THÈSE

PRESENTEE PAR

Jose-Alejandro QUINTERO GARCIA

POUR OBTENIR LE TITRE DE DOCTEUR DE

L'UNIVERSITE JOSEPH FOURIER - GRENOBLE 1

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

SPECIALITE INFORMATIQUE

==========

Parallélisation de la classification d'objets dans un modèle de connaissances multi-points de vue

==========

DATE DE SOUTENANCE : 30 JUIN 1993

COMPOSITION DU JURY :

M. Michel ADIBA (président)
M. Jean-Marc GEIB (rapporteur)
M. Christophe ROCHE (rapporteur)
M. Jean-Louis ROCH (examinateur)
M. François RECHENMANN (directeur)

THESE PREPAREE AU SEIN DU LABORATOIRE LIFIA/IMAG

A mon seul et dernier amour ...

Ojos grandes y profundos que miran
desconsolados cómo transcurre el mundo,
vibran de emoción con las ilusiones que forja nuestra mente,
lloran cuando ellos se posan sobre el ser amado,
y se pierden en una mirada postrera con la realidad,
ojos grandes y profundos, esos ojos son los tuyos.

AQ

Résumé

La classification est un mécanisme fondamental d'inférence de connaissance, car elle permet de résoudre un problème en le localisant dans une structure de catégories de problèmes connus et en regardant les solutions connues. Dans le modèle de connaissances TROPES, la classification des instances se fait dans une hiérarchie de classes manipulant différents points de vue d'un concept. Ce travail est consacré aux mécanismes parallèles pour la classification des instances dans une base de connaissances.

Notre démarche s'appuie d'une part sur l'étude de modèles parallèles de résolution de problèmes en intelligence artificielle. Parmi les modèles, nous trouvons les systèmes de tableau noir et les langages acteurs. D'autre part ce travail parle sur les architectures de systèmes répartis et de machines parallèles modernes. Cette approche sur deux fronts convergents nous permet de prendre en compte à la fois les progrès dans l'expression et l'utilisation du parallélisme, et les tendances et possibilités d'utilisation de modèles parallèles pour la résolution de problèmes.

Le résultat principal de cette étude est la conception d'un modèle parallèle pour la classification des instances dans un modèle de connaissances multi-points de vue tel que TROPES, et la réalisation d'un système de classification sur un système réparti en utilisant à la fois le modèle de tableau noir et le modèle de langage acteur.

Mots Clés
Intelligence artificielle, classification, système réparti, parallélisme, tableau noir, langage acteur.

Abstract
Classification is a fundamental knowledge inference mechanism; in fact, one solves a problem by classifying it in a structure of known problems, thus getting the best adapted known solution. In TROPES knowledge representation system, classification consists in finding the right place for an instance in different class structures corresponding to different view points of the instance's concept. This research in concerned with the parallel mechanisms for instance classification in knowledge representation systems, and particularly, the parallelisation of TROPES multi-perspective instance classification.
Our work is founded on a study on parallel solving problem models in artificial intelligence, such as blackboards ans agents as well as on leading tendencies in distributed systems architectures and parallel machines.
Basic contributions of this work are the design of a parallel model for classifying instances in a multi-perspective knowledge representation system and the development of a classification system on a distributed architecture based on a mixed blackboard - agents approach.

Key words
Artificial Intelligence, classification, distributed systems, parallelism, blackboards, actors.

Je suis très sensible à l'honneur que me fait M. Michel Adiba, professeur à l'Université Joseph Fourier, en acceptant la présidence de ce jury. Qu'il soit assuré de ma reconnaissance pour l'attention qu'il porte à cette thèse.
Je voudrais remercier M. Jean-Marc Geib et M. Christophe Roche pour avoir accepté de rapporter sur cette thèse, malgré tout leur travail. Je suis très sensible à l'intérêt qu'ils ont porté à mon travail ; qu'ils reçoivent tous mes remerciements.
Je remercie M. Jean-Louis Roch pour s'être intéressé à mon travail. Ses conseils et critiques avisés ont été précieux. Je lui suis très reconnaissant de s'être autant impliqué. Si je ne puis m'acquitter de la dette que j'ai envers lui, qu'il en trouve au moins ici le témoignage.
Je remercie Mme Brigitte Plateau pour le temps qu'elle a consacré à mon travail et le soutien scientifique qu'elle m'a apporté.
Je tiens à remercier M. François Rechenmann de m'avoir accueilli dans son équipe. Grâce à lui j'ai pu mener ce travail à son terme. Il m'a donné des idées, fait des suggestions, des commentaires et des critiques, il m'a relancé, encouragé et soutenu quand il le fallait. Qu'il reçoive tous mes remerciements.
Je remercie l'INRIA (France) et l'Université des Andes (Bogota-Colombie), en particulier Alberto Sarria, pour les financements de mes stages en France afin de finir cette thèse.
À Olga, Pierre et Arnie. Nous avons commencé le DEA ensemble, puis la thèse. Grâce à vous, il a été prouvé que l'on peut trouver de très bons amis n'importe où et n'importe quand, même pendant une thèse et même si l'on se trouve parfois loin, très loin. Les souvenirs que j'aurais de Grenoble seront plus ceux que j'ai partagés avec vous que ceux de la thèse.
À *Cécile*. Je te remercie pour tout le temps que tu as consacré dernièrement et pour toutes les traductions de mon "français-español" en français. Mais surtout, je te remercie de ton amitié, même si parfois je ne dis ou ne fais que des bêtises. Tu seras toujours dans mes souvenirs les plus agréables.
À Patrice. C'est un peu difficile d'exprimer les remerciements que j'ai envers toi pour toutes les conversations qu'on a eues, pour ton soutien, tes histoires, tes conseils. Simplemente muchísimas gracias.
À Jérôme. Pour toutes les corrections de mon document, ton aide pendant ma thèse, ainsi que ton soutien pendant mon DEA et ton amitié.
À Jérôme. Pour toutes les corrections de mon document, ton aide et ton amitié.
À Florence, Nina, Bruno, pour votre amitié. Et à tous les autres membres du projet SHERPA.
À Rodrigo, pour m'avoir aidé à écrire la partie formelle du chapitre 1 et pour le temps qu'il m'a consacré.
À Claudine Mérieux pour sa gentillesse.
A mi casita linda, a todos mis amigos y a los estudiantes de la Universidad de los Andes que de una u otra forma trabajaron conmigo.
À mon seul et dernier amour ...

