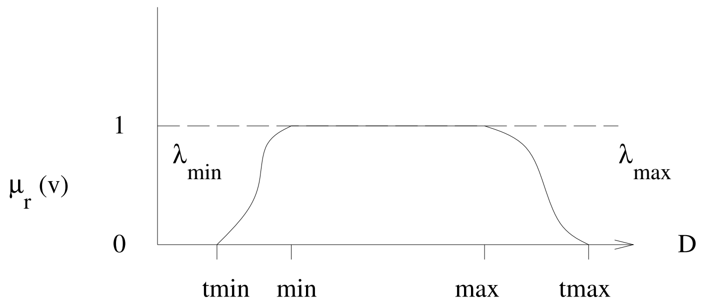

Figure 5.4 : Trapèze non-linéaire

- **ensemble flou discret** : ce type d'ensemble flou convient à tous les domaines. En effet il permet d'associer à n'importe quel élément du domaine sa possibilité. Dans ce cas :

$$r = \{(\alpha, \alpha_\tau) | \alpha \in D \text{ et } \alpha_\tau \in [0,1]\}$$

Pour les domaines non hiérarchiques :

$$\mu_r(v) = \begin{cases} \alpha_\tau & \text{si } \exists \alpha \text{ tel que } \alpha = v \\ 0 & \text{sinon} \end{cases}$$

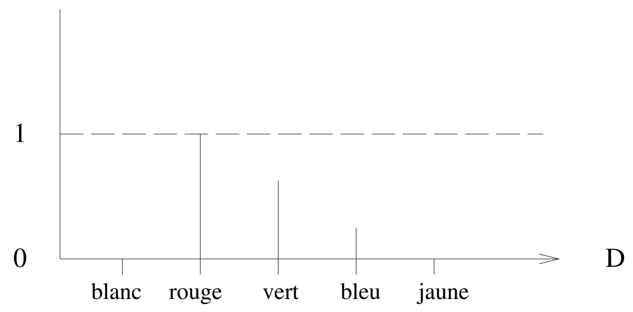

Figure 5.5 : Ensemble flou discret sur un domaine symbolique

Par exemple (figure 5.5), nous pouvons avoir l'ensemble flou sur le domaine {blanc, rouge, vert, bleu, jaune} défini par {(rouge,1), (vert,0.6), (bleu,0.4)}.

Nous proposons une méthode de calcul pour les domaines hiérarchiques :

$$\mu_r(v) = \begin{cases} \alpha_\tau & \text{si } \exists \alpha \text{ tel que } \alpha = v \\ \mu_r(pere(v)) & \text{si } pere(v) \text{ existe et } \forall \alpha \in \omega_r, \alpha \neq v \\ 0 & \text{sinon} \end{cases}$$

Dans notre exemple (figure 5.6) nous avons:

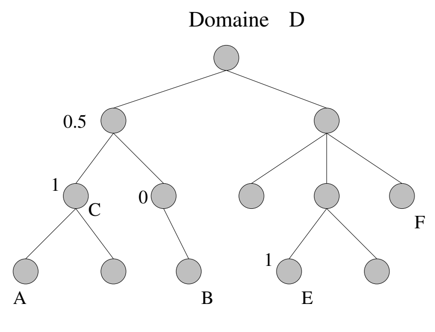

Figure 5.6 : ensemble flou sur domaine hiérarchique

$$\mu_r(A) = 1,\ \mu_r(B) = 0,\ \mu_r(C) = 1,\ \mu_r(E) = 1,\ \mu_r(F) = 0$$

- **fonction discrète interpolée**: nous proposons d'interpoler un ensemble flou discret sur un domaine ordonné (entier, réel et liste).

$$\mu_r(v) = \Gamma_r(v)$$

Avec Γ_r la fonction d'interpolation.

Par exemple, nous pouvons lier les points par des segments (figure 5.7).

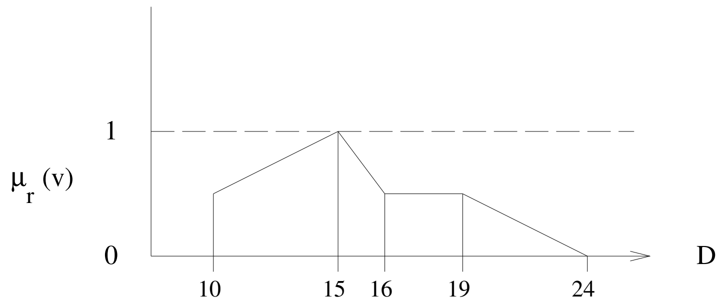

Figure 5.7 : ensemble flou discret interpolé

- **superposition d'ensembles flous** : un ensemble flou peut être lui-même composé d'ensembles flous. La nouvelle fonction caractéristique représente le maximum de toutes les composantes sur tout le domaine.

$$r = \{\beta | \beta \in R_D\}$$

$$\mu_r(v) = \max_i(\mu_{\beta_i}(v))$$

Valeurs symboliques

Nous proposons d'associer à une caractéristique définie sur un domaine numérique une liste de symboles (cf figure 5.8). A chaque symbole correspond un ensemble flou sur le domaine de la caractéristique. Cette possibilité est intéressante lors de la description du problème. En effet, l'utilisateur préférera souvent répondre de manière symbolique plutôt que de spécifier lui-même un ensemble flou. Il pourra ainsi choisir une réponse parmi les symboles prédéfinis. Par exemple, à la question "Quelle est la taille du livre?", on pourra répondre *grand*. Bien sûr, on aurait pu choisir un domaine symbolique directement, mais dans ce cas, il y a une perte d'information. Or cette information sera utilisée dans les calculs de la compatibilité et de l'incompatibilité lors de l'indexation par classification floue et dans les mesures de similarité de l'indexation par plus proches voisins.

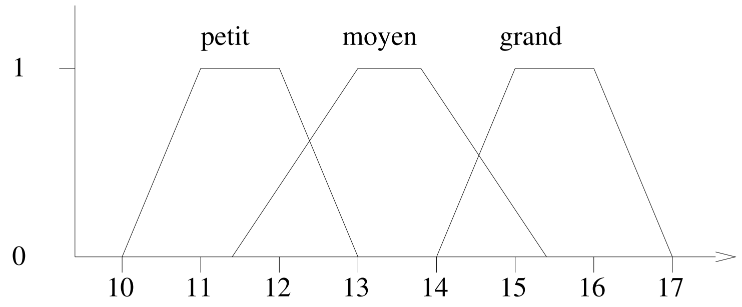

Figure 5.8 : valeurs symboliques floues

5.2 Recherche des cas similaires

Nous allons d'abord définir quels sont les paramètres de l'étape de recherche puis nous présenterons la conception de l'indexation par classification floue et de l'indexation par plus proches voisins.

5.2.1 Contrôle de la recherche

L'utilisateur doit pouvoir contrôler l'étape de recherche et ainsi choisir les méthodes d'indexation qui seront effectuées. Nous permettons donc de choisir : soit l'indexation par classification floue, soit l'indexation par plus proches voisins soit la combinaison des deux.

5.2.2 Indexation par classification floue

Dans un premier temps nous allons décrire l'algorithme de l'indexation par classification floue puis nous définirons les paramètres qui permettent de contrôler cette indexation.

Algorithme d'indexation par classification floue

L'algorithme d'indexation par classification floue prend en entrée la hiérarchie mère de la description du problème. Notre modèle permet effectivement de choisir n'importe quelle hiérarchie ainsi le problème peut porter sur n'importe quel objet et n'importe quel point de vue. Le résultat est un ensemble de cas.

Nous pouvons distinguer deux grandes étapes : la classification du problème dans la hiérarchie de prototypes, puis la récupération des cas.

1. **Classification** :
   Pour la classification nous nous basons grandement sur l'algorithme de raffinage descendant de CLASSIC qui utilise des coefficients de compatibilité et d'incompatibilité (cf annexe B page 77). Ainsi, nous partons de la racine de la hiérarchie qui a été choisie et nous essayons de classer le problème dans un ou plusieurs prototypes.

   A la différence de CLASSIC, le résultat est un ensemble flou discret sur le domaine de la hiérarchie. La possibilité associée à chaque prototype du domaine représente la synthèse de sa compatibilité et de son incompatibilité avec la description du problème. Nous voulons que le résultat soit un ensemble flou, car lors du classement d'une sous partie, cet ensemble sera la valeur de la caractéristique désignant la sous-partie. Ainsi notre modèle de représentation des caractéristiques est uniforme.

Un algorithme est exécuté à chaque nouveau prototype à examiner, mais les étapes sont légèrement différentes de celles de CLASSIC car nous avons rajouté la notion d'index dynamiques et nous ne gérons pas les règles de production. La figure 5.9 représente les étapes de cet algorithme.

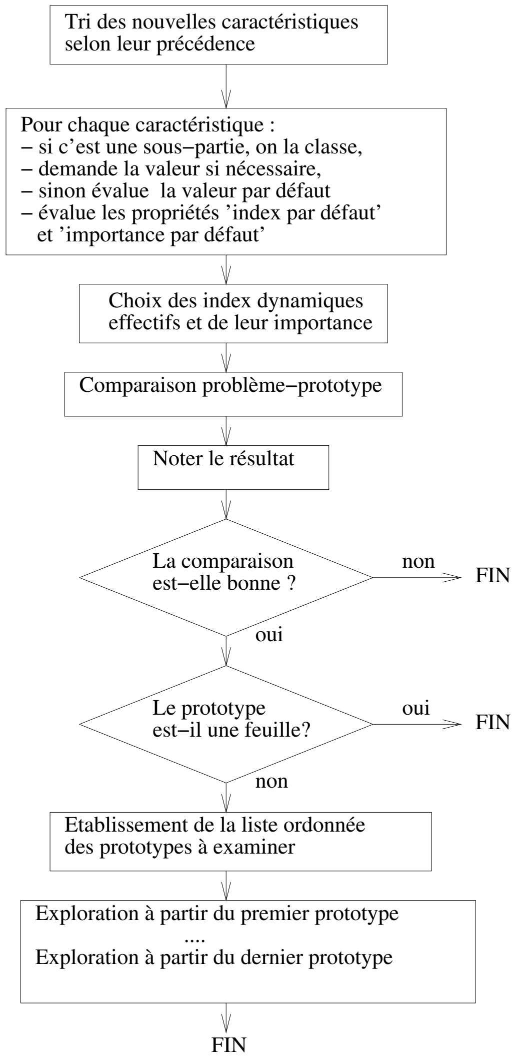

Figure 5.9 : Algorithme effectué à chaque prototype

(a) Les nouvelles caractéristiques du prototype sont triées suivant leur ordre de précédence.

(b) De manière à compléter la description du problème, chaque caractéristique est examinée :

i. si la caractéristique est une sous-partie et qu'elle possède la propriété 'à demander', le système lance récursivement la classification sur la hiérarchie considérée, le résultat de la classification est alors interprété comme un ensemble flou sur un domaine *hiérarchie de prototypes*,

ii. sinon si la caractéristique possède la propriété 'à demander', le système demande la valeur,

iii. sinon si la valeur par défaut existe, la valeur de la caractéristique devient la valeur par défaut et reste inconnue dans le cas contraire,

iv. les propriétés 'index par défaut' et 'importance par défaut' sont évaluées et affectées aux propriétés 'index' et 'importance'.

(c) Les index dynamiques pour le prototype courant sont alors choisis définitivement.

(d) Le système compare le problème et le prototype en calculant pour chaque index dynamique, les degrés de compatibilité et d'incompatibilité. Nous prenons comme convention que chaque caractéristique a une restriction de prototype p et une restriction de valeurs r sur le domaine D. Si p et/ou r sont inconnues, les deux coefficients sont égaux à 1, sinon nous utilisons les formules suivantes:

$$\begin{aligned} \text{compatibilité}(p,r) &= \prod(p|r) \\ &= \max_{v\in D}(\min(\mu_p(v),\mu_r(v))) \end{aligned}$$

$$\begin{aligned} \text{incompatibilité}(p,r) &= 1 - N(p|r) \\ &= 1 - \min_{v\in D}(\max(\mu_p(v), 1-\mu_r(v))) \\ &= \max_{v\in D}(\min(1-\mu_p(v),\mu_r(v))) \end{aligned}$$

$\prod$ et N représentent respectivement les coefficients de possibilité et de nécessité définis dans l'annexe A page 73.

(e) Nous calculons ensuite la compatibilité et l'incompatibilité globale à partir des coefficients sur chaque index dynamique C_i du prototype P :

$$\begin{aligned} \text{compatibilité}_P = &\quad \min_i(\max(\text{compatibilité}(C_i),\ \text{1-importance}(C_i))) \\ \text{incompatibilité}_P = &\quad \max_i(\min(\text{incompatibilité}(C_i),\ \text{importance}(C_i))) \end{aligned}$$

Puis nous combinons avec les coefficients de compatibilité et d'incompatibilité hérités :

$$\begin{aligned} \text{compatibilité} = &\quad \min(\text{compatibilité}_{pere(P)}, \text{compatibilité}_P) \\ \text{incompatibilité} = &\quad \max(\text{incompatibilité}_{pere(P)}, \text{incompatibilité}_P) \end{aligned}$$

(f) D'après les seuils de compatibilité et d'incompatibilité nous pouvons évaluer si le résultat de la comparaison est acceptable (cf annexe B page 78).

(g) Si la comparaison est acceptable, les fils du prototype courant s'ils existent sont ajoutés à la liste des prototypes à examiner. L'instance courante est également considérée comme une solution et doit être intégrée à l'ensemble flou qui sera le résultat de la classification. Pour calculer sa possibilité, nous synthétisons les coefficients de compatibilité (cp) et d'incompatibilité (icp), et dans ce but nous proposons d'utiliser au choix une des deux formules prédéfinies :

$$\Delta_1(cp, icp) = 1 - \sqrt{\frac{icp^2 + (1-cp)^2}{2}}$$

$$\Delta_2(cp, icp) = \frac{cp + (1 - icp)}{2}$$

Δ_1 correspond à la distance du couple (cp,icp) au couple (1,0) – qui représente la meilleure satisfaction possible – ramenée à [0,1] de manière à ce que $\Delta_1 = 1$ lorsque la distance est nulle et que $\Delta_1 = 0$ lorsque la distance est de $\sqrt{2}$. Δ_2 correspond au degré de satisfaction moyen. Toutefois, l'instance du prototype père du prototype courant qui se trouve forcément dans l'ensemble flou des solutions, doit être enlevé de cet ensemble de manière à obtenir un ensemble contenant uniquement les feuilles de la classification.

(h) Les prototypes sont alors examinés dans l'ordre défini par la liste des prototypes.

2. **Récupération des cas** :
Les cas associés à tous les prototypes éléments du support de l'ensemble flou résultant de la classification, ainsi que les cas associés à tous leurs ancêtres sont retournés.

Contrôle de l'indexation par classification floue

Le contrôle du processus de l'indexation par classification floue doit permettre de modifier le comportement standard. Pour la classification en elle-même, nous gardons tous les mécanismes de contrôle de CLASSIC (cf annexe B page 79), mais nous ajoutons également la possibilité d'effectuer le filtrage de manière non interactive. Dans ce cas, les valeurs, les index et leur importance seront déterminés par une description préalable du problème.

5.2.3 Indexation par plus proches voisins

Le problème de l'indexation par plus proches voisins réside dans la recherche de métriques de similarité. Nous allons essayer de mettre en évidence les propriétés des métriques puis nous proposerons un ensemble de métriques applicables à notre représentation des cas. Nous détaillerons alors l'algorithme d'indexation par plus proche voisins et les paramètres de contrôle nécessaires.

Propriétés des métriques de similarité

Nous rappelons certaines définitions mathématiques [Sap90]. Soit E un ensemble, une *distance* est une application de $E \times E$ dans $\mathbb{R}^+$ telle que :

$$\begin{cases} d(i,j) = d(j,i) \\ d(i,j) \geq 0 \\ d(i,j) = 0 \Leftrightarrow i = j \\ d(i,j) \leq d(i,k) + d(k,j) \end{cases}$$

Une *dissimilarité* est une application de $E \times E$ dans $I\!R^+$ telle que :

$$\begin{cases} d(i,j) = d(j,i) \\ d(i,j) \geq 0 \\ d(i,i) = 0 \end{cases}$$

Et enfin une *similarité* est une application de $E \times E$ dans $I\!R^+$ telle que :

$$\begin{cases} s(i,j) = s(j,i) \\ s(i,j) \geq 0 \\ s(i,i) \geq s(i,j) \end{cases}$$

Dans un contexte CBR, nous allons essayer de formaliser et de justifier les propriétés d'une métrique de similarité.

Tous d'abord, comme les mesures de similarité sur les différentes caractéristiques doivent être agrégées selon leur importance, il faut que ces mesures soient normalisées c'est-à-dire ramenées à un intervalle borné. Nous choisissons classiquement l'intervalle [0,1]. Il faut ensuite définir la valeur de la similarité entre un élément et lui-même. Dans les travaux de CBR, la similarité d'un élément avec lui-même est toujours égal à 1. Nous proposons alors la définition de la *cbr-similarité* qui est une application de $E \times E$ dans [0,1]:

$$\begin{cases} sim(i,j) = sim(j,i) \\ sim(i,i) = 1 \end{cases}$$

Une *cbr-similarité* est donc une *similarité* avec en plus la propriété $s(i,i) = 1$.

Proposition de métriques de similarité

Nous allons choisir et proposer des métriques pour évaluer la similarité entre deux éléments de chaque domaine. Mais dans notre modèle, les valeurs de caractéristiques sont des restrictions (ensembles flous); nous proposerons alors une mesure entre deux restrictions.

Pour les métriques entre deux éléments x et y d'un domaine, nous définissons la fonction δ qui dépend du domaine :

- Domaines numériques :
$$\delta(x,y) = 1 - \frac{|x-y|}{D_{max} - D_{min}}$$
- Domaine liste :
$$\delta(x,y) = 1 - \frac{|x-y|}{|D|}$$
- Domaine ensemble :
$$\delta(x,y) = 1 \text{ si } x = y, \ 0 \text{ sinon}$$

- Domaine hiérarchie de symboles :

$$\delta(x,y) = 1 - \frac{\psi(MSCA(x,y),x) + \psi(MSCA(x,y),y)}{\psi(D_{racine},x) + \psi(D_{racine},y) + 2}$$

  Cette mesure s'inspire de la mesure de ReMind pour les caractéristiques de type symbole (cf section 3.3.3). Toutefois, la mesure de ReMind présente un petit défaut, car, dès que $MSCA(x,y) = D_{racine}$, nous avons $\delta(x,y) = 0$. Il est ainsi impossible de déterminer parmi y, z l'élément le plus proche de x si $MSCA(x,y) = MSCA(x,z) = D_{racine}$. Pour corriger cet effet, nous avons ajouté un *2* au dénominateur et ainsi tout se passe comme s'il existait une racine virtuelle du domaine qui est le père de la racine effective du domaine.