Introduction....9

1. Le modèle de connaissances TROPES.... 11
1.1 Représentation des connaissances.... 11
1.1.1. Les objets 11
1.1.2. Les classes et les instances.... 11
1.1.3. Classes et sous-classes 12
1.1.4 Classification d'un objet.... 12
1.2. Description de TROPES.... 13
1.2.1. Les concepts 13
1.2.2. Les points de vue, les classes et les passerelles 13
1.2.3. Les attributs d'une classe.... 15
1.2.4. Appartenance d'un objet à une classe.... 16
1.3. Classification d'un objet dans TROPES.... 17
1.3.1. Classes "sûres" ou "impossibles".... 17
1.3.2. Point de vue actif vs. point de vue inactif 20
1.3.3. Algorithme séquentiel de classification dans TROPES 24
1.3.2.1. Obtention d'information 24
1.3.2.2. Appariement.... 25
1.3.3.3. Propagation de marques 26
1.3.3.4. Mise à jour et choix du point de vue.... 27
1.4. Correction de l'algorithme de classification 28
1.5 Conclusions.... 39

2. Modèle parallèle de TROPES 41
2.1. Classification en parallèle d'un objet dans TROPES.... 41
2.2. Algorithme de Classification en parallèle 44
2.2.1. Le choix d'un point de vue.... 45
2.2.2. Exécution en parallèle d'instances de la même procédure 47
2.2.2.1. Obtention d'information 48
2.2.2.2. Appariement.... 52
2.2.2.3. Propagation de marques 52
2.2.2.4. Mise à jour d'information.... 58
2.2.3. Exécution en parallèle d'instances de toutes les procédures.... 59
2.3 Conclusions.... 61

3. Modèles parallèles de résolution de problèmes en intelligence artificielle.... 63
3.1. Le "tableau noir".... 64
3.1.1. Historique.... 64
3.1.2. Métaphore.... 64
3.1.3. Le modèle des systèmes de "tableau noir" 65
3.1.3.1. Les sources de connaissances.... 65
3.1.3.2. Structure de données 67
3.1.3.3. Le contrôle.... 67
3.1.4. Outils 69
3.1.5. Parallélisation du modèle.... 70
3.1.5.1. Aspects parrallélisables dans le modèle de "tableau noir".... 70
3.1.5.2. Réalisation matérielle du modèle sur une machine parallèle ou système réparti 73
3.1.6. La mise en œuvre de DSBB : un "tableau noir" concurrent.... 75
3.2. Le modèle acteur.... 79
3.2.1. Historique.... 80
3.2.2. Définition du modèle.... 80
3.2.3. Description de LDBA : langage acteur réparti.... 81
3.2.3.1. Les primitives de LDBA 82
3.2.3.2. La création des acteurs 82
3.2.3.3. L'envoi de messages 83

3.2.3.4. Spécification d'un comportement de remplacement 84
3.2.3.5. Résumé des opérations..... 86
3.2.3.6. Les opérations éloignées..... 86
3.2.3.7 La mise en œuvre du LDBA..... 90
3.3. Les différentes mises en œuvre de l'algorithme parallèle de classification 96
3.4. Conclusions..... 97

4. Analogie entre le modèle TROPES et le modèle de tableau noir 99
4.1. Approche générale..... 99
4.2. La base de connaissances de TROPES et le tableau..... 100
4.2.1. Association entre le tableau noir et les bases de connaissances en TROPES. 101
4.2.2. Les concepts et le tableau noir..... 103
4.2.3. Les points de vue, les passerelles et le tableau noir..... 105
4.2.4. Les objets et le tableau noir 108
4.3. Les sources de connaissances et les opérations de TROPES..... 109
4.4. Le contrôle et TROPES..... 112
4.5. Conclusions..... 115

5. Analyses de complexité de l'algorithme parallèle de classification dans TROPES.... 117
5.1. Le modèle séquentiel..... 117
5.1.1. Complexité de la procédure séquentielle d'obtention d'information..... 117
5.1.2. Complexité de la procédure séquentielle d'appariement..... 119
5.1.3. Complexité de la procédure séquentielle de propagation de marques 120
5.1.4. Complexité de la procédure séquentielle de mise-à-jour..... 126
5.1.5. Complexité de la procédure séquentielle du choix du prochain point de vue..... 126
5.1.6. Complexité totale de l'algorithme de classification séquentielle 127
5.2. Complexité du modèle parallèle..... 128
5.2.1. Complexité de la procédure parallèle d'obtention l'information 132
5.2.2. Complexité de la procédure parallèle d'appariement..... 132
5.2.3. Complexité de la procédure parallèle de propagation de marques 133
5.2.4. Complexité de la procédure parallèle de mise-à-jour..... 135
5.2.5. Complexité totale de l'algorithme de classification parallèle 136
5.3. Analyse des deux modèles 137
5.4. Le modèle de Circuits..... 140
5.5 Conclusions..... 145

6. Mise en œuvre de l'algorithme parallèle de classification..... 147
6.1. Idée générale 147
6.2. Analyse du nombre de processus et de la communication..... 147
6.2.1. L'architecture 148
6.2.2. Les configurations 149
6.2.2.1. L'hypercube..... 150
6.2.2.2. La grille 150
6.2.2.3. Le tore..... 151
6.2.2.4. Le supernode..... 151
6.2.2.5. Le bus..... 151
6.2.3. La terminaison..... 152
6.2.4. La communication..... 152
6.2.5. Le coût de transmission 153
6.2.6. Le placement des tâches 154

6.2.7. Le stockage de données ……157
6.3. La mise en œuvre de l'algorithme parallèle de classification en utilisant l'analogie TROPES - Tableau noir……158
6.4 Conclusions……172

Conclusions ……173

Annexe 1 : Le système d'exploitation MACH……177

Annexe 2 : Parcours en parallèle d'un graphe ……181

Annexe 3 : Classification des Architectures parallèles……183

Annexe 4 : Syntaxe du langage de TROPES ……185

Annexe 5 : Mise en œuvre de LA CLASSIFICATION EN paralléle ……193

Blibliographie ……201

INTRODUCTION

La classification est un mécanisme fondamental d'inférence de connaissances, car elle permet de résoudre un problème en le localisant dans une structure de catégories de problèmes connus et en regardant les solutions qui leur sont attachées.

Néanmoins, le terme classification peut être entendu de différentes façons. La définition qui nous intéresse est la classification des instances ou objets dans une base de connaissances. Classer une instance consiste à trouver les classes auxquelles elle appartient. Cette procédure entraîne des comparaisons entre une classe et un objet particulier. L'opération de classification est une opération très importante sur une base de connaissance structurée en une hiérarchie de classes.

Dans le modèle de connaissances TROPES [MAR 89], une base de connaissances est composée d'un ou plusieurs concepts. Ces concepts sont complètement indépendants et déterminent une partition de l'univers du discours. Les objets qui appartiennent à un concept peuvent être vus selon différents points de vue. Un point de vue peut être défini en général comme une position conceptuelle de laquelle un observateur regarde un objet. Donc, l'univers est divisé en différents points de vue, chacun ayant ses propres classes et ses propres attributs. Ces points de vue forment des partitions différentes d'un même ensemble d'individus. Ainsi, au lieu de travailler avec un seul graphe de classes par concept, la base contient une structure hiérarchique de classes pour chaque point de vue de chaque concept. Un point de vue sur un concept est donc un arbre de classes dans lequel une classe ne peut pas avoir plus d'une sur-classe. La classe racine de ces points de vue, qui appartiennent au concept, est la classe décrivant l'ensemble complet des éléments du concept, car dans le premier niveau de la hiérarchie de classes, nous n'avons fait aucune partition de l'univers du concept. Quoiqu'il s'agisse du même ensemble d'individus, la racine des différents points de vue d'un concept peut être décrite, dans les différents points de vue, par des ensembles différents d'attributs. Donc, dans le modèle TROPES, la classification des instances se fait dans une hiérarchie de classes manipulant simultanément différents points de vue d'un concept.

D'autre part, un intérêt considérable existe actuellement pour l'exploitation du parallélisme; il se manifeste aussi bien dans la programmation des applications intrinsèquement parallèles que dans la conception d'architectures d'ordinateurs parallèles. La programmation parallèle devient un sujet très important dans tous les domaines de l'informatique, parmi lesquels l'intelligence artificielle.

La classification d'une instance dans le modèle TROPES est un problème hautement parallélisable. D'abord, l'instance peut être classifiée en parallèle dans tous les points de vue; dans ces points de vue, la procédure de classification entraîne une comparaison entre les classes qui appartiennent au point de vue et un objet particulier (l'instance). Cette comparaison peut être faite en parallèle. Or, la comparaison d'une instance avec une classe entraîne l'évaluation des différentes valeurs par rapport aux contraintes de tous les attributs de la classe. L'évaluation des différents attributs et l'évaluation des valeurs par rapport aux différents contraintes peuvent être faites en parallèle.

Notre démarche s'appuie d'une part sur l'étude de modèles parallèles de résolution de problèmes dans l'intelligence artificielle, en particulier les systèmes de tableau noir et les langages acteurs, et d'autre part sur les architectures de systèmes répartis et de machines parallèles modernes. Cette approche sur deux fronts convergents nous permet de prendre en compte à la fois les progrès dans l'expression et l'utilisation du parallélisme, et les tendances et possibilités d'utilisation de modèles parallèles pour la résolution de problèmes.

Le résultat principal de cette étude est la conception d'un modèle parallèle pour la classification des instances dans le modèle de connaissances multi-points de vue TROPES, et la réalisation d'un système de classification sur un système réparti en utilisant à la fois le modèle de tableau noir et le modèle de langage acteur.

Organisation du document

Dans le chapitre 1, nous présentons le modèle TROPES [MAR 89] et l'algorithme de classification pour ce modèle, qui ont été conçus au sein du projet SHERPA.

Le chapitre 2 est consacré la parallélisation du processus de classification des instances dans TROPES, où nous faisons une analyse du modèle et des différentes possibilités de parallélisme existant dans ce processus : nous montrons que la classification d'instances est un problème

hautement parallélisable. Finalement, nous présentons un algorithme parallèle pour la classification d'instances dans TROPES.
Dans le chapitre 3, nous présentons quelques modèles parallèles pour la résolution de problèmes en intelligence artificielle, parmi lesquels le modèle de "tableau noir", le modèle de langage "acteur" et le modèle "multi-agents". Dans les conclusions de ce chapitre, nous expliquons pourquoi le modèle de "tableau noir" a été choisi pour modéliser la classification en parallèle dans TROPES. Parmi ces raisons, nous pouvons mentionner par exemple certaines différences avec le modèle "acteur", dans la façon de résoudre un problème :

- Dans le modèle "acteur", un problème est résolu par délégation des tâches d'un acteur à un autre acteur, tandis que pour le modèle de "tableau noir" la résolution d'un problème est faite par coopération entre les différentes sources de connaissances.
- Dans le modèle "acteur" la communication entre les acteurs se fait par messages, tandis que pour le modèle de "tableau noir" elle se fait à travers le tableau.