- Domaine hiérarchie de prototypes :

$$\delta(x,y) = \min(\delta(\tilde{x},\tilde{y}), \frac{\sum_{c\in\Omega_{x,y}} p(c) * \theta(c_x,c_y)}{\sum_{c\in\Omega_{x,y}} p(c)})$$

  x et y sont des instances appartenant à la même hiérarchie de prototypes, $\tilde{x}$ et $\tilde{y}$ représentent les noms des prototypes dont x et y sont les instances. $\Omega_{x,y}$ est l'ensemble des caractéristiques communes et connues de x et y. Si c est un élément de Ω, c_x et $p(c)$ représentent respectivement la valeur et l'importance de la caractéristique c de l'objet x. c_x est une restriction (ensemble flou). θ est une *cbr-similarité* qui permet de comparer deux ensembles flous (cf section suivante).

  $\delta(\tilde{x},\tilde{y})$ représente la similarité entre les prototypes de ces objets en considérant la hiérarchie des prototypes comme une hiérarchie de symbole.

  $\frac{\sum\limits_{c\in\Omega_{x,y}} p(c)*\theta(c_x,c_y)}{\sum\limits_{c\in\Omega_{x,y}} p(c)}$ est la formule d'agrégation classique.

  Nous aurions pu utiliser uniquement la formule d'agrégation mais la fonction que nous proposons améliore le calcul de la similarité. En effet, la formule d'agrégation ne prend en compte que les caractéristiques communes et connues, et de notre côté, nous prenons en compte également les différences entre les objets en utilisant la hiérarchie des prototypes.

Ces différentes mesures sont des *cbr-similarités.*

Pour le calcul de la similarité entre deux restrictions qui sont des ensembles flous, nous proposons la formule suivante :

$$\theta'(x,y) = \frac{\sum\limits_{\alpha\in\omega_x,\ \beta\in\omega_y} \delta(\alpha,\beta) * \mu_x(\alpha) * \mu_y(\beta)}{\sum\limits_{\alpha\in\omega_x,\ \beta\in\omega_y} \mu_x(\alpha) * \mu_y(\beta)}$$

$$\theta(x,y) = A(x,y) * \theta'(x,y)$$

Avec la fonction de normalisation :

$$A(x,y) = \frac{1}{\max(\theta'(x,x), \theta'(y,y))}$$

θ' calcule la moyenne des cbr-similarités entre chaque élément des deux ensembles flous. Cette moyenne est pondérée par le produit des valeurs des fonctions caractéristiques. θ' est une *similarité* et pour obtenir une *cbr-similarité* nous devons multiplier θ' par une fonction de normalisation A.

Nous aurions pu utiliser des formules issues de l'analyse de données, mais elles mettent souvent en jeu des calculs de barycentres, or nous avons défini des domaines comme les hiérarchies de symboles où il n'existe pas de notion de barycentre.

Algorithme d'indexation par plus proches voisins

L'algorithme prend en entrée un ensemble flou représentant la description du problème et un ensemble de cas.

1. Il faut évaluer la similarité entre la description du problème et chaque cas. Dans notre cas, il s'agit d'évaluer la fonction θ avec l'ensemble flou de la description du problème et l'ensemble flou exact constitué par le cas.

2. Les cas sont alors triés.

3. Suivant le nombre maximum de cas à rechercher, le système retourne un sous ensemble de la liste des cas.

Contrôle de l'indexation par plus proches voisins

Le contrôle permet de spécifier le nombre maximum de cas qui doivent être sélectionnés. Il est également intéressant de pouvoir changer de métriques.

5.3 Conclusion

Nous avons défini très précisément les structures employées pour la représentation des cas et l'organisation de la mémoire.

Nous avons également détaillé les différents processus ainsi que leurs paramètres de contrôle pour la recherche, pour l'indexation par classification floue et pour l'indexation par plus proches voisins. Nous avons aussi proposé de nouvelles métriques de similarité adaptées à une représentation composite et floue des cas.

Chapitre 6

Réalisation du module de recherche

Nous avons choisi d'utiliser la couche objet d'IlogTalk pour la réalisation de notre système dans le but de permettre de manière générale l'évolution du système. L'annexe C comprend une introduction des mécanismes de la couche objet d'IlogTalk. En effet, il faut être capable d'ajouter de nouvelles propriétés pour les caractéristiques des prototypes, ou de modifier le comportement de l'indexation. Nous allons présenter l'architecture du système, puis tout en essayant de montrer sa stabilité, nous allons détailler d'une part la réalisation de la représentation de cas et de l'organisation de la mémoire, et d'autre part la réalisation du module de recherche. Pour plus d'informations sur les classes et les fonctions génériques du système que nous allons introduire, veuillez vous reporter à l'annexe D.

6.1 Architecture du système

La figure 6.1 présente l'architecture globale de notre système.

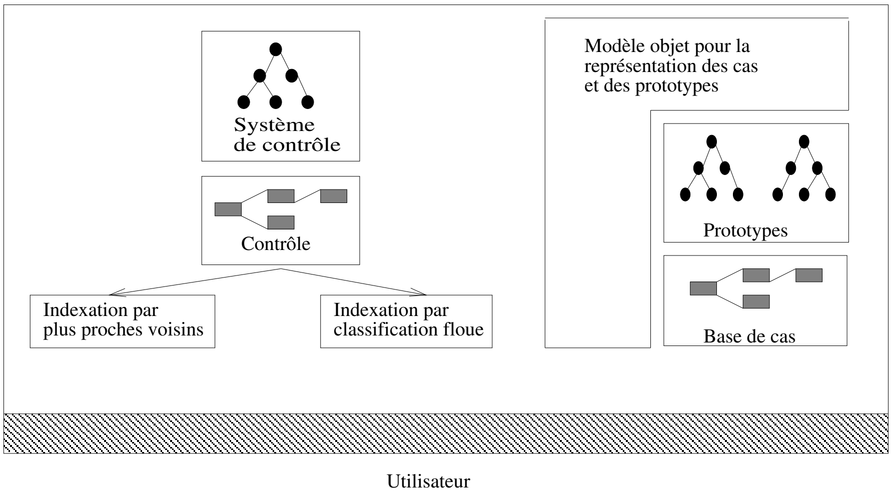

Figure 6.1 : Architecture du système.

Notre système se compose premièrement d'un modèle objet permettant la représentation des hiérarchies de prototypes et des cas.

D'autre part, nous avons également un modèle objet pour contrôler la phase de recherche. Ainsi des instances particulières des classes de contrôle permettent de paramétrer les différentes méthodes d'indexation par classification floue et par plus proches voisins.

6.2 Représentation des cas et organisation de la mémoire

Nous allons décrire dans un premier temps les hiérarchies de prototypes et les cas, puis nous verrons la structure exacte des prototypes. Les différentes explications qui vont suivre se reportent à la figure 6.2.

6.2.1 Hiérarchies de prototypes et cas

Nous avons défini qu'un prototype est composé de caractéristiques qui sont décrites par des propriétés. En IlogTalk, nous allons naturellement représenter un prototype par une classe et les caractéristiques par des slots (ou champs). Ainsi, nous pouvons utiliser les mécanismes d'héritage.

Les différentes hiérarchies de prototypes sont réunies sous le prototype `CASE` pour plus de clarté et pour typer les instances des prototypes. Si des informations communes à tous les cas devaient être utilisées, elles doivent être définies dans cette classe.

De plus, nous avons défini la métaclasse `PROTOTYPE` de manière à avoir un comportement spécifique des prototypes. Ainsi, grâce au slot `CASE-LIST` de cette métaclasse, nous pouvons associer aux différents prototypes une liste d'instances particulières qui représentent les cas de la mémoire qui seront accessibles par l'indexation par classification hiérarchique. Si d'autres informations spécifiques aux prototypes étaient nécessaires, il faudrait les ajouter dans cette métaclasse.

La création d'un prototype se fait grâce à la macro `DEFPROTOTYPE`, et la gestion de la liste des cas associée à un prototype se fait avec les méthodes génériques `ADD-CASE-TO-PROTOTYPE`, `REMOVE-CASE-FROM-PROTOTYPE`, `CASE-LIST` et `SET-CASE-LIST`.

6.2.2 Description d'un prototype

Un prototype est donc une classe avec des slots, il reste à représenter les propriétés ainsi que leurs valeurs particulières que sont les domaines, les restrictions et les valeurs symboliques.

Les propriétés

Le problème réside dans la représentation de deux types de propriété :

- **les propriétés statiques :**
  Nous utilisons les possibilités offertes par IlogTalk pour définir de nouveaux types de slots auxquels peuvent être associées des *facettes*. En effet, un type de slot est représenté par une classe et les slots de cette classe sont alors les facettes. Il existe deux types de slots prédéfinis auxquels sont associés des comportements différents : `LOCAL-SLOT-DESCRIPTION` pour les slots d'instances et `SHARED-SLOT-DESCRIPTION` pour les slots partagés par les instances. Pour les cas, nous ne voulons pas que les caractéristiques soient partagées, donc nous allons spécialiser la classe `LOCAL-SLOT-DESCRIPTION` pour créer deux nouvelles classes `FIELD`

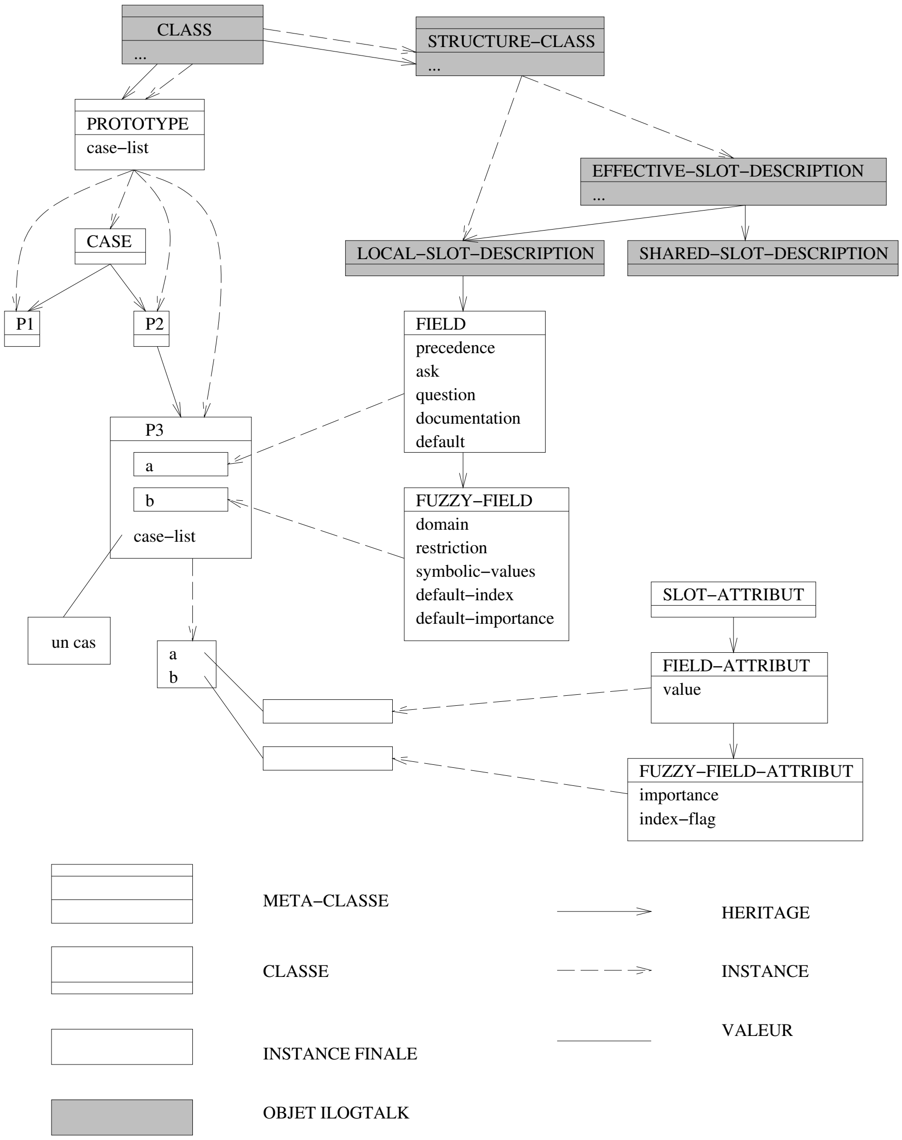

Figure 6.2 : Les prototypes, les facettes, et les attributs.

et `FUZZY-FIELD` correspondant aux deux types de caractéristique : caractéristiques normales (non-index) et caractéristiques à valeurs floues (index potentiels). Ainsi la propriété 'type de caractéristique' n'est pas représentée comme une véritable facette, mais l'information de cette propriété tient dans le type du slot. Les autres propriétés sont réellement des facettes et sont décrites dans les classes de slots. Le mécanisme d'héritage de ces facettes a été réalisé grâce à la spécialisation du mécanisme d'héritage standard (notamment de la méthode générique `COMPUTE-EFFECTIVE-SLOT`).

- **les propriétés dynamiques :**

  Les propriétés dynamiques sont liées aux instances; il s'agit donc d'*attribut de slots*. Les attributs de slots n'existent pas de manière standard dans Ilogtalk, mais nous allons adapter les mécanismes pour les réaliser. Ainsi nous spécialisons la méthode générique `INITIALIZE-INSTANCE` pour les objets de type `CASE` pour que l'instanciation d'un prototype crée automatiquement des objets contenant les propriétés dynamiques. Ces objets sont les instances de classes qui dépendent du type de la caractéristique (`FIELD-ATTRIBUT` ou `FUZZY-FIELD-ATTRIBUT`), et ils sont automatiquement affectés comme valeurs des slots de l'instance du prototype. En fait, lors de l'instanciation, la méthode générique `MAKE-SLOT-ATTRIBUT` que nous avons définie, est appelée avec comme paramètre le type du slot et elle retourne alors l'objet à affecter.

Ces mécanismes sont alors rendus invisibles par des méthodes d'accès aux différentes propriétés. De plus, grâce à notre représentation utilisant des hiérarchies de classes et à la méthode générique `MAKE-SLOT-ATTRIBUT`, l'ajout de nouvelles propriétés se fait facilement.

Les domaines

Il faut pouvoir non seulement représenter les domaines que nous avons définis avec leurs opérations mais également offrir la possibilité de créer de nouveaux types de domaines. Dans ce but, nous allons utiliser une hiérarchie de classes où chaque classe représente un type de domaine ou une abstraction de type de domaine (cf figure 6.3) .

Chaque domaine possède alors des attributs hérités et locaux qui permettent de le décrire. Ainsi, le domaine ensemble est défini par une liste de symboles, les domaines numériques possèdent une borne supérieure et une borne inférieure qui sont donc héritées pour les domaines des réels et des entiers. Le domaine des réels utilise également un pas pour permettre une discrétisation. Le domaine liste est, comme le domaine ensemble défini par une liste de symboles. Les domaines hiérarchiques sont définis par leur racine. Nous avons choisi de représenter les hiérarchies de symboles par une hiérarchie de classes de manière à utiliser les mécanismes existants sur les classes.

Nous donnons quelques exemples de création de domaines :

```
(defconstant *integer* (make-integer-domain (smallest-fix) (biggest-fix)))
(defconstant *real-interval* (make-real-domain 0 30))
(defconstant *boolean* (make-set-domain '(true false)))
(defconstant *size* (make-list-domain '(XS S M L XL XXL)))

(defsymbol cuisine-francaise ())
```

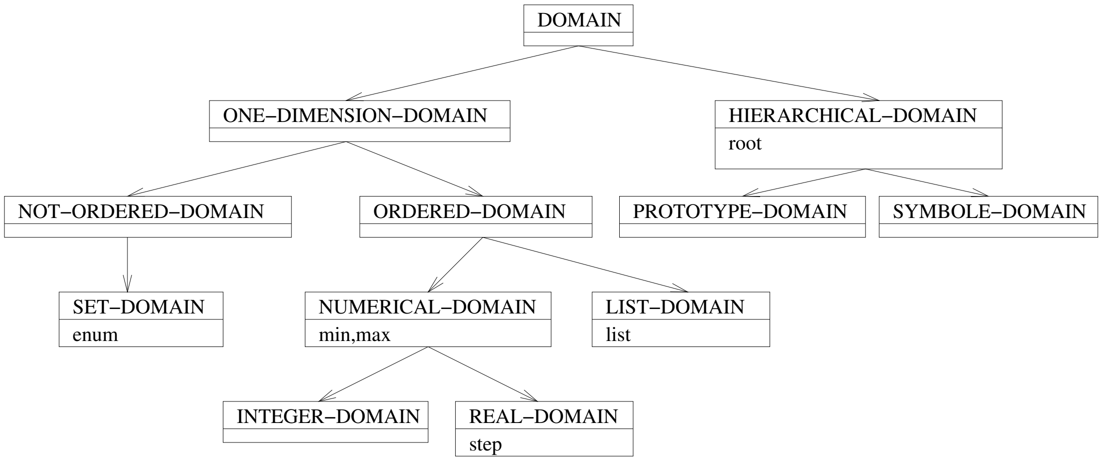

Figure 6.3 : Hiérarchie de domaines.

```
(defsymbol ratatouille (cuisine-francaise))
(defsymbol bouillabaisse (cuisine-francaise))
(defsymbol fricassee (cuisine-francaise))
(defsymbol rataouille-provencale  (ratatouille))
(defsymbol rataouille1 (ratatouille))

(defconstant *cuisine-francaise* (make-symbol-domain 'cuisine-francaise))
```

La hiérarchisation des domaines permet d'une part d'étendre par spécialisation les différents types existants et d'autre part de définir des opérations génériques indépendantes de la représentation physique. Ces opérations sont représentées par des méthodes génériques sur les classes de domaines et peuvent être spécialisées sur de nouveaux domaines si nécessaire.

Par exemple, pour les domaines ordonnés, nous avons les méthodes génériques `MINUS` pour faire une soustraction, `LOWER-THAN` (et `LOWER-OR-EQUAL`) pour savoir si une valeur est plus petite (ou égale) qu'une autre. Sur les domaines hiérarchiques, `FATHER` donne le père d'un élément et `SONS` les fils, `SUB-NODE-P` indique si un élément est plus spécifique qu'un autre, `MSCA` définit la fonction $MSCA$ (cf section 3.3.3), et `NUMBER-OF-LINKS` donne le nombre d'arcs entre deux éléments. Pour tous les domaines, nous avons `IS-ELEMENT` pour savoir si une expression est élément du domaine et `EQUAL-VALUE` qui indique si deux valeurs sont égales.

Les restrictions

Dans le même but que pour les domaines, nous allons représenter les restrictions dans une hiérarchie de classes (cf figure 6.4).

Chaque classe contient les différents paramètres des ensembles flous considérés. Nous donnons quelques exemples de création de restrictions.

```
(defconstant *big* (make-interpoleted-enum-fuzzy-set '((M . 0.4) (XL. 1)
                                                       (XXL . 1))))
```

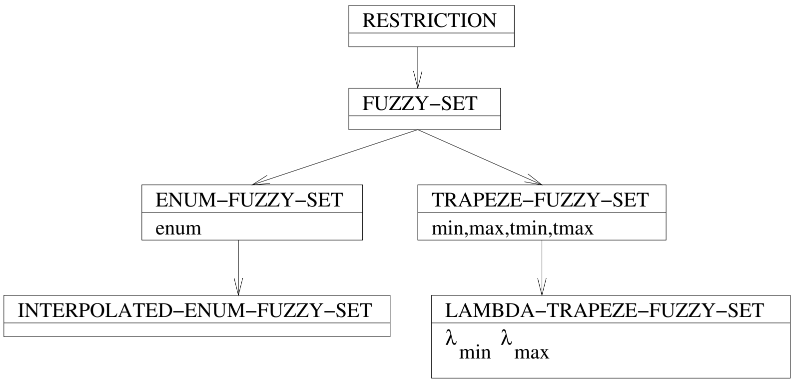

Figure 6.4 : Hiérarchie de restrictions.

```
(defconstant *r1* (make-trapeze-fuzzy-set 2 3 5 7))
(defconstant *r2* (make-trapeze-fuzzy-set 5 7 10 12))
(defconstant *ratatouille* (make-enum-fuzzy-set '((ratatouille . 1))))
```

La fonction caractéristique de chaque type de restriction est calculée par une méthode générique RESTRICTION-VALUE qui prend en compte une restriction, un domaine particulier, et une expression élément du domaine. Ainsi pour ajouter un nouveau type de restriction, il suffit de spécialiser une classe existante et de spécialiser la fonction générique RESTRICTION-VALUE pour les domaines correspondants si nécessaire.

Voici quelques exemples de calcul de la fonction caractéristique d'une restriction:

```
(RESTRICTION-VALUE *size* *grand* 'L)
0.7
(RESTRICTION-VALUE *real-interval* *r1* 10)
0
(RESTRICTION-VALUE *cuisine-francaise* *ratatouille* 'ratatouille1)
1
```

Valeurs symboliques

Nous avons défini la classe SYMBOL-VALUES pour représenter les valeurs symboliques. Les fonctions génériques SYMBOL-LIST, SET-SYMBOL-LIST , REMOVE-SYMBOL-VALUE, SET-SYMBOL-RESTRICTION, GET-SYMBOLS permettent de gérer la liste d'association entre un symbole et une restriction. Voici un exemple de création d'un objet représentant les valeurs symboliques :

```
(defconstant *symbol-values*
             (make-symbol-values `((normal . ,*r1*) (grand . ,*r2*))))
```

6.2.3 Exemple

Nous pouvons maintenant présenter un exemple de création de prototypes:

```
(defprototype piece ()
  ((longueur slot-class fuzzy-value
             slot-initargs (domain *real-interval*
                             ask t
                             question "longueur de la piece ?"
                             precedence 1
                             default *r1*
                             symbolic-values *symbol-values*
                             documentation "longueur de la piece"))
   (largeur slot-class fuzzy-value
            slot-initargs (domain *real-interval*
                            ask t
                            question "largeur de la piece ?"
                            precedence 2
                            documentation "largeur de la piece"))))

(defprototype petite-piece (piece)
  ((longueur slot-initargs (retriction (make-trapeze-fuzzy-set 0 0 3 4)))
   (largeur  slot-initargs (retriction (make-trapeze-fuzzy-set 0 0 3 4)))))
```

6.3 Module objet pour la phase de recherche

Nous allons présenter la réalisation du contrôle de la phase de recherche, puis les protocoles permettant d'implanter les algorithmes d'indexation par classification (filtrage) et par plus proches voisins (sélection).

6.3.1 Classes de contrôle

Nous devons représenter les informations qui permettent de paramétrer le déroulement de la recherche. Cette représentation doit être extensible de manière à définir de nouveaux paramètres ou de nouvelles méthodes d'indexation pour la recherche. Nous devons permettre également le choix entre divers paramétrages.

Dans ce but, nous allons représenter les informations de contrôle dans une hiérarchie de classes (cf figure 6.5) où la classe `LOCAL-CONTROL` est la racine des classes de contrôle pour des prototypes, la classe `INDEXATION-CONTROL` est la racine des classes de contrôle pour les différentes méthodes d'indexation mises en jeu dans le filtrage et la sélection, et la classe `RESEARCH-CONTROL` est la classe de contrôle du processus de recherche tout entier. Une fois que nous aurons défini des classes particulières pour chaque type de contrôle, les instances de ces classes peuvent être considérées comme des paramétrages différents et indépendants.

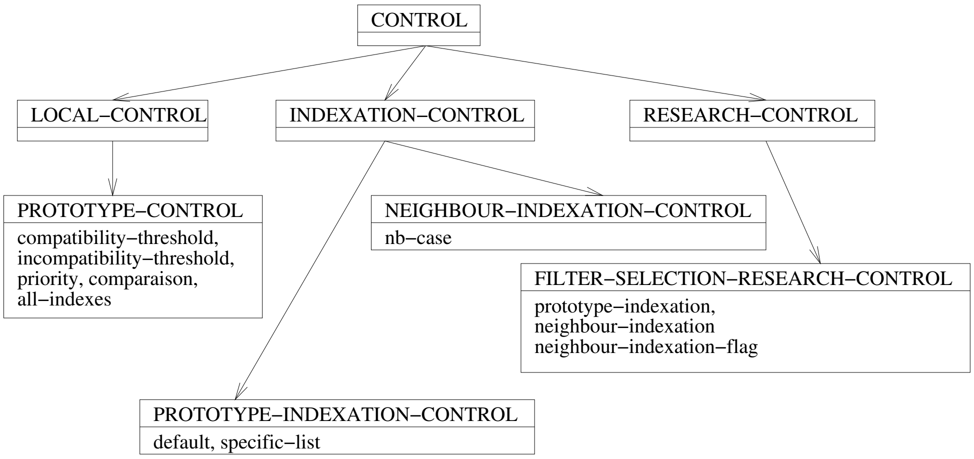

Figure 6.5 : Hiérarchie des contrôles.

Contrôle de la recherche

Le contrôle de la recherche que nous proposons dans la classe `FILTER-SELECTION-REASEARCH-CONTROL` propose de décomposer la recherche en une étape de filtrage (représentée par un objet de type `PROTOTYPE-INDEXATION-CONTROL`) et une étape optionnelle de sélection (objet de type `NEIGHBOUR-INDEXATION-CONTROL`).

Contrôle des prototypes

Nous avons ainsi la classe `PROTOTYPE-CONTROL` qui permet de représenter toutes les informations de contrôle pour un prototype : seuil de compatibilité et d'incompatibilité, priorité, type de comparaison, acceptation de chaque index. Nous créons un objet de contrôle de la facon suivante :

```
(defconstant *prototype-ctrl*
           (make-prototype-control 0.4 1 5 'inherited t))
```

Contrôle de l'indexation par classification

La classe `PROTOTYPE-INDEXATION-CONTROL` permet de définir tous les paramètres pour l'indexation par classification : contrôle de prototypes par défaut et liste d'association entre des prototypes et leurs informations de contrôle associées. La gestion d'un défaut et d'une liste d'association spécifique permet l'affectation de paramètres soit pour tous les prototypes qui n'ont pas de paramètres spécifiques soit de manière spécifique.

Nous donnons un exemple de création de l'objet de contrôle où l'instance `*prototype-indexation-ctrl*` représente le contrôle par défaut :

```
(defconstant *prototype-indexation-ctrl*
           (make-prototype-indexation-control *prototype-ctrl* ()))
```

Les fonctions génériques `SET-DEFAULT-CONTROL`, `SET-SPECIFIC-LIST`, `SET-PROTOTYPE-CONTROL`, `GET-PROTOTYPE-CONTROL` permettent de gérer l'objet de contrôle.

Contrôle de l'indexation par plus proches voisins

Les paramètres de l'indexation par plus proches voisins sont définis dans la classe `NEIGHBOUR-INDEXATION-CONTROL`. En fait, le seul paramètre est le nombre maximum de cas à sélectionner.

6.3.2 Protocoles

Nous avons besoin d'implanter les différentes méthodes d'indexation et de recherche de manière à pouvoir les faire évoluer facilement si c'est nécessaire. Dans ce but nous nous sommes inspirés de la notion de protocole que l'on trouve dans la réalisation même des langages objet de haut niveau comme IlogTalk et CLOS où il existe des protocoles d'héritage et d'instanciation par exemple.

Qu'est ce qu'un protocole ?

Un protocole est déterminé par une hiérarchie d'étapes et de sous-étapes. Toutes les étapes sont représentées par des fonctions génériques, et les liens de la hiérarchie sont alors des appels à d'autres fonctions génériques. Le but d'un protocole est de bien spécifier les différentes étapes et de permettre leur personnalisation pour pouvoir adapter le processus global grâce à la spécialisation des fonctions génériques (cf figure 6.6).

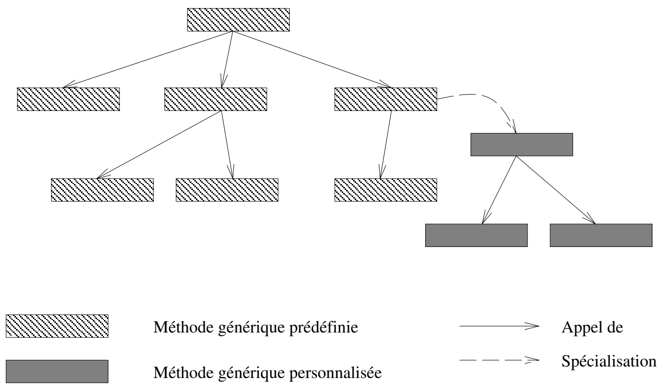

Figure 6.6 : Spécialisation d'un protocole.

Les feuilles de cette hiérarchie représentent les étapes de base qui sont faciles à modifier. Par contre, plus on monte dans la hiérarchie, plus les étapes représentent les mécanismes de l'algorithme et sont donc difficiles à modifier, à moins de changer totalement d'algorithme.

Dans notre problème, toutes les fonctions génériques mises en jeu prennent un objet de contrôle comme premier paramètre, donc pour spécialiser une étape il faut créer une sous-classe d'une classe de contrôle et créer de nouvelles méthodes correspondant aux fonctions génériques du protocole dont on veut modifier le comportement. Ces nouvelles méthodes doivent prendre en paramètre un objet instance de la nouvelle classe de contrôle. La nouvelle classe de contrôle peut bien sûr définir de nouveaux slots qui peuvent être utilisés dans les nouvelles méthodes.

Protocole de recherche

```
CASE-RESEARCH
    INDEXATION-BY-CLASSIFICATION
    CHANGE-INDEXES-FOR-NEIGHBOUR
    INDEXATION-BY-NEAREST-NEIGHBOUR
```

Ce protocole peut être spécialisé par une sous-classe de `FILTER-SELECTION-RESEARCH-CONTROL`. Si la recherche constiste dans l'utilisation d'une seule méthode d'indexation, il faut directement lancer le protocole de la méthode d'indexation voulue avec l'objet de contrôle correspondant.

Protocole d'indexation par classification floue

```
INDEXATION-BY-CLASSIFICATION
    CASE-CLASSIFICATION
        BEGIN-EXAMINATION-PROTOTYPE
        SPECIALIZE-CASE
        COMPLETE-CASE
            GET-PROTOTYPE-FIELDS
            COMPLETE-FIELD
            COMPLETE-CHECK
        COMPUTE-COMPATIBILITY-INCOMPATIBILITY
            GET-PROTOTYPE-INDEXES
            INDEX-COMPATIBILITY
                COMPUTE-INDEX-COMPATIBILITY
            INDEX-INCOMPATIBILITY
                COMPUTE-INDEX-INCOMPATIBILITY
            IS-INDEX-ACCEPTABLE
            COMPUTE-PROTOTYPE-COMPATIBILITY-INCOMPATIBILITY
            IS-PROTOTYPE-ACCEPTABLE
        COMPUTE-RESULT-LIST
            COMPATIBILITY-TO-POSSIBILITY
        COMPUTE-PROTOTYPE-LIST
        END-EXAMINATION-PROTOTYPE
    GET-PROTOTYPE-CASE-LIST
```

Ce protocole peut être spécialisé par une sous-classe de `PROTOTYPE-INDEXATION-CONTROL`.

Protocole d'indexation par plus proches voisins

```
INDEXATION-BY-NEAREST-NEIGHBOUR
    COMPARE-CASE
        SELECT-INDEXES-TO-COMPARE
        COMPUTE-INDEX-SIMILARITY
        COMPUTE-CASE-SIMILARITY
    SORT-CASE-LIST
    SELECT-CASES
```

Ce protocole peut être spécialisé par une sous-classe de `NEIGHBOUR-INDEXATION-CONTROL`.

Exemple de spécialisation du protocole d'indexation par classification

Nous pouvons montrer comment réaliser une partie du contrôle par spécialisation d'un protocole. Effectivement, pour réaliser l'indexation par classification de manière non interactive comme nous l'avions spécifié, nous pouvons créer une nouvelle classe de contrôle qui hérite de `PROTOTYPE-INDEXATION-CONTROL`. Ensuite, il suffit d'écrire de nouvelles méthodes pour `COMPLETE-FIELD` et `COMPLETE-CHECK` de manière à ce que les questions ne soient plus posées à l'utilisateur mais que les valeurs des champs soient déterminées par la consultation d'une description du problème stockée dans l'objet de contrôle. `BEGIN-EXAMINATION-PROTOTYPE` doit également être spécialisée pour éviter d'afficher des messages.

6.4 Conclusion et limites de l'implantation

Nous avons conçu une architecture la plus stable possible de manière à pouvoir faire évoluer le système le plus facilement. Nous avons donc utilisé tout le potentiel de représentation du langage IlogTalk : métaclasse, classes et fonctions génériques.

L'indexation par classification floue a été implantée et nous proposons un exemple complet dans l'annexe D. Toutefois, l'implantation du protocole d'indexation par plus proches voisins et du protocole de recherche est en cours de réalisation.

Chapitre 7

Conclusion

Notre objectif était de concevoir et de réaliser un système d'aide à la recherche des cas similaires dans le cadre des problèmes de conception. Nous avons commencé par une étude globale du raisonnement par cas, puis nous nous sommes intéressés plus particulièrement à la phase de recherche des cas. Pour notre système, il a fallu choisir une représentation des cas adaptée aux problèmes de conception : notre choix s'est porté sur une représentation basée sur des *hiérarchies de prototypes* qui permettent de gérer des cas *composites* . De plus nous avons permis l'expression de *l'imprécis* et *des préférences* dans les cas et dans l'énoncé des problèmes, grâce à l'utilisation des *ensembles flous.*

Pour les mécansimes de recherche, nous avons proposé *l'indexation par classification floue* qui permet d'utiliser les hiérarchies de prototypes. Cette indexation cherche à classer le nouveau problème parmi des prototypes de problèmes, et la gestion du flou en utilisant la théorie des possibilités permet d'avoir un résultat plus réaliste. Les cas appartenant aux prototypes de problèmes sont alors examinés un par un dans l'indexation par plus proches voisins. Cette indexation est basée sur des métriques de similarité et permet de sélectionner le cas le plus proche du problème. Pour cette indexation, nous avons proposé de *nouvelles métriques* qui sont adaptées à la représentation composite et floue des cas.

Nous avons enfin réalisé un module de recherche permettant la représentation des cas et des hiérarchies de prototypes ainsi que l'indexation par classification floue. Ce module a été conçu pour être évolutif et a été implanté dans le langage IlogTalk.

La validation de notre module est en cours d'une part avec un exemple portant sur la conception de satellites de télécommunications de type SpaceBus et d'autre part dans le cadre d'une collaboration entre le projet SECOIA (INRIA) et le projet FABEL (GMD). Toutefois, nos premiers tests montrent quelques inconvénients de l'indexation par classification floue : les hiérarchies de prototypes sont utilisées à la fois pour représenter les cas et comme support pour l'indexation par classification. La définition des hiérarchies de prototypes doit donc forcément prendre en compte le fait que les hiérarchies seront utilisées pendant l'indexation.

Durant cette étude, nous avons été amenés à aborder d'une part les travaux existants en CBR et d'autre part les liens entre le CBR et différents domaines: conception, psychologie, classification, ensemble flou et théorie des possibilités, base de données floues, analyse de données. Nous avons également étudié différentes applications CBR comme ReMind, CHEF et CADET, et les deux genérateurs de systèmes experts SMECI (et son module flou) et CLASSIC.

Les perspectives de notre étude sont nombreuses, citons en particulier :

- gestion de domaines de caractéristiques plus abstraites (par exemple des caractéristiques liées au comportement des objets),
- amélioration et validation cognitive de l'indexation par classification floue et sa comparaison avec le seul (à notre connaissance) système de raisonnement analogique manipulant du flou, le système FLORAN[Sal89],
- recherche de meilleures métriques de similarité pour l'indexation par plus proches voisins toujours basées sur des ensembles flous,
- étude de nouvelles méthodes d'indexation en particulier basées sur des algorithmes d'induction, et de leur intégration dans notre système.

De plus, dans le cadre d'un générateur de système de CBR, il faudrait étudier, en liaison avec la phase de recherche, les phases d'adaptation et d'apprentissage.

Annexe A

Les ensembles flous et la théorie des possibilités

A.1 Les ensembles flous

A.1.1 Définition

Les ensembles flous ont été introduits par Zadeh [Zad65] Un ensemble flou A sur le domaine D est défini par la donnée d'une fonction μ_A à valeur dans [0,1]. μ_A est appelé *fonction caractéristique* de A et $\mu_A(x)$ représente le degré d'appartenance d'un élément x de D à A. La fonction caractéristique peut être soit continue soit discrète.

La figure A.1 montre un ensemble flou général et la figure A.2 représente un ensemble flou de type trapèze très souvent utilisé et étant décrit par quatre caractéristiques (a, b, tolérance$_a$, tolérance$_b$).