Ces différences influent beaucoup pour la conception des algorithmes de contrôle. Par exemple, l'état global de la classification en un instant donné est beaucoup plus facile à déterminer avec le modèle de "tableau noir" qu'avec le modèle "acteur".
Dans le quatrième chapitre, nous présentons une façon de modéliser TROPES en utilisant le modèle de tableau noir, afin d'envisager plus facilement le parallélisme dans les différentes opérations qui existent sur une base de connaissances.
Dans le chapitre 5, nous faisons une analyse de la correction de l'algorithme en montrant par exemple que le choix d'un point de vue pour classer une instance n'influe pas dans le résultat final de la classification; néanmoins pour l'algorithme séquentiel, le choix d'un point de vue peut améliorer sa performance. Dans ce chapitre est aussi présentée une analyse de la complexité du processus de classification d'une instance dans TROPES. La complexité est présentée pour le modèle séquentiel de classification en utilisant le modèle RAM [AHO 74] et pour le modèle parallèle en utilisant le modèle CRCW COMMUNE PRAM [KAR 90]. Cependant, c'est le modèle EREW PRAM qui a été utilisé pour la mise en œuvre de l'algorithme parallèle, avec un coût de log(**p**) (où **p** est le nombre de processeurs) pour cette simulation. La comparaison des complexités des algorithmes séquentiel et parallèle fait apparaître les avantages de ce dernier. Pour l'algorithme parallèle, le nombre de points de vue d'un concept, de sous-classes d'une classe, d'attributs définis dans une classe et le nombre de contraintes définies pour une attribut n'ont aucune importance : la complexité est toujours la même; elle ne dépend que de la hauteur maximale du point de vue. C'est-à-dire que la complexité a été réduite en termes de nombre de points de vue, classes, etc.
Nous concluons avec des commentaires sur le modèle parallèle de classification et avec des perspectives et modifications envisagées.

1. LE MODELE DE CONNAISSANCES TROPES

Un modèle puissant de représentation des connaissances doit avoir les éléments nécessaires pour que l'expert puisse exprimer sa connaissance des concepts, des règles, des relations, etc. Il doit aussi pouvoir utiliser efficacement l'information qui a été donnée et raisonner avec elle pour résoudre des problèmes et pour acquérir de nouvelles connaissances [MAR 89]. Finalement, le modèle doit être capable d'expliquer le raisonnement suivi pour résoudre une tâche et de partager sa connaissance avec l'utilisateur [FIK 85]. De façon générale, un modèle de représentation de connaissances est composé de données qui représentent la connaissance et des mécanismes de raisonnement ou "opérations" sur ces données pour résoudre des problèmes.

1.1 Représentation des connaissances

Cette section décrit les éléments d'un modèle de représentation de connaissances déclaratives. Tout élément dans la base de connaissances est accessible directement et peut être exploité par des mécanismes d'inférence spécialisés tels que l'héritage, le filtrage ou la classification.

1.1.1. Les objets

Une entité, ou objet, du monde réel est une granule élémentaire d'information à laquelle est associé un nom. Bien que les objets soient des unités élémentaires, ils regroupent une grande quantité de connaissances. Toute l'information d'un objet est stockée sous forme d'attributs ou "slots". Un attribut d'une classe est une propriété que possèdent tous les individus de la classe. Chaque slot est décrit par un ensemble de facettes (contraintes) qui définit le type de l'attribut, les valeurs autorisées, la valeur par défaut, etc. Dans les modèles de classe/instance, il existe une différence entre l'objet décrivant un ensemble d'individus (classe) et l'objet qui décrit un individu particulier (instance). Une instance a une valeur attachée à chaque slot tandis que dans une classe, chaque slot a une description de l'ensemble des valeurs que peut prendre ce slot dans les instances de la classe.

Dans les modèles classe/instance tels que SHIRKA [REC 88], il existe une différence entre les schémas de classe et les schémas d'instance, les premiers stockant les contraintes sur l'attribut et les deuxièmes stockant sa valeur pour l'instance.

1.1.2. Les classes et les instances

Dans le modèle classe/instance, il y a deux sortes d'objets : les classes et les instances. Une instance est liée à une classe par le lien "est-un". Une classe peut être vue comme un "pattern" de ses membres. Par exemple dans SHIRKA, une classe a les mêmes attributs que ses instances. Dans la classe, ces attributs ont des facettes indiquant les valeurs qu'ils peuvent prendre pour les individus de l'ensemble décrit. Ainsi, à part la facette de typage qui définit le type de l'attribut, il peut y avoir des facettes restreignant l'intervalle ou le domaine des valeurs acceptables, ou des facettes donnant une valeur par défaut ou fixée pour l'attribut. Dans les instances chaque attribut a une valeur qui satisfait les contraintes de la classe.

SHIRKA utilise l'approche ensembliste stricte, dans laquelle une classe représente un ensemble d'individus décrit par un schèma, devant être respecté par toutes les instances de l'ensemble qu'elle définit. Par exemple: pour la classe "les voitures", une instance peut être I = "la fiat".

Désormais, les instances de la base de connaissances seront appelées **objets**.

1.1.3. Classes et sous-classes

Une classe permet de décrire un groupe d'objets semblables; cette ressemblance entre les objets d'une classe est d'autant moins grande que la classe impose peu de contraintes sur les propriétés des objets qui lui appartiennent.
Pour faire une classification des objets du monde, l'être humain part des catégories générales imposant peu de contraintes à ces objets et détermine ensuite des sous-catégories plus ou moins spécifiques en ajoutant de nouvelles contraintes aux contraintes existantes dans la catégorie générale [MAR 89].
Dans les bases de connaissances organisées en classes, il y a une spécialisation des classes générales en des classes plus spécifiques ou sous-classes. La relation entre une sous-classe et la classe plus générale qui la contient, sa sur-classe, se fait par le lien "sorte-de".
Une classe inférieure ne peut être une spécialisation d'une classe supérieure que si elle possède au moins les mêmes attributs que la classe supérieure, car tout élément de la sous-classe doit rester un élément de la sur-classe. L'affinement de la sur-classe par la sous-classe peut se faire par ajout des caractéristiques qui n'apparaissent pas dans la classe ou par restriction des domaines des valeurs possibles des attributs de la sur-classe. Par exemple : pour la classe "les moyens de locomotion" une sous-classe possible est "les voitures".

1.1.4 Classification d'un objet

Durant sa vie, l'être humain apprend une grande quantité de connaissances qu'il garde de façon structurée dans sa mémoire. Face à une situation inconnue, il cherche dans sa mémoire des situations similaires déjà vécues pour en tirer des conclusions. Ce raisonnement entraîne donc une étape de classification dans laquelle la nouvelle connaissance est comparée avec les apprentissages précédents et classée dans la structure.
La classification est une des activités les plus puissantes du raisonnement humain. Dans les différents domaines de la science nous trouvons des classifications de la connaissance propre au domaine; par exemple en biologie nous avons des taxonomies animales et végétales.
Dans le domaine de l'intelligence artificielle, Clancey [CLA 85] a montré que la plupart des systèmes experts incluent des étapes de classification.

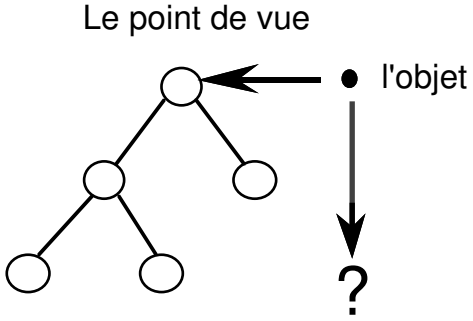


Figure 1.1 : l'objet appartient déjà à la classe racine. Le processus de classification cherche à descendre l'objet vers les classes les plus basses de la hiérarchie.

Néanmoins, le terme classification peut être entendu de différentes façons. La définition qui nous intéresse est la classification des objets dans une base de connaissances. Classer un objet I est trouver les classes auxquelles il appartient (figure 1.1). Au début, l'objet I appartient à la classe racine. La classification consiste à faire descendre l'objet dans les sous-classes de cette classe racine. L'opération de classification est une manipulation très importante dans une base de connaissances structurée en une hiérarchie de classes.

1.2. Description de TROPES

TROPES (la description du modèle a été repris de [MAR 89]) est un système de représentation par objets, basé sur les notions de classe et d'objet et qui permet le traitement indépendant des différents points de vue d'un concept.

1.2.1. Les concepts

Dans le modèle TROPES, une base des connaissances B est composée d'un ou plusieurs concepts. Ces concepts sont complètement indépendants et déterminent une partition de l'univers du discours (figure 1.2). Ainsi, au lieu de travailler avec un seul graphe de classes, la base contient une structure hiérarchique de classes pour chaque concept.

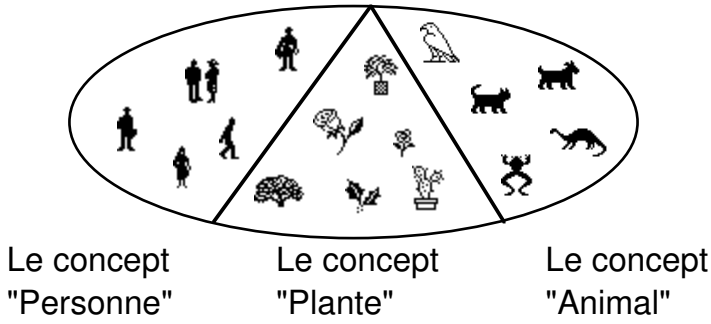


Figure 1.2 : L'univers "exemple" est divisé en trois concepts indépendants : le concept "personne", le concept "plante" et le concept "animal".

La base de connaissances de la figure 1.2 est décrit dans TROPES par :

```
base
{       nom     "exemple";
        concepts [Personne,Plante,Animal]
}
```

Dans l'annexe 4, il y a une spécification complète du langage de TROPES et un exemple de base de connaissances.

1.2.2. Les points de vue, les classes et les passerelles

Dans TROPES, les objets qui appartiennent à un concept peuvent être vus selon différents points de vue. Un point de vue peut être défini en général comme une position conceptuelle de laquelle un observateur regarde un objet. Il existe l'approche spatiale donnée par Minsky [MIN 75], dans laquelle un objet est représenté sous la forme d'un tableau de "frames", où chaque frame représente un point de vue de l'objet à partir d'une certaine position spatiale. Mais il existe aussi l'approche "multi-expertise" utilisée dans la plupart des systèmes experts. L'univers est divisé en différents points de vue, et chaque point de vue définit ses propres attributs. Il est important de noter qu'on part de l'hypothèse d'un monde unique, figé et objectif, ce qui entraîne que l'union de toutes les perspectives d'un objet produit une représentation cohérente de l'objet [MAR 89]. Ainsi, un concept est composé d'un ensemble de points de vue (figure 1.3).

Par exemple, la base de connaissances de la figure 1.3 est décrit dans TROPES par :

```
base
{       nom     "exemple";
        concepts ["Un concept"] ;
        points-de-vue [PV1,PV2,PV3] ;
}
```

Les différents points de vue d'un concept forment des structurations différentes d'un même ensemble d'individus. Mais il peut y avoir des liens logiques entre elles. Ces liens sont les passerelles. Une passerelle entre deux points de vue PVi et PVj, dans TROPES, est une classe, qui a une représentation dans PVi et une représentation dans PVj.

L'exemple trivial d'une passerelle est la racine du concept qui représente, dans tous les points de vue la même extension, l'univers de discours du concept. Tout objet du concept appartient à la classe racine du concept dans chaque point de vue. Du point de vue ensembliste, nous pouvons conclure que même si l'objet est rattaché à une classe C plus bas dans l'arbre de classes, l'objet doit appartenir à la racine, car la classe de rattachement C est une spécialisation de la racine. Autrement dit la classe C est un sous-ensemble de la classe racine.

D'autre part, les passerelles peuvent être vues comme un mécanisme d'inférence très puissant, car s'il existe des passerelles dans les niveaux inférieurs de l'arbre, le fait de savoir qu'un objet appartient à une de ces passerelles, selon un certain point de vue, permet de faire des inférences sur sa position dans la structure d'un autre point de vue (figure 1.3).