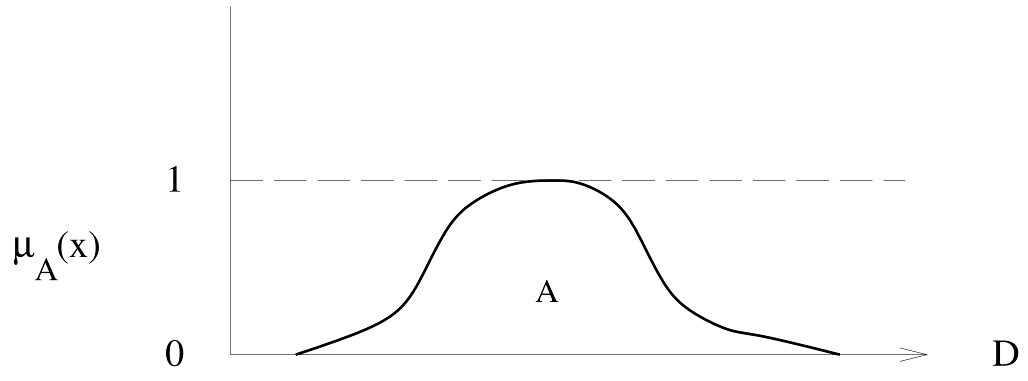

Figure A.1 : Un ensemble flou

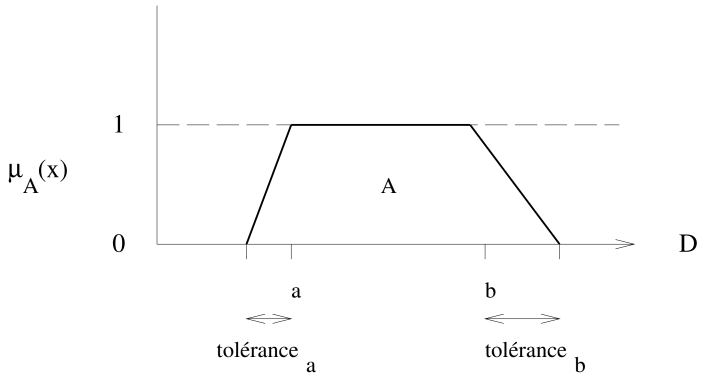

Figure A.2 : Un ensemble flou de type trapèze

A.1.2 Opérations sur les ensembles flous

Diverses familles d'axiomes de définitions ont été proposées. Etant donnés A et B deux ensembles flous sur le domaine D, les définitions les plus couramment adoptées sont les suivantes [Ha91b] :

- l'inclusion :
$$\mu_A(x) \leq \mu_B(x) \text{ si } B \subseteq A$$
- l'égalité :
$$\mu_A(x) = \mu_B(x) \text{ si } A = B$$
- l'intersection :
$$\mu_{A\cap B}(x) = \min(\mu_A(x), \mu_B(x))$$
- l'union :
$$\mu_{A\cup B}(x) = \max(\mu_A(x), \mu_B(x))$$
- la complémentation :
$$\mu_{\bar{A}(x)} = 1 - \mu_A(x)$$
- le produit cartésien :
$$\mu_{A\times B}(x, y) = \min(\mu_A(x), \mu_B(y))$$
- la cardinalité (A dénombrable) :
$$card(A) = \sum_{x_j \in A} \mu_A(x_j)$$

A.1.3 Utilisation

Les ensembles flous sont utilisés pour représenter différents concepts :

- l'exactitude (cf figure A.3 et A.4),

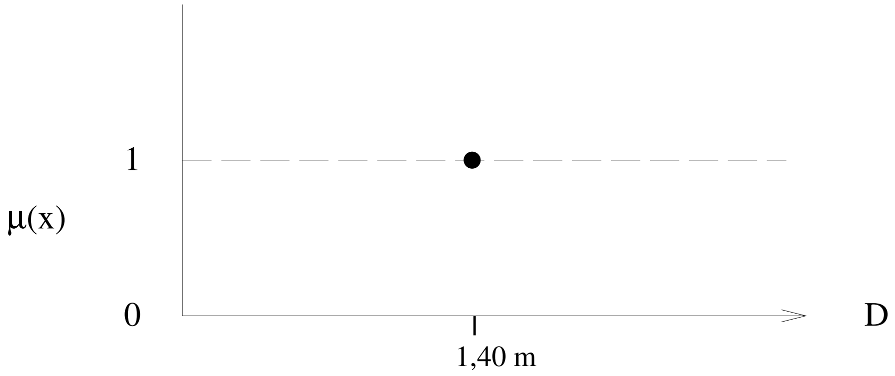

Figure A.3 : "Cet enfant mesure exactement 1,40 m"

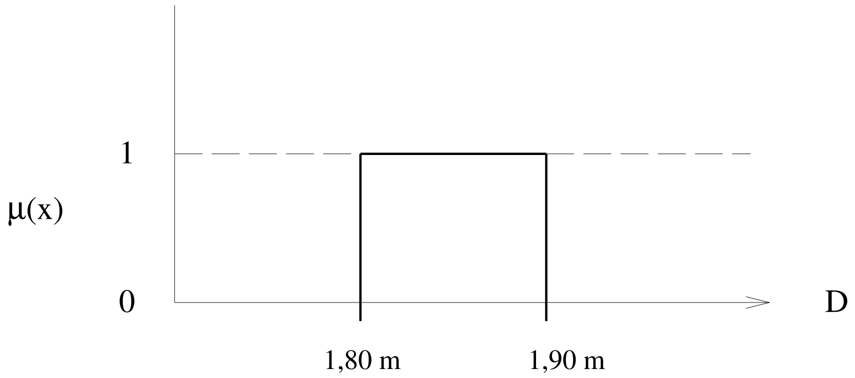

Figure A.4 : "Il mesure entre 1,80 et 1,90m exactement"

- l'imprécis et l'incertain (cf figure A.5, A.6 et A.7)),

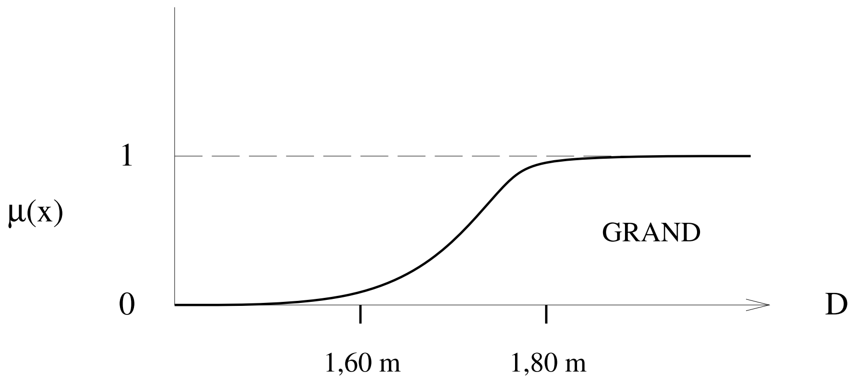

Figure A.5 : Le concept de "grand"

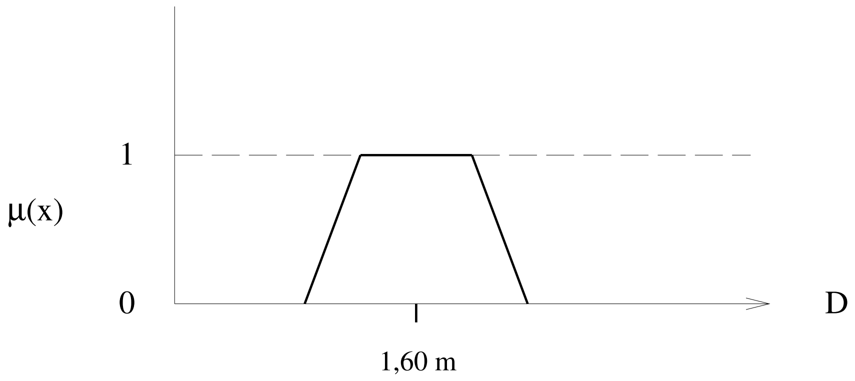

Figure A.6 : "Il mesure environ 1,60m"

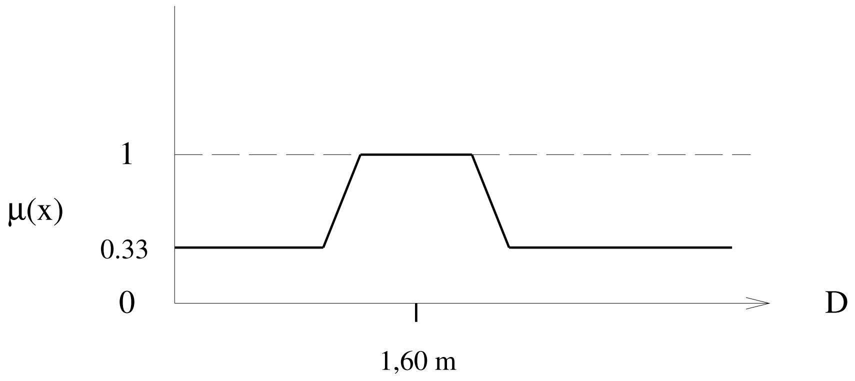

Figure A.7 : "Il mesure environ 1,60m avec un degré d'incertitude de 0,33"

- les préférences (cf figure A.8),

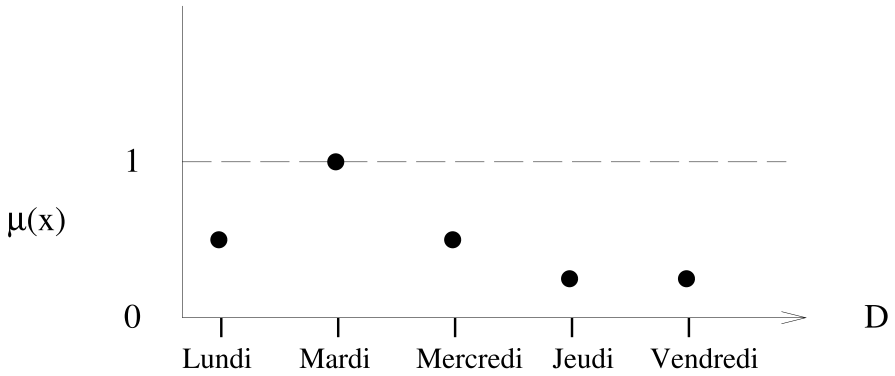

Figure A.8 : "Le professeur pourra donner son cours en début de semaine, de préférence le mardi"

A.2 Résultat de la théories des possibilités

Zadeh dans [Zad78] a proposé d'étendre la théorie des possibilités en utilisant les ensembles flous comme des distributions de possibilités. Il est alors possible de calculer les coefficients de possibilité et de nécessité, et étant donnés A et B deux ensembles flous, on a :

- la possibilité :

$$\begin{aligned} \prod(A|B) &= \max_{x \in D}(\min(\mu_A(x), \mu_B(x))) \\ &= \text{degré d'intersection de } A \text{ et } B \end{aligned}$$

- la nécessité :

$$\begin{aligned} N(A|B) &= \min_{x \in D}(\max(\mu_A(x), 1 - \mu_B(x))) \\ &= \text{degré d'inclusion de } B \text{ dans } A \end{aligned}$$

Pour plus de détails sur les fondements théoriques de ces résultats, veuillez vous reporter à [Zad78] [DP87] [DDT88].

Ainsi, soient A et B deux ensembles flous, soit la proposition "X est B" alors les quantités $\prod(A|B)$ et $N(A|B)$ représentent respectivement la possibilité et la nécessité que X soit A.

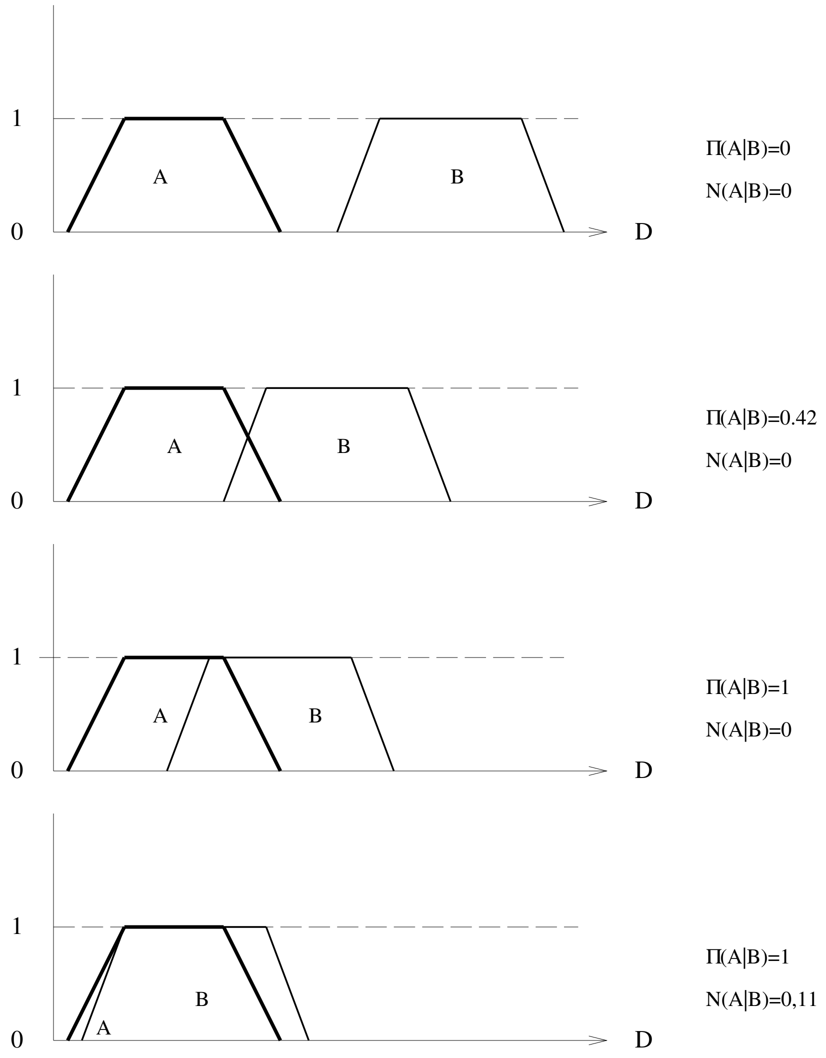

Figure A.9 : Exemple de calcul de possibilités et de nécessités

Annexe B

Le générateur de systèmes experts en classification CLASSIC

B.1 Présentation

CLASSIC est un générateur de systèmes experts en classification et en diagnostic développé par l'INRIA [ILO87] (en Le_Lisp V15). Il permet de modéliser l'incertain et l'imprécis grâce à l'utilisation de la théorie des possibilités.

CLASSIC est conçu autour de trois éléments principaux:

- une base de connaissances,
- un objet à classer,
- un moteur de classification.

B.2 La base de connaissances

Les connaissances peuvent être représentées dans deux formalismes : une hiérarchie de prototypes, et des règles de production.

B.2.1 Les hiérarchies des prototypes

Les prototypes sont organisés dans une hiérarchie d'héritage simple. Chaque prototype possède une liste de champs. Les champs peuvent être de trois types :

- ensemble : ensemble de symboles, par exemple {rouge, vert, bleu};
- intervalle : intervalle de réels, par exemple l'intervalle [10, 20];
- prototype : ce type sert à désigner un objet qui peut être vu comme une sous-partie, ce sous-objet est associé à sa propre hiérarchie.

Le domaine de valeurs de chaque champ peut être restreint dans un prototype plus spécifique. Par exemple, un champ de type intervalle qui a pour domaine l'intervalle [10 20] dans un certain prototype, pourra être restreint à [10 15] dans le prototype fils. De même, un champ de type ensemble peut être restreint à un sous-ensemble et un champ de type prototype à un sous-prototype.

Les champs possèdent d'autres caractéristiques (facettes) qui sont utilisées lors de la classification :

- un coefficient d'importance au niveau du prototype,
- une valeur par défaut,
- un indicateur pour savoir si la valeur du champ doit être demandée à l'opérateur,
- un indicateur pour savoir si la valeur du champ doit être calculée,
- des commentaires.

B.2.2 Les règles

Les règles sont attachées à un prototype et servent en particulier à déterminer les valeurs de certains champs. Les règles ont des prémisses qui ont toutes un coefficient d'importance et des conclusions qui ont des coefficients de confiance. Les prémisses peuvent utiliser des prédicats flous tolérants ($\tilde{\doteq}$, $\tilde{<}$, $\tilde{>}$, proche-de, loin-de) représentés par des ensembles flous. Par exemple, nous pouvons écrire la règle suivante :

```
IF taille proche-de  180 THEN stature=grande
```

Cette règle attribut à stature la valeur grande avec une possibilité de 1 si taille vaut 180, 178 ou 182; ou la valeur grande avec une possibilité de 0.5 si taille vaut 175.5 ou 184.5.

B.3 L'objet à classer

L'objet à classer est une instance d'un prototype. Cet objet est défini par :

- son nom,
- son ou ses prototypes avec lesquels l'objet est compatible (au départ cet ensemble de prototypes est inconnu),
- la totalité des champs présents dans la hiérarchie.

Si le champ est de type intervalle, la valeur du champ est soit inconnue soit un réel avec une possibilité. Si le champ est de type ensemble, la valeur du champ est soit un ensemble flou sur l'ensemble des symboles, soit inconnue. Dans le cas d'un champ de type prototype, la valeur du champ est à son tour un objet.

Lors de la classification, certaines valeurs seront calculées (par des règles ou par des fonctions) ou demandées à l'opérateur de manière à décrire l'objet et en même temps le classer.

B.4 Le moteur

B.4.1 Fonctionnement standard

CLASSIC utilise la technique du raffinage descendant, c'est-à-dire que le système parcourt la hiérarchie de prototypes du plus général au plus spécifique pour enfin trouver le ou les prototypes qui décrivent le mieux l'objet à classer.

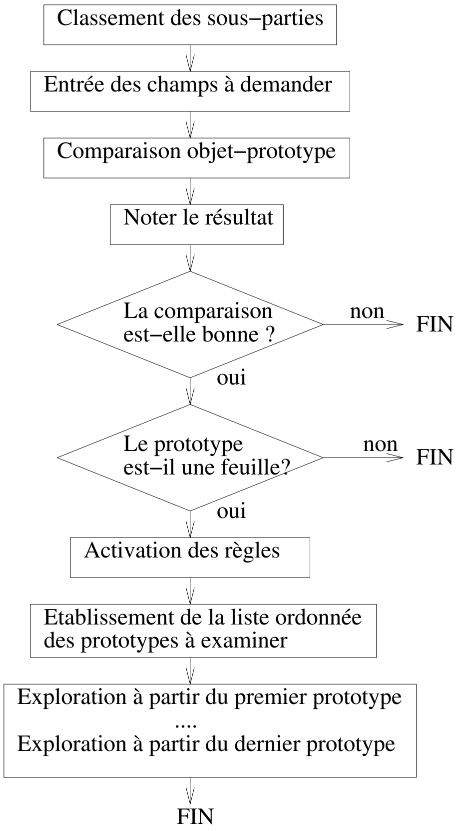

Figure B.1 : Comportement du moteur à chaque prototype

A chaque prototype, CLASSIC effectue cinq grandes étapes (figure B.1):

1. Classement des sous-parties :
   Si l'objet contient des sous-parties non classées, le moteur les classe d'abord. Ainsi nous pouvons continuer le processus.