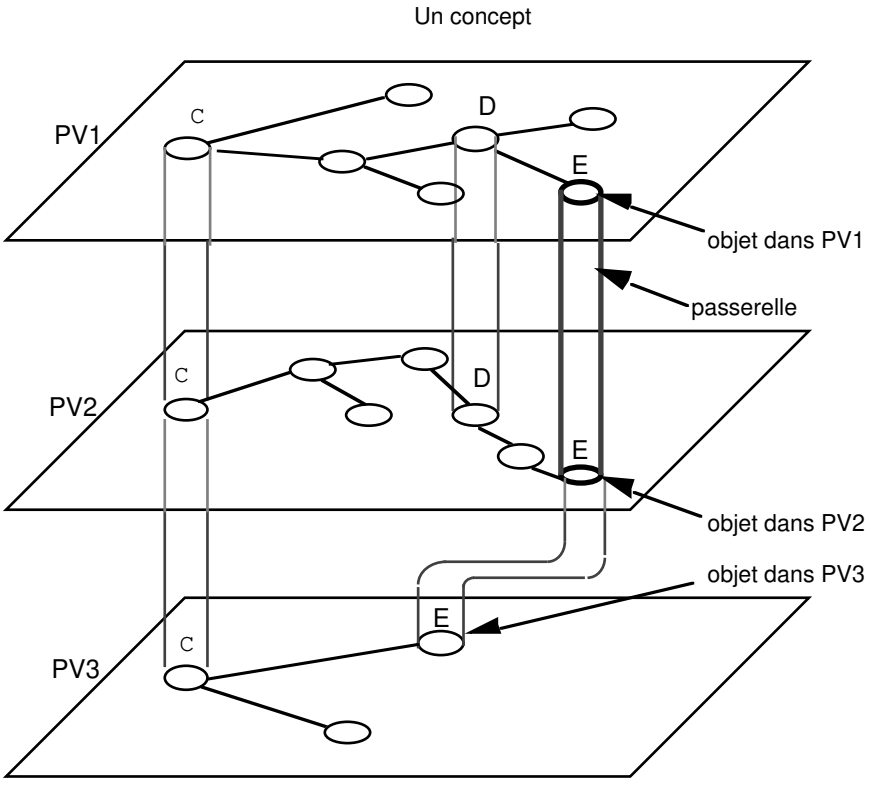


Figure 1.3 : Pour la base de connaissances "exemple" composée du concept "Un concept", si un objet appartient à la classe E du point de vue PV1, nous pouvons inférer que cet objet appartient aussi à la classe E dans les points de vue PV2 et PV3.

D'autre part, étant donné qu'un objet est rattaché à une classe dans tous les points de vue du concept, une opération sur ce concept doit être réalisée sur l'intégralité des points de vue du concept. Par exemple, lors d'une requête, si nous voulons savoir les contraintes des différents attributs d'une instance, il faut regarder dans tous les points de vue.

Il existe deux possibilités dans le modèle TROPES lors du déroulement d'une opération sur un point de vue PVi :

- Il existe au moins une passerelle entre PVi et PVj, donc un changement sur le point de vue PVi peut entraîner un changement sur l'autre point de vue PVj.

- Il n'existe aucune passerelle entre les points de vue, donc le déroulement de l'opération dans chaque point de vue est complètement indépendant du déroulement dans l'autre point de vue.

Donc, le déroulement d'une opération sur un point de vue peut entraîner des changements dans l'état des opérations sur l'autre point de vue. Par exemple, lors de la classification d'un objet, si celui-ci appartient à une classe E dans un point de vue PVi, et la classe est une passerelle entre PVi et PVj, alors l'objet appartient aussi à la même classe dans le point de vue PVj. Dans ce cas, la classification de l'objet dans PVj va être dirigée vers la classe E (figure 1.3). Une classe C d'un point de vue PVi est notée C/PVi.

Par exemple, la passerelle de D/PV1 vers D/PV2 de la base de connaissances de la figure 1.3 est décrit dans TROPES par :

```
pass
{       de      = exemple.D/PV1;
        vers    = exemple.D/PV2;
}
```

1.2.3. Les attributs d'une classe

La définition d'une classe C d'un concept dans TROPES est faite, sous un point de vue PV_i, par sa sur-classe C' dans ce point de vue, et par les attributs affinés ou définis dans la classe et visibles dans ce point de vue [MAR 89] :

$C/PV_i = C'/PV_i + [a_{c1}:t_1 , ... , a_{cj}:t_j , ... , a_{cn}:t_n]$[1].

Un attribut d'une classe est une propriété que possèdent tous les individus de la classe. Tous ces attributs affinés ou définis dans la classe C/PV_i font partie de l'ensemble d'attributs du concept, et sont visibles dans le point de vue PV_i.

La classe C est une spécialisation de la classe C', donc, les attributs de la classe C, sous le point de vue PV_i, peuvent être des attributs hérités de la classe C' ou de ses sur-classes, des attributs affinés ou des attributs définis dans la classe elle-même.

La description complète d'un attribut $\mathbf{a_i}$ pour une classe C/PV_i comporte un ensemble de contraintes décrivant le sous-domaine des valeurs du concept de base, valides pour l'attribut $\mathbf{a_i}$ d'un objet I de la classe C; ces contraintes incluent la description du type de l'attribut et optionnellement des contraintes sur les valeurs permises ou la cardinalité de l'attribut pour le cas de attributs multi-valués.

Il existe plusieurs descripteurs d'attribut : descripteurs de type, de valeurs valides, de cardinalité et de la valeur par défaut.

- Le descripteur de type indique si l'attribut est mono-valué (le descripteur est "un") ou si l'attribut est multi-valué (le descripteur est "ensemble-de").
- Les descripteurs de valeurs valides sont "domaine" ou "sauf". Le descripteur "domaine" indique l'ensemble des valeurs possibles : soit par énumération exhaustive des éléments, soit par des intervalles. Le descripteur "sauf" indique les valeurs incluses dans la description du domaine qui ne sont pas valides.
- Le descripteur de cardinalité "card" indique le nombre de valeurs simples que peut avoir la valeur de l'attribut.
- Le descripteur par défaut indique la valeur la plus commune pour un attribut d'une classe.

Par exemple :

```
classe
{       Personnes/Personnel
        age     un      entier
                domaine [0 90];
        situation-de-famille    un      chaîne
                domaine         "célibataire","marié","veuf","divorcé" ;
        revenu          un      entier
                domaine         [0 15000]
                sauf            1000, 2000;
}
```

---

[1] a_{c1} représente un attribut affiné ou défini dans la classe C et t_1 ses contraintes dans cette classe.

1.2.4. Appartenance d'un objet à une classe

Chaque concept a un ensemble d'attributs clés, qui est l'ensemble minimal d'attributs qui doit être évalué dans une instance pour pouvoir dire qu'elle existe dans le concept. La clé d'un objet est la combinaison des valeurs de ces attributs, qui décrit l'objet de façon unique dans le concept.

Un objet I appartient à une classe C, sous un point de vue PVi, si l'objet I appartient au concept il satisfait toutes les contraintes des attributs de la classe C/PVi. L'ensemble d'attributs pour cette classe est composé des attributs hérités, des attributs affinés et des attributs définis.

Un objet d'un concept est rattaché à la totalité des points de vue du concept à travers ses classes d'appartenance. Ces classes sont les classes les plus spécialisées de chaque point de vue pour lesquelles l'instance satisfait les contraintes. Donc, les valeurs pour les attributs dans l'objet I satisfont toutes les contraintes des attributs dans les différentes classes d'appartenance (figure 1.4).

Les objets sont donc composés d'un ensemble de valeurs pour les différents attributs de leurs classes d'appartenance. Le mot valeur est utilisé de façon plutôt générale. Il peut exister des attributs qui sont multi-valués, pour lesquels l'utilisateur donne un ensemble de valeurs. Dans ce cas, l'ensemble représente la valeur pour l'attribut.

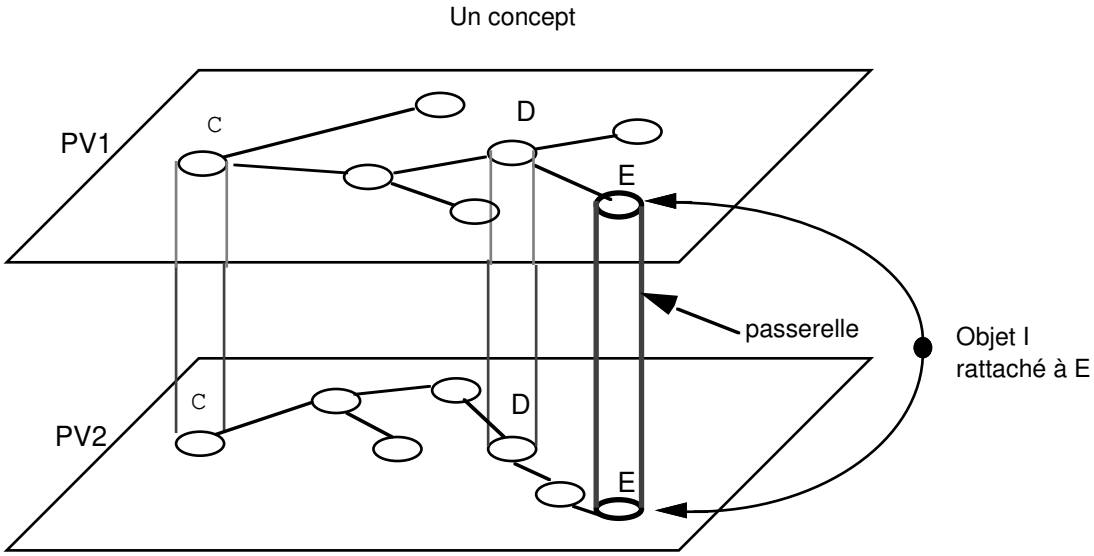


Figure 1.4 : Appartenance d'un objet à divers points de vue.

De plus, lorsque deux classes d'appartenance de I, dans différents points de vue, font référence au même attribut, sa valeur dans I satisfait les contraintes des deux classes.

Finalement, l'ensemble d'objets qui appartiennent à la base de connaissances est l'union des ensembles d'objets rattachés aux différents concepts.

1.3. Classification d'un objet dans TROPES

Dans cette section, il est présenté un algorithme séquentiel de classification des objets dans le modèle de connaissances TROPES.

1.3.1. Classes "sûres" ou "impossibles"

L'état d'une classe vient du fait de la comparaison entre un objet et une classe. Cette comparaison peut donner trois résultats différents[2] : "sûre" si l'instance appartient à la classe, "impossible" si les valeurs des attributs de l'instance sont en contradiction avec les contraintes de ces attributs dans la classe, et "possible" si l'instance n'est pas en contradiction avec la classe, mais s'il manque des informations pour être sûr de son appartenance à la classe.

D'autre part, une classe "sûre" peut être "sûre non terminale" ou "sûre terminale". Une classe est "sûre non terminale" lorsque son état est "sûre" et qu'elle n'a que des sous-classes ayant l'état "sûre" ou "possible". Une classe est dite "sûre terminale" lorsque son état est "sûre" et

[2] Ces états ont été définis d'abord dans SHIRKA [REC 88], et après dans TROPES.

qu'elle est une feuille de l'arbre de classes, ou que l'état de toutes ses sous-classes est "impossible" (figure 1.5).