2. Comparaison objet-prototype :
   La comparaison s'effectue en trois étapes :

   (a) Calcul des coefficients de compatibilité et d'incompatibilité pour chaque champ : tout d'abord, suivant les facettes des champs, le système demande la valeur des champs à l'opérateur, calcule la valeur ou utilise la valeur par défaut. De manière générale, étant donnée la caractéristique C, p l'ensemble flou défini dans le prototype pour un champ et v l'ensemble flou représentant la valeur du champ dans l'objet, on a:

$$\begin{aligned} \text{compatibilité}(C) &= \quad \Pi(p|v) \\ \text{incompatibilité}(C) &= \quad 1 - N(p|v) \end{aligned}$$

Pour le type intervalle, p est un ensemble flou de type trapèze avec une tolérance de 5% et dont les bornes intérieures sont les bornes de l'intervalle. v est un ensemble flou réduit à une valeur (la valeur du champ avec sa possibilité).

Pour le type ensemble, p est un ensemble flou exact (tous les symboles définissant le type ont une possibilité de 1, les autres ont une possibilité de 0), et v est un ensemble flou discret.

(b) Calcul des coefficients de compatibilité et d'incompatibilité pour le prototype : grâce aux coefficients calculés sur chaque champ, nous pouvons obtenir les coefficients globaux.

$$\begin{aligned} \text{compatibilité} &= \quad \min_i(\max(\text{compatibilité}(Champ_i), \text{1-importance}(Champ_i))) \\ \text{incompatibilité} &= \quad \max_i(\min(\text{incompatibilité}(Champ_i), \text{importance}(Champ_i))) \end{aligned}$$

(c) Rejet ou acceptation du prototype : Suivant les coefficients globaux ainsi calculés et les seuils définis nous obtenons une zone de rejet et une zone d'acceptation comme le montre la figure B.2.

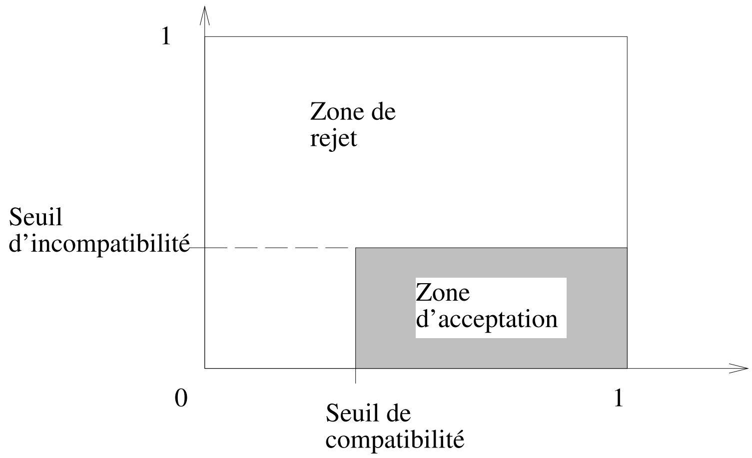

Figure B.2 : Zones de rejet et d'acceptation

3. Activation des règles :
   Nous ne rentrerons pas dans le détail des mécanismes de fonctionnement des règles. Toutefois, il faut savoir qu'un coefficient d'activation de la règle est calculé en prenant en compte la possibilité de chaque prémisse et l'importance des prémisses.

4. Etablissement de la liste ordonnée des prototypes à examiner :
   Le système ajoute les prototypes fils d'un prototype accepté dans la liste des prototypes à examiner.

5. Exploration des prototypes :
   Le système explore les prototypes dans l'ordre de la liste des prototypes à examiner. Ce mécanisme implique une exploration en profondeur d'abord.

B.4.2 Modification du fonctionnement

Les attributs de contrôle permettent de modifier le comportement normal du moteur. Ils peuvent être spécifiés au niveau de chaque prototype ou bien globalement pour tous les prototypes. Ils peuvent modifier :

- la comparaison, grâce à plusieurs moyens :
  - le calcul des coefficients peut ou non prendre en compte les coefficients calculés pour le prototype père,
  - les seuils de compatibilité et d'incompatibilité peuvent être changés,
  - la comparaison peut s'arrêter dès qu'un champ n'est pas dans la zone d'acceptation au lieu de prendre en compte tous les champs.
- l'établissement de la liste des prototypes : chaque prototype peut avoir une priorité, ainsi il est possible d'ordonner les prototypes à examiner.
- l'exploration des prototypes : l'exploration peut se faire soit en parcourant l'arbre tout entier, soit en descendant dans une branche uniquement, soit en s'arrêtant sur la première feuille.

Annexe C

TELOS : la couche objet de IlogTalk

Notre but est de présenter de manière simple les mécanismes les plus importants de la couche objet de IlogTalk (appelée TELOS) à travers des exemples, et il ne s'agit pas d'une description exhaustive. Pour plus de renseignements, veuillez vous reporter au manuel de référence [ILO93]. Nous allons donc commencer par une brève description de cette couche objet, puis nous pourrons alors illustrer certains mécanismes par des exemples tirés de notre application.

C.1 Description de TELOS

TELOS comme CLOS se distingue des autres langages à objets par le fait que tous les objets de l'environnement sont des instances d'une classe, et qu'il existe un "Meta-Object Protocol" permettant de personnaliser les mécanismes de la couche objet. Nous allons présenter les mécanismes de base d'une part pour la définition des objets (classes, instances, métaclasses) et d'autre part pour la gestion de ces objets par les objets particuliers que sont les fonctions génériques.

C.1.1 Classes, slots et instances

Nous allons d'abord expliquer comment définir une classe puis nous verrons comment instancier cette classe et ainsi créer de nouveaux objets.

Définition d'une classe

Une classe est un objet qui permet de définir les caractéristiques appelées *slots* d'une famille d'instances. Les classes sont organisées dans une hiérarchie d'héritage permettant de définir les classes de la plus générale à la plus spécifique. La racine du graphe d'héritage est la classe `object`.

Une classe peut être définie par la forme de définition `defclass`, par exemple :

```
(defclass piece ()
     ((longueur initform 0 accessor longueur initarg long initarg cote)
      (largeur initform 0 reader largeur writer set-largeur
               initarg larg initarg cote))
     constructor (make-piece (long larg))
     constructor (make-piece-carre (cote))
     predicate piece-p)
```

`defclass` prend en arguments : le nom de la classe, une liste de noms des super-classes (classes dont la classe doit hériter), une liste de descriptions de slots, et des options de classes. Si la liste des supers-classes est vide, la classe hérite par défaut de la classe `object`.

Un slot est décrit par son nom suivi des options de slots. Une option de slot est représentée par un mot-clé suivi de sa valeur. Il existe plusieurs options de slots :

- `slot-class` permet de spécifier la classe du slot. Il est en effet possible de créer de nouvelles classes de slots pour définir des facettes ou des comportements différents. Il existe deux classes de slots prédéfinies `local-slot-description` pour les slots d'instances (c'est la valeur par défaut) et `shared-slot-description` pour les slots partagés.

- `slot-initargs` donne la liste d'initialisation pour un slot d'une classe particulère. Cette liste permet de donner les valeurs des facettes spécifiques à la classe du slot.

- `initarg` définit un mot-clé d'initialisation. Ce mot-clé permet d'identifier un ou plusieurs slots, et il sera utilisé dans la définition des constructeurs et dans l'appel à `make-instance` pour initialiser l'instance.

- `initform` représente la valeur d'initialisation par défaut.

- `accessor` définit un accesseur en lecture et un autre en écriture. Cet accesseur pourra être utilisé pour lire ou écrire une valeur dans le slot. Il faut noter que l'écriture se fait alors par un `setf`.

- `reader` définit un accesseur en lecture.

- `writer` définit un accesseur en écriture. L'écriture se fait directement par l'appel de l'accesseur, à la différence d'un accesseur en écriture créé par `accessor`.

Certaines options de slots peuvent apparaître plusieurs fois et d'autres non.

De la même manière, les options de classes sont représentées par des mots-clés et une valeur.

- `metaclass` donne la métaclasse à utiliser.

- `metaclass-initargs` donne la liste d'initialisation pour une classe qui est construite à partir d'une métaclasse particulière.

- `constructor` définit un constructeur. Un constructeur est une fonction qui permet de créer une instance.

- `printer` donne une fonction d'impression.

- `predicate` définit un prédicat. Un prédicat permet de tester si un objet est du type défini par la classe.

Instances

Une instance peut être créée soit par un appel à `make-instance` soit par l'utilisation d'un constructeur.

```
? (defstatic p1 (make-instance 'piece 'larg 5 'long 3))
= p1
? (defstatic p2 (make-piece 6 4))
= p2
? (defstatic p3 (make-piece-carre 4))
= p3
```

Il est alors possible de lire les valeurs des slots de l'instance :

```
? (longueur (static p3))
= 4
? (largeur (static p3))
= 4
? (slot-value (static p1) 'longueur)
= 3
```

Il est également possible d'écrire dans ces slots :

```
? (setf (longueur (static p1)) 6)
= 6
? (set-largeur (static p1) 2)
= 2
? (set-slot-value (static p1) 'longueur 10)
= 10
```

Il faut noter que `slot-value` et `set-slot-value` permettent d'accéder à un slot mais utilisent toujours les accesseurs définis par défaut contrairement à CLOS.

C.1.2 Métaclasses

Les métaclasses sont des classes particulières dont les instances sont des classes. Concrètement, une métaclasse est un objet qui est instance d'une classe et qui hérite d'une métaclasse. La métaclasse generale du système est la métaclasse `class`. Il faut souligner le phénomène de *bootstrap* qui veut que la métaclasse *classe* hérite d'elle-même.

Une métaclasse se définit comme une classe :

```
(defclass classe-avec-commentaires (class)
    ((commentaires initform "" accessor commentaires initarg doc)))
```

Il est alors possible de créer une instance de cette métaclasse qui est une classe.

```
(defclass appartement ()
    ((pieces initform () accessor pieces))
    metaclass classe-avec-commentaires
    metaclass-initargs (doc "hello"))
```

Et comme pour n'importe quelle instance, il est possible de lire ou d'écrire dans les slots.

```
? (commentaires (find-class 'appartement))
= hello
? (setf (commentaires (find-class 'appartement)) "bonjour")
= bonjour
```

C.1.3 Fonctions génériques

Une fonction générique est identifiée par un nom et représente un ensemble de methodes. Chaque méthode permet de spécialiser ses paramètres, de manière à avoir un comportement spécifique. En fait, lors de l'appel d'une fonction générique, le système sélectionne toutes les méthodes applicables pour les paramètres de l'appel, puis il ordonne ces différentes méthodes de la plus spécifique à la plus générale. La méthode la plus spécifique est déclenchée et peut faire appel aux méthodes plus générales grâce à la fonction `call-next-method`.

Pour les besoins de l'exemple, nous créons des classes et des instances:

```
(defclass A1 () ((x accessor x initarg x)))
(defclass A2 (A1) ())
(defclass B1 () ((y accessor y initarg y)))
(defstatic a (make-instance 'A1 'x 2))
(defstatic b (make-instance 'A2 'x 3))
(defstatic c (make-instance 'B1 'y 4))
```

Nous déclarons une fonction générique avec deux méthodes:

```
(defgeneric calcul (a b))
(defmethod calcul ((a A1) (b B1))
(* (x a) (y b)))
(defmethod calcul ((a A2) (b B1))
(+ (x a) (y b)))
```

Puis nous appelons la fonction générique de deux façon différentes:

```
? (calcul (static a) (static c))
= 8
? (calcul (static b) (static c))
= 7
```

Dans le premier cas, c'est la première méthode qui a été exécutée, et dans le deuxième cas l'autre méthode.

Les fonctions génériques permettent notamment de représenter les protocoles qui sont à la base des mécanismes de IlogTalk : protocole d'instanciation et d'héritage. Un protocole consiste en une série de fonctions génériques qui sont appelées (lors de l'instanciation ou de la réalisation de l'héritage) et qui peuvent être spécialisées tout comme dans notre exemple. Le protocole d'instanciation est constitué de deux fonctions génériques : `allocate-instance` pour l'allocation de la mémoire et `insitialize-instance` pour l'initialisation. Le protocole d'héritage est toutefois beaucoup plus complexe.

C.2 Exemples tirés de notre application

Nou voulons montrer les possibilités de TELOS à travers trois exemples que nous avons réalisés pour les besoins de notre application : création de classes virtuelles, utilisation des accesseurs, definition de facettes.

C.2.1 Création de classes virtuelles

Une classe virtuelle est une classe qui ne peut pas avoir d'instances. Toutefois ce caractère n'est pas hérité dans les sous-classes. Ce type de classe permet de mieux représenter des classes abstraites. Il s'agit de définir un comportement nouveau pour un ensemble de classes, donc nous allons créer une nouvelle métaclasse que nous appelons `virtual-class`.

```
(defclass virtual-class (class) ())
```

Pour que les classes qui sont instances de cette nouvelle métaclasse puissent hériter des autres classes et inversement, nous déclarons les métaclasses `virtual-class` et `class` compatibles dans les deux sens de l'héritage.

```
(defmethod compatible-metaclass-p ((subclass virtual-class) (superclass class))
  (ignore subclass superclass)
  t)
(defmethod compatible-metaclass-p ((subclass class) (superclass virtual-class))
  (ignore subclass superclass)
  t)
```

Pour que la création d'instances soit impossible pour des classes de type `virtual-class`, nous spécialisons le protocole d'instanciation en définissant une nouvelle méthode pour `allocate-instance` :

```
(defmethod allocate-instance ((class virtual-class) init-list)
  (ignore init-list)
  (error 'allocate-instance "virtual class" (class-name class)))
```

Nous définissons également une nouvelle macro de création de classes virtuelles :

```
(defmacro defvirtualclass (name supers slots &rest options)
  `(defclass ,name ,supers ,slots ,@options
     metaclass virtual-class))
```

Pour finir, un petit exemple :

```
? (defvirtualclass C1 ()
          ((a accessor a)))
= c1
? (defclass C2 (C1) ())
= c2
? (defdynamic x (make-instance 'C1 'a 12))
 ** allocate-instance : virtual class : c1
= ()
? (defdynamic x (make-instance 'C2 'a 12))
= x
```

C.2.2 Utilisation des accesseurs

Notre problème est de réaliser un mécanisme équivalent à celui de CLOS où les accesseurs peuvent être spécialisés et être utilisés comme démons d'accès en lecture ou en écriture.

Cette possibilité n'est pas explicitement offerte par IlogTalk, car il n'existe pas de fonctions permettant l'accès aux slots sans passer par les accesseurs. Or si nous voulons redéfinir les accesseurs, il faut avoir un accès direct pour ne pas boucler lors d'un accès. Nous définissons alors nos propres fonctions d'accès bas niveau.

```
(defun basic-slot-value (object slot-name)
  (let ((sl (find-slot (class-of object) slot-name)))
    (ifn sl (error 'basic-slot-value "invalid slot name" slot-name)
         (pointer-row-ref object (slot-description-position sl)))))

(defun basic-set-slot-value (object slot-name value)
  (let ((sl (find-slot (class-of object) slot-name)))
    (ifn sl (error 'basic-set-slot-value "invalid slot name" slot-name)
         (pointer-row-set object (slot-description-position sl) value))))
```

Par défaut, IlogTalk crée des accesseurs à un slot pour chaque classe où il est défini ou hérité. Il faut changer ce mécanisme pour permettre l'héritage des accesseurs pour les slots des classes plus spécifiques. Nous redéfinissons les méthodes de spécialisation des accesseurs qui sont appelées lors de la réalisation de l'héritage. Ces méthodes ne font que retourner l'ancien accesseur au lieu d'en créer des nouveaux.

```
(defmethod specialize-slot-description-writer ((ef local-slot-description)
                                               (c class)
                                               writer)
  (ignore ef c)
  writer)
(defmethod specialize-slot-description-reader ((ef local-slot-description)
                                               (c class)
                                               reader)
  (ignore ef c)
  reader)
```

Nous présentons maintenant un petit exemple :

```
? (defclass C1 ()
        ((a reader read-a writer set-a)
         (b reader read-b writer set-b)))
= c1
? (defclass C2 (C1) ())
= c2
?
? (defmethod read-a ((ob C1))
     (print "C1:lecture du champ")
     (basic-slot-value ob 'a))
```

```
= read-a c1
?
? (defmethod set-a ((ob C1) value )
     (print "C1:ecriture du champ")
     (basic-set-slot-value ob 'a value))
= set-a c1
?
? (defmethod set-a ((ob C2) value )
     (print "C2:ecriture du champ")
     (basic-set-slot-value ob 'a value))
= set-a c2
?
? (defmethod set-b ((ob C2) value )
     (print "C2:ecriture du champ")
     (call-next-method))
= set-b c2

? (defdynamic x (make-instance 'C1))
C1:lecture du champ
= x
? (defdynamic y (make-instance 'C2))
C1:lecture du champ
= y
?
? (set-a (dynamic x) 10)
C1:ecriture du champ
= 10
? (set-a (dynamic y) 20)
C2:ecriture du champ
= 20
? (set-b (dynamic y) 30)
C2:ecriture du champ
= 30
? (read-a (dynamic x))
C1:lecture du champ
= 10
? (read-a (dynamic y))
C1:lecture du champ
= 20
? (read-b (dynamic y))
= 30
```

Il est à noter que la méthode `set-b` fait appel à l'accesseur défini par défaut pour la classe `C1` par l'appel de `call-next-method`. Il faut remarquer également l'appel de l'accesseur en lecture lors de l'instanciation.

Dans notre système, nous avons utilisé cette possibilité pour faire notamment du contrôle de typage sur l'accès en écriture des slots.

C.2.3 Définition de facettes

Il faut créer une classe définissant les facettes :

```
? (defclass my-slot-type (local-slot-description)
    ((facet1 accessor facet1 initarg f1 )
      (facet2 accessor facet2 initarg f2))
    metaclass structure-class)
= my-slot-type
```

Puis, lors de la création d'un slot dans une classe, il faut spécifier la classe du slot avec le mot-clé `slot-class` et une liste d'initialisation des facettes avec le mot-clé `slot-initargs`. La liste d'initialisation est constituée des mots clés d'initialisation suivis des valeurs.

```
? (defclass C1 ()
    ((slot slot-class my-slot-type
           slot-initargs (f1 10 f2 'hello)
           reader read-slot)))
= c1
```

Il est alors possible de lire et d'écrire dans les facettes :

```
? (facet1 (find-slot (find-class 'C1) 'slot))
= 10
? (setf (facet1 (find-slot (find-class 'C1) 'slot)) 15)
= 15
? (facet1 (find-slot (find-class 'C1) 'slot))
= 15
```

L'héritage des facettes se fait par la spécialisation de la fonction générique `compute-effective-slot-description` pour deux cas différents. En effet, cette fonction générique est appelée de deux manière différentes : lorsqu'il y a redéfinition du slot et lorsqu'il n'y a pas redéfinition.

Lorsqu'il y a redéfinition, il faut combiner les valeurs héritées et les valeurs nouvellement spécifiées. Un mécanisme normal est de copier la valeur des facettes qui n'ont pas été redéfinies tout en gardant les nouvelles valeurs.

```
(defmethod compute-effective-slot-description ((class class)
         (directslot direct-slot-description)
         (islot my-slot-type) otherslots)
  (let ((ef (call-next-method)))
     (loop for x in '(facet1 facet2)
           do (unless (slot-bound-p (slot-value ef x))
                (set-slot-value ef x (slot-value islot x))))
      ef))
```

C'est dans cette fonction que l'on peut réaliser des mécanismes plus complexes comme des vérifications de cohérence entre les valeurs des facettes héritées et leur nouvelle valeur ou bien un héritage par concaténation pour les commentaires par exemple.

Lorsqu'il n'y a pas redéfinition, le mécanisme normal correspond à la copie des valeurs des facettes héritées.

```
(defmethod compute-effective-slot-description ((class class)
      (dslot my-slot-type)
      islot otherslots)
  (let ((ef (call-next-method)))
    (loop for x in '(facet1 facet2)
          do  (set-slot-value ef x (slot-value dslot x)))
    ef))
```

Annexe D

Interface fonctionnelle

Le but de cette annexe est de présenter l'interface fonctionnelle offerte au développeur et à l'utilisateur. Ainsi, nous présentons les définitions des classes, des fonctions génériques et des constantes globales du système. Nous allons commencer par la représentation des hiérarchies de prototypes et des cas puis nous détaillerons l'interface du module de recherche avec les classes de contrôle et le protocole d'indexation par classification floue.

D.1 Représentation des prototypes et des cas

Dans la représentation des prototypes et des cas, nous distinguons la gestion des prototypes et des cas eux-mêmes puis la gestion des propriétés des caractéristiques et enfin la gestion des valeurs des propriétés : domaines, restrictions et valeurs symboliques.

D.1.1 Gestion des prototypes et des cas

La métaclasse `prototype` permet de spécifier un comportement particulier pour les prototypes.

```
(defclass prototype (class)
  ((case-list initform () reader case-list writer set-case-list))
  predicate prototype-p)
```

La classe `CASE` est la racine des prototypes.

```
(defclass case () ()
  predicate case-p
  metaclass prototype)
```

(`DEFPROTOTYPE` *name supers slots*) [*macro*]

- ◇ Construit un nouveau prototype.
- ◇ Arguments :
  - *name* nom du prototype
  - *supers* liste des supers prototypes réduite à un élément
  - *slots* liste de descriptions de slots
- ◇ Valeur de retour : un prototype

`(PROTOTYPE-OF` *case* `)` [*fonction générique*]

- ◇ Retourne le prototype dont le cas *case* est une instance.
- ◇ Arguments :
  case un cas
- ◇ Valeur de retour : un prototype

`(PROTOTYPE-DIRECT-SUBPROTOTYPES` *prototype*`)` [*fonction générique*]

- ◇ Retourne la liste des noms des prototypes qui héritent directement de *prototype*.
- ◇ Arguments :
  prototype un prototype
- ◇ Valeur de retour : liste de prototypes

`(PROTOTYPE-DIRECT-SUPERPROTOTYPE` *prototype*`)` [*fonction générique*]

- ◇ Retourne le père de *prototype*.
- ◇ Arguments :
  prototype un prototype
- ◇ Valeur de retour : un prototype

`(SUBPROTOTYPE-P` *prototype1 prototype2*`)` [*fonction générique*]

- ◇ Indique si *prototype1* est un sous-prototype de *prototype2*.
- ◇ Arguments :
  prototype1 un prototype
  prototype2 un prototype
- ◇ Valeur de retour : un booléen

`(PROTOTYPE-NAME` *prototype* `)` [*fonction générique*]

- ◇ Retourne le nom du prototype *prototype*.
- ◇ Arguments :
  prototype un prototype
- ◇ Valeur de retour : un symbole

`(FIND-PROTOTYPE` *prototype-name* `)` [*fonction générique*]

- ◇ Retourne le prototype dont le nom est *prototype-name*.
- ◇ Arguments :
  prototype-name un symbole
- ◇ Valeur de retour : un prototype ou () s'il n'existe pas

`(THE-PROTOTYPE` *prototype-or-name* `)` [*fonction générique*]

- ◇ Si *prototype-or-name* est un symbole, la fonction retourne le prototype dont le nom est *prototype-or-name*, sinon retourne *prototype-or-name*.
- ◇ Arguments :
  prototype-or-name un prototype ou un symbole

◇ Valeur de retour : un prototype ou () s'il n'existe pas

(`PROTOTYPE-DIRECT-FIELDS` *prototype*) [*fonction générique*]

◇ Retourne les noms des slots directs (i.e. slots créés ou redéfinis dans le prototype *prototype*).

◇ Arguments :
prototype un prototype

◇ Valeur de retour : liste de symboles

(`PROTOTYPE-FIELDS` *prototype*) [*fonction générique*]

◇ Retourne les noms des slots effectifs (i.e. slots créés ou hérités dans le prototype *prototype*).

◇ Arguments :
prototype un prototype

◇ Valeur de retour : liste de symboles

(`PROTOTYPE-NEW-FIELDS` *prototype*) [*fonction générique*]

◇ Retourne les noms des nouveaux slots (i.e. slots créés mais pas redéfinis dans le prototype *prototype*).

◇ Arguments :
prototype un prototype

◇ Valeur de retour : liste de symboles

(`ADD-CASE-TO-PROTOTYPE` *case prototype*) [*fonction générique*]

◇ Associe le cas *case* au prototype *prototype*. Si l'association était déjà faite, on ne fait rien.

◇ Arguments :
case un cas
prototype un prototype

◇ Valeur de retour : -

(`REMOVE-CASE-FROM-PROTOTYPE` *case prototype*) [*fonction générique*]

◇ Enlève l'association entre le cas *case* et le prototype *prototype*. Si l'association n'existait pas, on ne fait rien.

◇ Arguments :
case un cas
prototype un prototype

◇ Valeur de retour : -

D.1.2 Gestion des proriétés

(`FUZZY-FIELD-P` *case slot-name*) [*fonction générique*]
(`FIELD-P` *case slot-name*) [*fonction générique*]
(`FIELD-PRECEDENCE` *case slot-name*) [*fonction générique*]
(`FIELD-QUESTION` *case slot-name*) [*fonction générique*]
(`FIELD-DOCUMENTATION` *case slot-name*) [*fonction générique*]
(`FIELD-DOMAIN` *case slot-name*) [*fonction générique*]
(`FIELD-RESTRICTION` *case slot-name*) [*fonction générique*]
(`FIELD-DEFAULT` *case slot-name*) [*fonction générique*]
(`FIELD-DEFAULT-INDEX` *case slot-name*) [*fonction générique*]
(`FIELD-DEFAULT-IMPORTANCE` *case slot-name*) [*fonction générique*]
(`FIELD-SYMBOLIC-VALUES` *case slot-name*) [*fonction générique*]
(`FIELD-IMPORTANCE` *case slot-name*) [*fonction générique*]
(`FIELD-ASKED-P` *case slot-name*) [*fonction générique*]
(`FIELD-INDEX-P` *case slot-name*) [*fonction générique*]
(`FIELD-VAL` *case slot-name*) [*fonction générique*]

- ◇ Lecture d'une propriété.
- ◇ Arguments :
  case un cas
  slot-name un symbole
- ◇ Valeur de retour : valeur de la propriété

(`SET-FIELD-IMPORTANCE` *case slot-name value*) [*fonction générique*]
(`SET-FIELD-ASKED` *case slot-name value*) [*fonction générique*]
(`SET-FIELD-INDEX` *case slot-name value*) [*fonction générique*]
(`SET-FIELD-VAL` *case slot-name value*) [*fonction générique*]

- ◇ Ecriture d'une propriété dynamique.
- ◇ Arguments :
  case un cas
  slot-name un symbole représentant le nom du slot
  value la nouvelle valeur
- ◇ Valeur de retour : -

D.1.3 Gestion des domaines

Nous présentons la définition des différentes classes de domaines.

```
(defvirtualclass domain () ()
  predicate domain-p)

(defvirtualclass hierarchical-domain (domain)
  ((hierarchy initarg hierarchy reader hierarchy writer set-hierarchy))
  predicate hierarchical-domain-p)
```

```
(defclass prototype-domain (hierarchical-domain) ()
  constructor (make-prototype-domain (hierarchy))
  predicate prototype-domain-p )

(defclass symbol-domain (hierarchical-domain) ()
  constructor (make-symbol-domain (hierarchy))
  predicate symbol-domain-p)

(defvirtualclass one-dimension-domain (domain) ()
  predicate one-dimension-domain-p)

(defvirtualclass ordered-domain (one-dimension-domain) ()
  predicate ordered-domain-p)

(defvirtualclass not-ordered-domain (one-dimension-domain) ()
  predicate not-oredered-domain-p)

(defclass set-domain (not-ordered-domain)
  ((setof initform () initarg setof reader setof writer set-setof))
  constructor (make-set-domain (setof))
  predicate set-domain-p )

(defvirtualclass numerical-domain (ordered-domain)
  ((min-boundary initform () initarg min-boundary reader min-boundary
                 writer set-min-boundary)
   (max-boundary initform () initarg max-boundary reader max-boundary
                 writer set-max-boundary))
  predicate numerical-domain-p)

(defclass real-domain (numerical-domain)
  ((step-value initform 0.5 initarg step-value reader step-value
               writer set-step-value))
  constructor (make-real-domain (min-boundary max-boundary))
  constructor (make-step-real-domain (min-boundary max-boundary step-value))
  predicate real-domain-p )

(defclass integer-domain (numerical-domain) ()
  constructor (make-integer-domain (min-boundary max-boundary))
  predicate integer-domain-p )

(defclass list-domain (ordered-domain)
  ((listof initform () initarg listof reader listof writer set-listof))
  constructor (make-list-domain (listof))
  predicate list-domain-p )
```

(PRINT-TO-STRING-DOMAIN *domain***)** *[fonction générique]*

- Imprime dans une chaîne les caractéristiques du domaine *domain*.
- Arguments :
  domain un domaine
- Valeur de retour : une chaîne de caractères

(IS-ELEMENT *domain expression***)** *[fonction générique]*

- Indique si l'expression *expression* est un élément du domaine *domain*.
- Arguments :
  domain un domaine
  expression une expression lisp
- Valeur de retour : booléen

(EQUAL-VALUE *domain expression1 expression2***)** *[fonction générique]*

- Indique si les deux expressions *expression1* et *expression2* sont égales.
- Arguments :
  domain un domaine
  expression1 une expression élément du domaine *domain*
  expression2 une expression élément du domaine *domain*
- Valeur de retour : booléen

(MINUS *domain expression1 expression2***)** *[fonction générique]*

- Calcule la différence entre les deux expressions *expression1* et *expression2*.
- Arguments :
  domain un domaine ordonné
  expression1 une expression élément du domaine *domain*
  expression2 une expression élément du domaine *domain*
- Valeur de retour : un réel.

(LOWER-THAN *ordered-domain expression1 expression2***)** *[fonction générique]*

- Indique si l'expression *expression1* est plus petite que *expression2*.
- Arguments :
  domain un domaine
  expression1 une expression élément du domaine *domain*
  expression2 une expression élément du domaine *domain*
- Valeur de retour : booléen

(LOWER-OR-EQUAL *ordered-domain expression1 expression2***)** *[fonction générique]*

- Indique si l'expression *expression1* est plus petite ou égale à *expression2*. Les deux expressions doivent appartenir au domaine *domain*.
- Arguments :
  domain un domaine
  expression1 une expression lisp
  expression2 une expression lisp

◇ Valeur de retour : booléen

`(DEFSYMBOL` *name super*`)` [*macro*]

◇ Déclare un symbole dans une hiérarchie de symboles.

◇ Arguments :

name	nom du symbole
super	père de symbole

◇ Valeur de retour : le symbole (une classe)

`(THE-ELEMENT` *domain expression*`)` [*fonction générique*]

◇ Transforme une expression en un élément du domaine *domain*.

◇ Arguments :

domain	un domaine hiérarchique
expression	une expression

◇ Valeur de retour : un élément du domaine *domain* ou () si la transformation n'est pas possible.

`(FATHER` *domain expression*`)` [*fonction générique*]

◇ Retourne le père de l'élément *expression* du domaine *domain*

◇ Arguments :

domain	un domaine hiérarchique
expression	une expression élément du domaine *domain*

◇ Valeur de retour : un élément de *domain* ou () si pas de père

`(SONS` *domain expression* `)` [*fonction générique*]

◇ Retourne les fils de l'élément *expression* du domaine *domain*

◇ Arguments :

domain	un domaine hiérarchique
expression	une expression élément du domaine *domain*

◇ Valeur de retour : une liste d'éléments de *domain* ou () s'il n'y a pas de fils.

`(GET-TREE-NAME` *domain*`)` [*fonction générique*]

◇ Retourne l'image de la hiérarchie du domaine *domain*, codée dans une liste où le `car` est un noeud et le `cdr` la liste des fils de ce noeud.

◇ Arguments :

domain	un domaine hiérarchique

◇ Valeur de retour : une liste

`(SUBNODE-P` *domain expression1 expression2*`)` [*fonction générique*]

◇ Indique si l'expression *expression1* est un sous-élément de *expression2*.

◇ Arguments :

domain	un domaine hiérarchique
expression1	une expression élément du domaine *domain*
expression2	une expression élément du domaine *domain*

◇ Valeur de retour : booléen

(`MSCA` *domain expression1 expression2*) [*fonction générique*]

◇ Calcule le *MSCA* (cf section 3.3.3) des deux expressions *expression1* et *expression2*.

◇ Arguments :

domain	un domaine hiérarchique
expression1	une expression élément du domaine *domain*
expression2	une expression élément du domaine *domain*

◇ Valeur de retour : un élément du domaine *domain*

(`NUMBER-OF-LINKS` *domain expression1 expression2*) [*fonction générique*]

◇ Calcule le nombre d'arcs entre les deux expressions *expression1* et *expression2*.

◇ Arguments :

domain	un domaine hiérarchique
expression1	une expression élément du domaine *domain*
expression2	une expression élément du domaine *domain*

◇ Valeur de retour : un entier ou () s'il n'y a pas de chemin direct entre les deux expressions.

D.1.4 Gestion des restrictions

Nous présentons les différentes classes de restrictions.

```
(defclass restriction () ()
  predicate restriction-p)

(defvirtualclass fuzzy-set (restriction) ()
  predicate fuzzy-set-p)

(defclass enum-fuzzy-set (fuzzy-set)
  ((enum initform () initarg enum reader enum writer set-enum))
  constructor (make-enum-fuzzy-set (enum)))

(defclass interpolated-enum-fuzzy-set (enum-fuzzy-set) ()
  constructor (make-interpolated-enum-fuzzy-set (enum)))

(defclass trapeze-fuzzy-set  (fuzzy-set)
  ((min-trapeze initform 0 initarg min-trapeze reader min-trapeze
                writer set-min-trapeze)
   (max-trapeze initform 0 initarg max-trapeze reader max-trapeze
                writer set-max-trapeze)
   (tolerance-min initform 0 initarg tolerance-min
                  reader tolerance-min writer set-tolerance-min)
   (tolerance-max initform 0 initarg tolerance-max
                  reader tolerance-max writer set-tolerance-max))
   constructor (make-trapeze-fuzzy-set (tolerance-min min-trapeze
```

```
                                                    max-trapeze tolerance-max)))

(defclass lambda-trapeze-fuzzy-set  (trapeze-fuzzy-set)
  ((lambda-min initform 0 initarg lambda-min reader lambda-min
               writer set-lambda-min)
   (lambda-max initform 0 initarg lambda-max reader lambda-max
               writer set-lambda-max))
  constructor (make-lambda-trapeze-fuzzy-set (tolerance-min min-trapeze
                                              max-trapeze tolerance-max
                                              lambda-min lambda-max)))
```

(`ENUM-INTERPOLATION` *domain restriction expression*) [*fonction générique*]

- ⋄ Fonction d'interpolation pour une restriction de type ensemble flou discret interpolé.
- ⋄ Arguments :
  - *domain* un domaine ordonné
  - *restriction* un ensemble flou interpolé
  - *expression* une expression élément du domaine *domain*
- ⋄ Valeur de retour : valeur de l'interpolation, un réel entre 0 et 1

(`MIN-BOUNDARY-INTERPOLATION` *restriction expression*) [*fonction générique*]

- ⋄ Fonction d'interpolation pour la tolérance gauche d'un ensemble flou de type trapèze.
- ⋄ Arguments :
  - *restriction* un ensemble de type trapèze
  - *expression* un réel entre 0 et 1
- ⋄ Valeur de retour : valeur de l'interpolation, un réel entre 0 et 1

(`MAX-BOUNDARY-INTERPOLATION` *restriction expression*) [*fonction générique*]

- ⋄ Fonction d'interpolation pour la tolérance droite d'un ensemble flou de type trapèze.
- ⋄ Arguments :
  - *restriction* un ensemble de type trapèze
  - *expression* un réel entre 0 et 1
- ⋄ Valeur de retour : valeur de l'interpolation, un réel entre 0 et 1

(`RESTRICTION-VALUE` *domain restriction expression*) [*fonction générique*]

- ⋄ Calcul de la fonction caractéristique d'une restriction (ensemble flou).
- ⋄ Arguments :
  - *domain* un domaine
  - *restriction* une restriction applicable sur le domaine *domain*
  - *expression* une expression élément du domaine *domain*
- ⋄ Valeur de retour : un réel entre 0 et 1

(`NORMALIZE` *restriction*) [*fonction générique*]

- ◇ Normalise un ensemble flou discret (le maximum est ramené à 1).
- ◇ Arguments :
  restriction une restriction de type ensemble flou discret
- ◇ Valeur de retour : -

(`CHECK-RESTRICTION` *domain restriction*) [*fonction générique*]

- ◇ Indique si la restriction *restriction* est applicable sur le domaine *domain*.
- ◇ Arguments :
  domain un domaine
  restriction une restriction
- ◇ Valeur de retour : booléen

(`PRINT-TO-STRING-RESTRICTION` *restriction*) [*fonction générique*]

- ◇ Imprime dans une chaîne les informations d'une restriction.
- ◇ Arguments :
  restriction une restriction
- ◇ Valeur de retour : une chaîne de caractères

Nous avons deux restrictions prédéfinies : `*unknown*` pour une restriction inconnue et `*all-domain*` pour une restriction vide.

(`IS-UNKNOWN` *restriction*) [*fonction générique*]

- ◇ Indique si la restriction *restriction* est inconnue.
- ◇ Arguments :
  restriction une restriction
- ◇ Valeur de retour : un booléen

(`IS-ALL-DOMAIN` *restriction*) [*fonction générique*]

- ◇ Indique si la restriction *restriction* est vide.
- ◇ Arguments :
  restriction une restriction
- ◇ Valeur de retour : un booléen

D.1.5 Gestion des valeurs symboliques

La classe `symbol-values` permet de décrire une liste d'associations entre des symboles et des restrictions.

```
(defclass symbol-values ()
  ((symbol-list initform () initarg symbol-list
               reader symbol-list writer set-symbol-list))
  constructor (make-symbol-values (symbol-list))
  predicate symbol-values-p)
```

(**SET-SYMBOL-RESTRICTION** *symbol-values restriction symbol*) [*fonction générique*]

- ◇ Ajoute ou change l'association du symbole *symbol* avec la restriction *restriction* dans les valeurs symboliques *symbol-values*.
- ◇ Arguments :
  - *symbol-values* un objet représentant les valeurs symboliques
  - *restriction* une restriction
  - *symbol* un symbole
- ◇ Valeur de retour : -

(**REMOVE-SYMBOL-VALUE** *symbol-values symbol*) [*fonction générique*]

- ◇ Enlève la valeur symbolique du symbol *symbol* des valeurs symboliques *symbol-values* .
- ◇ Arguments :
  - *symbol-values* un objet représentant les valeurs symboliques
  - *symbol* un symbole
- ◇ Valeur de retour : -

(**GET-SYMBOLS** *symbol-values*) [*fonction générique*]

- ◇ Retoune l'ensemble des symboles.
- ◇ Arguments :
  - *symbol-values* un objet représentant les valeurs symboliques
- ◇ Valeur de retour : liste de symboles

(**GET-SYMBOL-VALUE** *symbol-values symbol*) [*fonction générique*]

- ◇ Retourne la restriction associée au symbole *symbol* dans les valeurs symboliques *symbol-values*.
- ◇ Arguments :
  - *symbol-values* un objet représentant les valeurs symboliques
  - *symbol* un symbole
- ◇ Valeur de retour : une restriction ou () s'il n'y a pas de valeur.

D.2 Module de recherche

D.2.1 Classes de contrôle

Nous présentons maintenant les différentes classes de contrôle.

```
(defvirtualclass control () ())

(defvirtualclass local-control (control) ())

(defclass prototype-control (local-control)
  ((compatibility-threshold initform 0.3 initarg compatibility-threshold
                            reader compatibility-threshold
                            writer set-compatibility-threshold)
```

```
   (incompatibility-threshold initform 1 initarg incompatibility-threshold
                              reader incompatibility-threshold
                              writer set-incompatibility-threshold)
   (priority initform 0 initarg priority reader priority writer set-priority)
   (comparaison initform 'inherited initarg comparaison
                reader comparaison writer set-comparaison)
   (all-indexes initform t initarg all-indexes reader all-indexes
               writer set-all-indexes))
  constructor (make-prototype-control (compatibility-threshold
                                       incompatibility-threshold
                                       priority
                                       comparaison all-indexes))
  predicate  prototype-control-p)

(defvirtualclass indexation-control (control) ())
(defclass prototype-indexation-control (control)
  ((default-control initarg default-control reader default-control
                    writer set-default-control)
   (specific-list initform () initarg specific-list reader specific-list
                  writer set-specific-list))
   constructor (make-prototype-indexation-control (default-control
                                                   specific-list)))
```

(SET-PROTOTYPE-CONTROL *pic prototype pc*) [*fonction générique*]

◇ Associe un contrôle spécifique pour le prototype *prototype*.

◇ Arguments :

pic	un contrôle d'indexation par classification
prototype	un prototype
pc	un contrôle de prototype

◇ Valeur de retour : -

(SET-TO-DEFAULT-CONTROL *pic prototype*) [*fonction générique*]

◇ Associe le contrôle par défaut pour le prototype *prototype*.

◇ Arguments :

pic	un contrôle d'indexation par classification
prototype	un prototype

◇ Valeur de retour : -

(GET-PROTOTYPE-CONTROL *pic prototype*) [*fonction générique*]

◇ Retourne le contrôle du prototype *prototype*.

◇ Arguments :

pic	un contrôle d'indexation par classification
prototype	un prototype

◇ Valeur de retour : un contrôle de prototype

D.2.2 Protocole d'indexation par classification floue

Nous présentons les fonctions génériques du protocole de classification floue avec leur comportement standard.

(`INDEXATION-BY-CLASSIFICATION` *pic prototype*) [*fonction générique*]

- Réalise l'indexation par classification en prenant comme racine le prototype *prototype*. Les fonctions génériques `CASE-CLASSIFICATION` puis `GET-PROTOTYPE-CASE-LIST` sont appelées.
- Arguments :
  pic un contrôle d'indexation par classification
  prototype un prototype
- Valeur de retour : une liste de cas

(`CASE-CLASSIFICATION` *pic prototype*) [*fonction générique*]

- Réalise la classification avec *prototype* comme racine.
- Arguments :
  pic un contrôle d'indexation par classification
  prototype un prototype
- Valeur de retour : une liste ponderée de prototypes

(`BEGIN-EXAMINATION-PROTOTYPE` *pic prototype*) [*fonction générique*]

- Cette fonction générique est appelée à chaque début d'examen d'un prototype et peut être spécialisée pour un comportement spécifique. Aucune action n'est exécutée dans le comportement standard.
- Arguments :
  pic un contrôle d'indexation par classification
  prototype un prototype
- Valeur de retour : -

(`SPECIALIZE-CASE` *pic case to-prototype*) [*fonction générique*]

- Spécialise le cas *case* vers le prototype *to-prototype*. Il y a création d'un nouveau cas de type *to-prototype* qui hérite des valeurs de *case*.
- Arguments :
  pic un contrôle d'indexation par classification
  case le cas à spécialiser
  to-prototype un prototype
- Valeur de retour : le nouveau cas

(`COMPLETE-CASE` *pic case*) [*fonction générique*]

- Complète la description d'un cas. Cette fonction appelle la fonction générique `COMPLETE-FIELD` pour tous les éléments de la liste donnée par `GET-PROTOTYPE-FIELDS`. Puis la fonction générique `COMPLETE-CHECK` est appelée.
- Arguments :

pic	un contrôle d'indexation par classification
case	le cas à compléter

- Valeur de retour : -

(`GET-PROTOTYPE-FIELDS` *pic case*) [*fonction générique*]

- Donne la liste des champs à compléter dans l'ordre de leur précédence.
- Arguments :

pic	un contrôle d'indexation par classification
case	le cas à compléter

- Valeur de retour : liste ordonnée des champs à examiner

(`COMPLETE-FIELD` *pic case field-name*) [*fonction générique*]

- Complète un champ. Le comportement standard s'occupe des interactions avec l'opérateur ainsi que du calcul des valeurs par défaut pour le champ, les index et les importances.
- Arguments :

pic	un contrôle d'indexation par classification
case	le cas à compléter
field-name	le nom du champ à compléter

- Valeur de retour : -

(`COMPLETE-CHECK` *pic case*) [*fonction générique*]

- Cette fonction est appelée lorsque le cas a été complété. Le comportement standard demande une vérification auprès de l'opérateur.
- Arguments :

pic	un contrôle d'indexation par classification
case	le cas à compléter

- Valeur de retour : -

(`COMPUTE-COMPATIBILITY-INCOMPATIBILITY` *pic case inherited-cp-icp*) [*fonction générique*]

- Calcule le couple compatibilité et incompatibilité. Cette fonction parcourt les index donnés par `GET-PROTOTYPE-INDEXES` et appelle `INDEX-COMPATIBILITY`, `INDEX-INCOMPATIBILITY`, `IS-INDEX-ACCEPTABLE`.

  Puis les fonctions génériques `COMPUTE-PROTOTYPE-COMPATIBILITY-INCOMPATIBILITY` et `IS-PROTOTYPE-ACCEPTABLE` sont appelées.
- Arguments :

pic	un contrôle d'indexation par classification
case	le cas à comparer
inherited-cp-icp	compatibilité et incompatibilité héritées

◇ Valeur de retour : (cp . icp) ou () si la comparaison n'est pas acceptable.

(`GET-PROTOTYPE-INDEXES` *pic case*) [*fonction générique*]

◇ Donne la liste des index à comparer.

◇ Arguments :
pic un contrôle d'indexation par classification
case le cas à comparer

◇ Valeur de retour : liste des noms des index

(`INDEX-COMPATIBILITY` *pic case index*) [*fonction générique*]

◇ Calcul de la compatibilité, cette fonction appelle `COMPUTE-INDEX-COMPATIBILITY`.

◇ Arguments :
pic un contrôle d'indexation par classification
case le cas à comparer
index l'index à comparer

◇ Valeur de retour : compatibilité

(`COMPUTE-INDEX-COMPATIBILITY` *pic domaine restriction value*) [*fonction générique*]

◇ Calcul de la compatibilité.

◇ Arguments :
pic un contrôle d'indexation par classification
domaine domaine de l'index
restriction restriction du prototype
value restriction de valeurs

◇ Valeur de retour : compatibilité

(`INDEX-INCOMPATIBILITY` *pic case index*) [*fonction générique*]

◇ Calcul de l'incompatibilité, cette fonction appelle `COMPUTE-INDEX-INCOMPATIBILITY`.

◇ Arguments :
pic un contrôle d'indexation par classification
case le cas à comparer
index l'index à comparer

◇ Valeur de retour : incompatibilité

(`COMPUTE-INDEX-INCOMPATIBILITY` *pic domaine restriction value*) [*fonction générique*]

◇ Calcul de l'incompatibilité.

◇ Arguments :
pic un contrôle d'indexation par classification
domaine domaine de l'index
restriction restriction du prototype
value restriction de valeurs

◇ Valeur de retour : incompatibilité

(`IS-INDEX-ACCEPTABLE` *pic case cp-icp*) [*fonction générique*]

- ◇ Accepte ou rejette la comparaison au niveau d'un index.
- ◇ Arguments :

pic	un contrôle d'indexation par classification
case	le cas à comparer
cp-icp	couple compatibilité et incompatibilité

- ◇ Valeur de retour : un booléen

(`COMPUTE-PROTOTYPE-COMPATIBILITY-INCOMPATIBILITY` *pic case inherited-cp-icp local-cp-icp*) [*fonction générique*]

- ◇ Calcule la compatibilité et l'incompatibilité globales
- ◇ Arguments :

pic	un contrôle d'indexation par classification
case	le cas à comparer
inherited-cp-icp	couple compatibilité et incompatibilité hérité
local-cp-icp	couple compatibilité et incompatibilité local

- ◇ Valeur de retour : couple compatibilité et incompatibilité global

(`IS-PROTOTYPE-ACCEPTABLE` *pic case cp-icp*) [*fonction générique*]

- ◇ Accepte ou rejette la comparaison au niveau du prototype.
- ◇ Arguments :

pic	un contrôle d'indexation par classification
case	le cas à comparer
cp-icp	couple compatibilité et incompatibilité global

- ◇ Valeur de retour : un booléen

(`COMPUTE-RESULT-LIST` *pic case old-list cp-icp*) [*fonction générique*]

- ◇ Calcule la liste des résultats.
- ◇ Arguments :

pic	un contrôle d'indexation par classification
case	le cas à comparer
old-list	l'ancienne liste de résultats
cp-icp	couple compatibilité et incompatibilité global ou ()

- ◇ Valeur de retour : nouvelle liste

(`COMPATIBILITY-TO-POSSIBILITY` *pic cp-icp*) [*fonction générique*]

- ◇ Transforme les coefficients de compatibilité et d'incompatibilité en un seul coefficient.
- ◇ Arguments :

pic	un contrôle d'indexation par classification
cp-icp	couple compatibilité et incompatibilité global

- ◇ Valeur de retour : réel entre 0 et 1

(`COMPUTE-PROTOTYPE-LIST` *pic case old-list cp-icp*) [*fonction générique*]

- Cette fonction est appelée pour mettre à jour la liste des prototypes à examiner lorsque la comparaison a été acceptée.
- Arguments :
  pic un contrôle d'indexation par classification
  case un cas
  old-list l'ancienne liste
  cp-icp couple compatibilité et incompatibilité global
- Valeur de retour : nouvelle liste

(`END-EXAMINATION-PROTOTYPE` *pic prototype*) [*fonction générique*]

- Cette fonction générique est appelée à chaque fin d'examen d'un prototype et peut être spécialisée pour un comportement spécifique. Aucune action n'est exécutée dans le comportement standard.
- Arguments :
  pic un contrôle d'indexation par classification
  prototype un prototype
- Valeur de retour : -

(`GET-PROTOTYPE-CASE-LIST` *pic prototype-list*) [*fonction générique*]

- Récupère les cas à partir de la liste pondérée des prototypes.
- Arguments :
  pic un contrôle d'indexation par classification
  prototype-list liste pondérée des prototypes
- Valeur de retour : liste de cas

Annexe E

Exemple de fonctionnement du système

Dans cette annexe, nous allons détailler un exemple de fonctionnement de la partie implantée de notre système. Nous commençerons par présenter en quoi consiste l'exemple, puis nous décrirons les hiérarchies de prototypes et enfin nous pourrons lancer l'indexation par classification floue.

E.1 Présentation de l'exemple

Nou voulons concevoir une voiture et pour simplifier le problème nous considérons que les contraintes ne peuvent porter que sur le prix de vente unitaire et les performances. Dans un contexte CBR, nous définissons qu'un *cas* correspond à une *voiture*, une voiture pouvant être représentée par ses propres caractéristiques et par les contraintes qu'elle résoud.

Ainsi, pour résoudre le problème, il s'agit dans la phase de recherche, de trouver l'ancienne voiture (l'ancien cas) qui est la plus proche du problème. Mais nous n'avons implanté que l'indexation par classification floue du problème, le résultat sera donc une liste de cas appartenant aux prototypes de problèmes.

E.2 Hiérarchies de prototypes

E.2.1 Présentation

Pour avoir une représentation structurée des cas nous définissons deux hiérarchies : une hiérarchie pour décrire les critères de performance et une autre hiérarchie pour les voitures. Nous présentons graphiquement ces deux hiérarchies dans les figures E.1 et E.2. Le but de la hiérarchie des performances est de classer les performances d'abord par rapport à la vitesse maximale, puis si la vitesse maximale est moyenne, la classification se précise grâce à de nouveaux paramètres (caractéristiques d'accélération).

Les données qui nous ont permis de construire ces hiérarchies sont issues du magasine "L'action automobile" [Aut93].

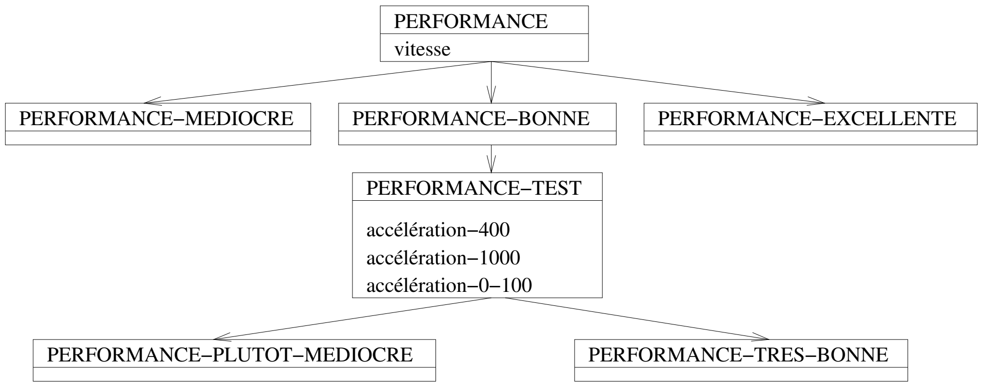

Figure E.1 : Hiérarchie de prototypes pour les performances.

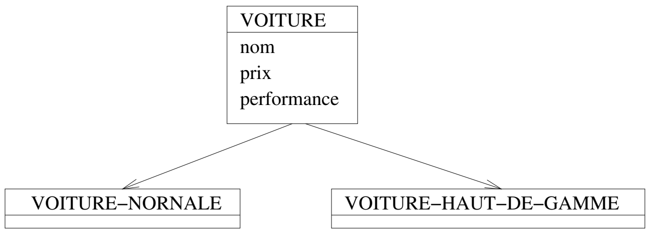

Figure E.2 : Hiérarchie de prototypes pour les voitures.

E.2.2 Hiérarchie de performances

Nous présentons la définition complète de la hiérarchie de prototypes pour les performances.

```
(defprototype performance ()
  ((vitesse slot-class fuzzy-field
            slot-initargs
               (domain (make-real-domain 150 250)
                       question "vitesse maximale de la voiture?"
                       documentation "vitesse maximale"
                       symbolic-values (make-symbol-values
                                 `((rapide . ,(make-t      rapeze-fuzzy-set
                                                210 220 250 250))
                                   (moyen . ,(make-trapeze-fuzzy-set
                                               195 200 205 215))
                                   (peu-rapide . ,(make-trapeze-fuzzy-set
                                                    150 150 190 200))))
               default-index t
               default-importance 1
               ask t))))

(defprototype performance-mediocre (performance)
  ((vitesse slot-class fuzzy-field
            slot-initargs (restriction (make-trapeze-fuzzy-set
                                         150 150 190 200)))))

(defprototype performance-excellente (performance)
  ((vitesse slot-class fuzzy-field
            slot-initargs (restriction (make-trapeze-fuzzy-set
                                         210 220 250 250)))))

(defprototype performance-bonne (performance)
  ((vitesse slot-class fuzzy-field
            slot-initargs (restriction (make-trapeze-fuzzy-set
                                         195 200 205 215)))))

(defprototype performance-test (performance-bonne)
  ((acceleration-400 slot-class fuzzy-field
                     slot-initargs
                        (domain (make-real-domain 10 20)
                         question "acceleration sur 400m ?"
                         documentation "acceleration 400m"
                         default-index t
                         default-importance 0.8
                         ask t
                         precedence 2))
```

```
   (acceleration-1000 slot-class fuzzy-field
                      slot-initargs
                        (domain (make-real-domain 25 35)
                         question "acceleration sur 1000m ?"
                         documentation "acceleration 1000m"
                         default-index t
                         default-importance 0.9
                         ask t
                         precedence 3))
   (acceleration-0-100 slot-class fuzzy-field
                       slot-initargs
                        (domain (make-real-domain 5 15)
                         question "0 a 100 km/h ?"
                         documentation "0 a 100 km/h"
                         default-index t
                         default-importance 1
                         ask t
                         precedence 1))))


(defprototype performance-plutot-mediocre (performance-test)
  ((acceleration-400 slot-class fuzzy-field
                     slot-initargs (restriction (make-trapeze-fuzzy-set
                                                 17 17.2 20 20)))
   (acceleration-1000 slot-class fuzzy-field
                      slot-initargs (restriction (make-trapeze-fuzzy-set
                                                 30 31 35 35)))
   (acceleration-0-100 slot-class fuzzy-field
                       slot-initargs (restriction (make-trapeze-fuzzy-set
                                                   9.5 10 15 15)))))

(defprototype performance-tres-bonne (performance-test)
  ((acceleration-400 slot-class fuzzy-field
                     slot-initargs (restriction (make-trapeze-fuzzy-set
                                                 10 10 15 16)))
   (acceleration-1000 slot-class fuzzy-field
                      slot-initargs (restriction (make-trapeze-fuzzy-set
                                                 25 25 27 29)))
   (acceleration-0-100 slot-class fuzzy-field
                       slot-initargs (restriction (make-trapeze-fuzzy-set
                                                   5 5 8 9)))))
```

E.2.3 Hiérarchie de voitures

Nous présentons également la définition complète de la hiérarchie de prototypes pour les voitures.

```
(defprototype voiture ()
  ((nom slot-class field)
   (prix slot-class fuzzy-field
         slot-initargs
            (domain (make-step-real-domain 50000 200000 1000)
             question "prix de la voiture?"
             documentation "prix"
             symbolic-values (make-symbol-values
                              `((chere . ,(make-trapeze-fuzzy-set
                                           130000 140000 200000 200000))
                                (moyen . ,(make-trapeze-fuzzy-set
                                           80000 100000 125000 135000))
                                (peu-chere . ,(make-trapeze-fuzzy-set
                                               50000 50000 70000 85000))))
             default-index t
             ask t
             precedence 1))
   (performance slot-class fuzzy-field
                slot-initargs
                   (domain (make-prototype-domain 'performance)
                    question "---- performance ----"
                    ask t))))


(defprototype voiture-haut-de-gamme (voiture)
  ((prix slot-initargs
          (restriction (make-trapeze-fuzzy-set 120000 130000 200000 200000)))
   (performance slot-initargs
          (restriction (make-enum-fuzzy-set '((performance-excellente . 1)))))))

(defprototype voiture-normale (voiture)
  ((prix slot-initargs
          (restriction (make-trapeze-fuzzy-set 90000 100000 130000 140000)))
   (performance slot-initargs
          (restriction (make-enum-fuzzy-set '((performance-bonne . 1)
                                               (performance-mediocre . 1)))))))
```

E.3 Indexation par classification floue

Nous allons d'abord montrer comment créer un objet de contrôle pour paramétrer l'indexation par classification floue. Puis nous détaillerons les différentes interactions du système avec l'opérateur, et enfin nous présenterons deux exemples.

E.3.1 Création des objets de contrôle

Il faut créer un objet de type `prototype-indexation-control` qui est constitué du contrôle par défaut des prototypes et d'une liste d'association spécifique.

```
(defdynamic pic (make-prototype-indexation-control
            (make-instance 'prototype-control 'compatibility-threshold 0.4)
            ()))
```

Dans notre exemple, la liste d'association spécifique est vide, et nous construisons un contrôle de prototypes avec un seuil de compatibilité de 0,4.

E.3.2 Dialogue avec l'opérateur

Le système attend une réponse de l'utilisateur dans deux cas : premièrement lorsqu'une valeur est à demander, et deuxièmement pour choisir les index et leur importance. Pour les tests, le système informe également l'opérateur sur le déroulement du processus.

Demande de valeurs

Un message de ce type apparaît à l'écran :

```
vitesse maximale de la voiture?
domain : real range 150....250.
default : inconnu
symbolic values : (rapide moyen peu-rapide)
>>?
```

Nous voyons que le domaine du champ est indiqué ainsi que la valeur par défaut et les valeurs symboliques. L'opérateur a alors trois possibilités pour répondre, il peut taper :

- le mot-clé `default` pour que la valeur par défaut soit prise comme valeur,
- un des symboles élément de la liste des valeurs symboliques,
- une expression lisp qui sera évaluée.

La troisième solution permet de spécifier sa propre restriction de valeurs qui peut être vue comme des préférences ou de l'imprécision dans l'énoncé du problème.

Choix des index

Une fois que toutes les valeurs des champs d'un prototype ont été complétées, un message de ce type apparaît :

```
name            index importance documentation
--------------------------------------------------
prix            t     1          prix
performance     ()    ()         performance de la voiture
>>?
```

Ce tableau donne la liste des champs qui peuvent être sélectionnés comme index avec l'indication s'ils ont déjà été choisis (par des mécanismes automatiques, ou par un choix d'index statiques). Dans l'affirmative, l'importance de l'index est également indiquée. Les phrases de documentation écrites dans les prototypes sont aussi affichées. Il est alors possible à l'opérateur de répondre de plusieurs manières, il peut taper :

- le mot-clé `ok` pour accepter les choix,
- l'expression `(`<*nom-champ*>`)` ou `(`<*nom-champ*> `())` pour désélectionner un index,
- l'expression `(`<*nom-champ*> `t)` ou `(`<*nom-champ*> `t` <*importance*>`)` pour sélectionner un index avec une importance par défaut de 1.

Dans les deux dernières possibilités, le tableau est réaffiché. Ainsi pour continuer, il faut obligatoirement répondre avec le mot clés `ok`.

Déroulement du processus

Le système affiche trois types de messages :

- `specialize to` <*nom-de-prototype*> pour avertir de l'examen d'un prototype,
- `index` <*nom-de-l'index*> <*(compatibilité,incompatibilité)*> pour indiquer les résultats de la comparaison d'un index,
- `prototype` <*nom-du-prototype*> <*(compatibilité,incompatibilité)*> pour indiquer les résultats de la comparaison d'un prototype.

E.3.3 Premier problème

Nous lançons la recherche par la fonction générique `case-classification` et non par la fonction `indexation-by-classification` car nous n'avons pas réalisé l'étape de stockage, et nous ne sommes pas en mesure de récupérer réellement les cas. Toutefois, si le stockage était réalisé, la récupération des cas serait triviale.

```
? (case-classification (dynamic pic) (find-prototype 'voiture))
```

Le système examine alors les champs du prototype `voiture` et, comme le premier champ à traiter est défini sur le domaine hiérarchie de prototypes de racine `performance`, il parcourt cette hiérarchie en premier.

```
specialize to voiture
---- performance ----
specialize to performance
vitesse maximale de la voiture?
domain : real range 150....250.
default : inconnu
symbolic values : (rapide moyen peu-rapide)
>>?rapide
```

Nous avons répondu à la première question grâce au symbole `rapide`, ce qui signifie qu'il s'agit de concevoir une voiture rapide. Il faut ensuite choisir les index, étant donné que `vitesse` est un index par défaut.

```
name            index importance documentation
---------------------------------------------------
vitesse         t     1          vitesse maximale
>>?ok

index vitesse(1,0)
prototype performance(1,0)

specialize to performance-bonne
index vitesse(0.25,1)
prototype performance-bonne(0.25,1)

specialize to performance-excellente
index vitesse(1,0.5)
prototype performance-excellente(1,0.5)

specialize to performance-mediocre
index vitesse(0,1)
prototype performance-mediocre(0,1)
```

Le système revient alors au prototype `voiture`.

```
prix de la voiture?
domain : real range 50000....200000.
default : inconnu
symbolic values : (chere moyen peu-chere)
>>?chere
```

Nous choisissons alors les index, le prix étant un index par défaut.

```
name            index importance documentation
---------------------------------------------------
prix            t     1          prix
performance     ()    ()
>>?(performance t)
```

```
name            index importance documentation
---------------------------------------------------
prix            t     1          prix
performance     t     1
>>?ok

index prix(1,0)
index performance(1,0)
prototype voiture(1,0)

specialize to voiture-haut-de-gamme
index prix(1,0)
index performance(0.646446609407,0)
prototype voiture-haut-de-gamme(0.646446609407,0)

specialize to voiture-normale
index prix(0.5,1)
index performance(0,0.646446609407)
prototype voiture-normale(0,1)

= ((#<voiture-normale #@00396780> . 0)
   (#<voiture-haut-de-gamme #@00397640> . 0.75))
```

Le problème s'est donc classé dans le prototype `voiture-haut-de-gamme`. Il suffirait donc de récupérer les cas qui correspondent à ce prototype pour effectuer l'indexation par classification floue complètement.

E.3.4 Second problème

De la même manière, on lance la classification.

```
? (case-classification (dynamic pic} (find-prototype 'voiture))
specialize to voiture
---- performance ----
specialize to performance
vitesse maximale de la voiture?
domain : real range 150....250.
default : inconnu
symbolic values : (rapide moyen peu-rapide)
>>?moyen
name            index importance documentation
---------------------------------------------------
vitesse         t     1          vitesse maximale
>>?ok
index vitesse(1,0)
prototype performance(1,0)
```

```
specialize to performance-bonne
index vitesse(1,0.5)
prototype performance-bonne(1,0.5)
```

Cette fois-ci, le problème est suffisamment compatible avec le prototype `performance-bonne` pour descendre dans `performance-test`.

```
specialize to performance-test
0 a 100 km/h ?
domain : real range 5....15.
default : inconnu
symbolic values : ()
>>?(make-trapeze-fuzzy-set 9 9.5 10 10.5)

acceleration sur 400m ?
domain : real range 10....20.
default : inconnu
symbolic values : ()
>>?(make-trapeze-fuzzy-set 16.5 17 17 17.5)

acceleration sur 1000m ?
domain : real range 25....35.
default : inconnu
symbolic values : ()
>>?(make-trapeze-fuzzy-set 29.5 30 30 31)
```

Aprè avoir répondu aux questions, on valide le choix par défaut des index.

```
name            index importance documentation
---------------------------------------------------
acceleration-400 t     0.8        acceleration 400m
acceleration-1000 t     0.9        acceleration 1000m
acceleration-0-100 t     1          0 a 100 km/h
>>?ok
index acceleration-400(1,0)
index acceleration-1000(1,0)
index acceleration-0-100(1,0)
prototype performance-test(1,0.5)

specialize to performance-tres-bonne
index acceleration-400(0.2,0)
index acceleration-1000(0.1,0.5)
index acceleration-0-100(0,0)
prototype performance-tres-bonne(0,0.5)

specialize to performance-plutot-mediocre
```

```
index acceleration-400(0.2,0)
index acceleration-1000(0.5,0.5)
index acceleration-0-100(1,0)
prototype performance-plutot-mediocre(0.2,0.5)

specialize to performance-excellente
index vitesse(0.25,1)
prototype performance-excellente(0.25,1)

specialize to performance-mediocre
index vitesse(0.3,1)
prototype performance-mediocre(0.3,1)
```

On revient alors au prototype `voiture`.

```
prix de la voiture?
domain : real range 50000....200000.
default : inconnu
symbolic values : (chere moyen peu-chere)
>>?moyen

name            index importance documentation
---------------------------------------------------
prix            t     1          prix
performance     ()    ()
>>?(performance t)

name            index importance documentation
---------------------------------------------------
prix            t     1          prix
performance     t     1
>>?ok

index prix(1,0)
index performance(1,0)
prototype voiture(1,0)

specialize to voiture-normale
index prix(1,0.65)
index performance(0,0)
prototype voiture-normale(0,0.65)

specialize to voiture-haut-de-gamme
index prix(0.7,1)
index performance(0,0.646446609407)
prototype voiture-haut-de-gamme(0,1)
```

```
= ((#<voiture-haut-de-gamme #@0039AAB0> . 0)
   (#<voiture-normale #@0038F7A0> . 0)
   (#<voiture #@003A70A0> . 1.))
```

Cette fois-ci, le problème ne s'est classé ni dans `voiture-haut-de-gamme` ni dans `voiture-normale`. Ainsi, nous pouvons retourner les cas qui sont associés au prototype `voiture`. Notons que l'ensemble des cas associés à `voiture` n'est pas égal à l'union des ensembles des cas associés à `voiture-normale` et à `voiture-haut-de-gamme`; au contraire, l'ensemble des cas associés à `voiture` est l'ensemble des cas qui n'ont pu être associés à `voiture-haut-de-gamme` ou à `voiture-normale`.

Références

[And92] V. Andrès. Simcair : un système d'interrogation floue multi-critère pour base de données floues. In *Les applications des ensembles flous*, Nimes, 1992.

[Aut93] L'Action Automobile. 45 sportives à moins de 150 000f, 1993. Hors serie n23.

[BFOS84] L. Breiman, J. H. Freidman, R. A. Olshen, and C. J. Stone. *Classification and Regression Tree.* Wadsworth and Brooks, Pacific Grove, California, 1984.

[Bor90] M.F. Borderes. Traitement des connaissances incertaines et imprécises, 1990.

[BZ92] T. Bardasz and I. Zeid. Dejavu: a case-based reasoning designer's assistant shell. In J. S. Gero, editor, *Artificial Intelligence in Design'92*, pages 477–496, 1992.

[CDG+93] C.H. Coulon, H. Durschke, W. Grather, B. Linowski, and W. Oertel. Fallverwaltung und fallretrieval: Implementierungen und tests. Fabel-report 7, GMD, Sankt Augustin, February 1993.

[CSFC93] P. Cunningham, B. Smyth, D. Finn, and E. Cahill. Retrieval issues in real-world CBR appliations: How far can we go with discrimination nets? In *Proc. of the IJCAI'93 workshop on Reuse of designs - an interdisciplinary approach*, Chambery, France, 1993.

[DDT88] H. Prade D. Dubois and C. Testemale. Weighted fuzzy pattern matching, 1988.

[DHR89] F. Daube and B. Hayes-Roth. A case-based mechanical redesign system. In *Proc. of the 11 th IJCAI*, pages 100–109, Detroit, MI, 1989.

[DP87] D. Dubois and H. Prade. *Théorie des possibilités : application à la représentation des connaissances en informatique.* Masson, 1987.

[Gen89] D. Gentner. The mechanism of analogical learning. In S. Vosniadou and A. Ortony, editors, *Similarity and analogical reasoning.* Cambridge University Press, New York, 1989.

[Ha91a] J.P. Haton and al. *Le raisonnement en intelligence artificielle*, chapter 9, pages 366–385. InterEditions, Paris, 1991.

[Ha91b] J.P. Haton and al. *Le raisonnement en intelligence artificielle*, chapter 4, pages 160–175. InterEditions, Paris, 1991.

[Har91] Paul Harmon. Case-based reasoning II. *Intelligent Software Strategies*, VII(12):1–9, 1991.

[HT89] K. J. Holyoak and P. R. Thagard. A computational model of analogical problems solving. In S. Vosniadou and A. Ortony., editors, *Similarity and analogical reasoning.* Cambridge University Press, New York, 1989.

[ILO87] ILOG. *CLASSIC : manuel de référence.* 2 avenue Galliéni, BP 85, 94253 Gentilly Cedex FRANCE, 1987.

[ILO92a] ILOG. *SMECI : Manuel de référence.* 2 avenue Galliéni, BP 85, 94253 Gentilly Cedex FRANCE, 1992.

[ILO92b] ILOG. *SMECI Module flou : manuel de l'Utilisateur et manuel de référence.* 2 avenue Galliéni, BP 85, 94253 Gentilly Cedex FRANCE, 1992.

[ILO93] ILOG. *IlogTalk : Manuel de référence (version 16.11 Beta).* 2 avenue Galliéni, BP 85, 94253 Gentilly Cedex FRANCE, 1993.

[Kea88] M. Keane. Where's the beef ? the absence of pragmatic factors in pragmatic theories of analogy. In *Proceedings of ECAI88*, Munich, August 1988.

[Kol83] J. Kolodner. Maintaining organization in a dynamic long-term memory. *Cognitive Science*, 7(4):243–280, 1983.

[Kol89] J. L. Kolodner. Judging which is the best case for a case-based reasoner. In Morgan Kaufman, editor, *DARPA 89:CBR Workshop*, pages 77–81, Holiday Inn, Pensacola Beach, Florida, May 1989.

[Kol91] J. Kolodner. Improving human decision makink through case-based decision aiding. *AI magazine*, 12(2):52–68, 1991.

[Kol92] J. Kolodner. An introduction to case-based reasoning. *Artificial Intelligence Review*, 6(1):3–34, 1992.

[KR89] J. Kolodner and C. Riesbeck. Cased-based reasoning (tutorial program). Eleventh International Joint Conference on Artificial Intelligence, August 1989.

[MA93] M. Manago and K.D. Althoff. A comparison between induction abd case-based reasoning. In *AVIGNON'93 International conference on Artificial Intelligence and their applications*, Avignon, France, 1993.

[Man89] M. Manago. Kate : intégration de techniques symboliques et numériques en apprentissage. In Cépaduès éditions, editor, *Induction symbolique et numérique à partir de données.* 111, rue Nicolas-Vauquelin, 31100 Toulouse, 1989.

[Mot93] S. Mott. Case-based reasoning: Market, applications, and fit with other technologies. *Expert Systems With Applications*, 6:97–104, 1993.

[New82] A. Newell. The knowledge level. *AI journal*, 18(2):82–127, 1982.

[Qui79] J. R. Quinlan. *Expert systems in the micro electronic age*, pages 168–201. Edinburgh University Press, 1979.

[RKW89] E. L. Rissland, J. Kolodner, and D. Waltz. Case-based resoning. In Morgan Kaufman, editor, *DARPA 89:CBR Workshop*, pages 1–13, Holiday Inn, Pensacola Beach, Florida, May 1989.

[RS89] C. K. Riesbeck and R. C. Schank. *Inside Case-Based Reasoning.* Erlbaum, Hillsdale, NJ, 1989.

[Sal89] S. Salotti. Flou et repr/'esentation centr/'ee objet pour raisonner par analog ie : le système floran. Rapport de recherche n 473, Unité Associée du CNRS UA 410, Mars 1989.

[Sap90] G. Saporta. *Probabilités, analyse des données et statistique.* Technip, 1990.

[Sch82] R. C. Schank. *Dynamic memory : a theory of reminding and learning in computers and people.* Cambridge University Press, Cambridge, 1982.

[Sla91] S. Slade. Case-based reasoning: A research paradigm. *AI magazine*, 12(1):42–55, 1991.

[SNC93] K. Sycara and D. Navin-Chandra. Case representation and indexing for innovative design reuse. In *Proc. of the IJCAI'93 workshop on Reuse of designs - an interdisciplinary approach*, Chambery, France, 1993.

[Sys92] Cognitive System. *ReMind : Developer's Reference Manual.* 220-230 Commercial St., Boston, MA 02109, 1992.

[Tro92] B. Trousse. Vers une véritable assistance logicielle à la conception. pages 95–114, january 1992.

[TV93] B. Trousse and W. Visser. Use of case-based reasoning techniques for intelligent computer-aided-design systems. In *International Conference on Systems, Man and Cybernetics*, Palais de l'Europe - Le Touquet - France, October 1993.

[VT93] W. Visser and B. Trousse. Reuse of designs : desperately seeking an interdisciplinary cognitive approach. In *Proc. of the IJCAI'93 workshop on Reuse of designs - an interdisciplinary approach*, Chambery, France, 1993.

[Wol89] J. Wolstencroft. Restructing, reminding and repair : What's missing from models of analogy? In *Proceedings of the Scandinavian Conference on AI*, Tempere, June 1989.

[Zad65] L.A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[Zad78] L.A. Zadeh. Fuzzy sets as a basis for a theory of possibility. *Fuzzy sets and systems*, 1:3–28, 1978.