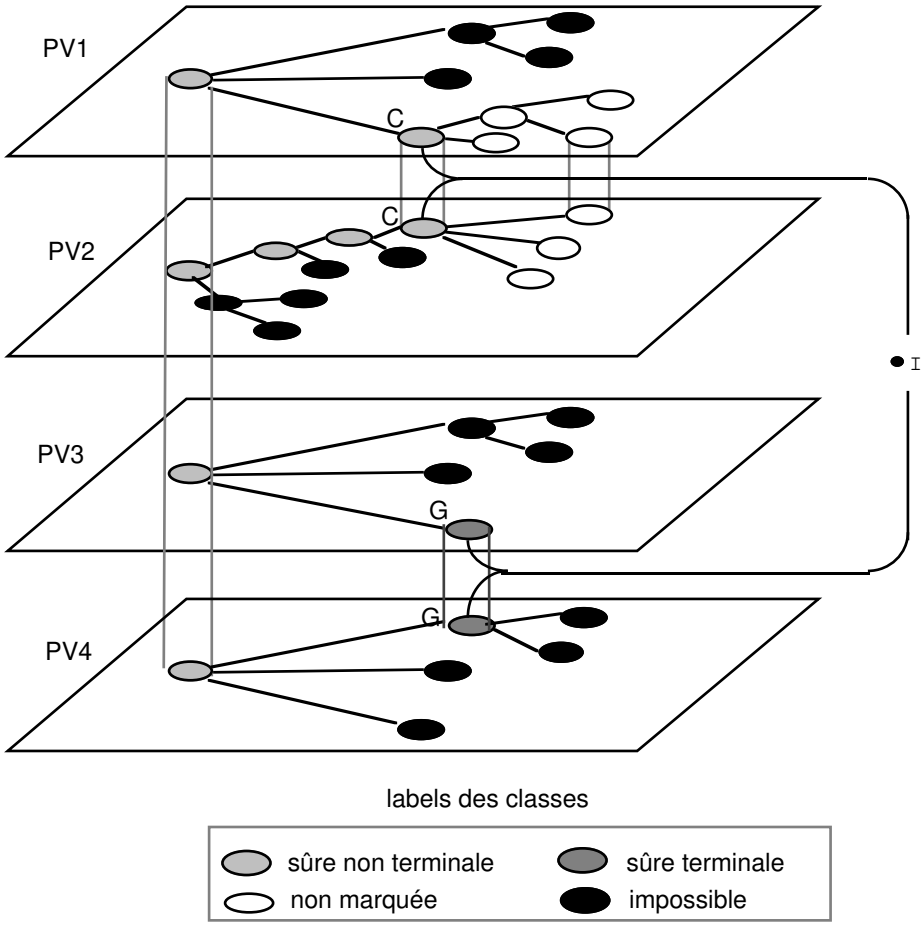


Figure 1.5 : Les états possibles pour les classes d'un point de vue.

L'état des passerelles est le même que pour les classes car une passerelle est une classe. Donc, ces états sont définis comme : "sûre", "possible" et "impossible". Si la passerelle C est "bi-directionnelle" (figure 1.5) entre deux points de vue PVi et PVj, l'état de la passerelle C est le même que celui-ci des classes qu'elle représente C/PVi et C/PVj i≠j. Par contre, si la passerelle P est "uni-directionnelle" ou "orientée" (figure 1.6), définie de la classe C/PVi vers la classe D/PVj, son état dépend des états des deux classes C/PVi et D/PVj (table 1.1).

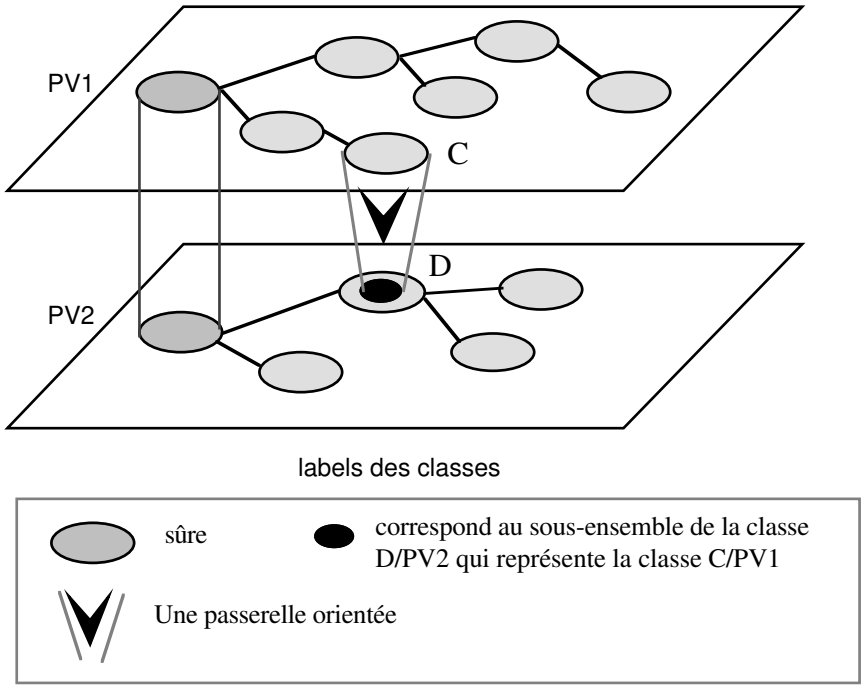


Figure 1.6 : passerelles orientées entre les points de vue.

C/PVi	D/PVj	L'état de la passerelle
sûre	sûre	sûre
sûre	impossible	(*)
sûre	possible	sûre
impossible	sûre	impossible
impossible	impossible	impossible
impossible	possible	impossible
possible	sûre	possible
possible	impossible	impossible (**)
possible	possible	possible

Table 1.1 : La passerelle est orientée. Elle part de la classe C/PVi vers la classe D/PVj.

(*) Ce cas n'est pas possible étant donné que du point de vue ensembliste la classe C/PVi est un sous-ensemble de la classe D/PVj. Donc, l'appartenance de l'instance à C/PVi entraîne l'appartenance de l'instance à D/PVj.

(**) Dans ce cas, l'état de la classe C/PVi doit aussi devenir "impossible", pour la même raison que (*).

Une passerelle est dite "active" si son état est "possible", sinon la passerelle est "inactive".

Nous reviendrons sur ces sujets traités dans (*) et (**) dans la section sur la propagation de marques par les passerelles.

1.3.2. Point de vue actif vs. point de vue inactif

Le but de cette section est de décrire les différentes sortes de point de vue lors de la classification, étant donné la connaissance de l'utilisateur sur les points de vue, qui font partie du concept. Nous montrons également quels sont les états possibles pour les points de vue pendant la classification d'un objet.

La puissance de TROPES, modèle multi-points de vue, pour la classification d'un objet vient du fait qu'un utilisateur, qui est expert dans un domaine particulier peut lancer la classification sur un sous-ensemble des points de vue du concept qu'il connaît le mieux. Ce sous-ensemble forme donc l'ensemble des points de vue "principaux". A partir de cet ensemble et grâce aux passerelles l'utilisateur peut inférer des informations d'autres points de vue "auxiliaires" ou "cachés" sur lesquels l'utilisateur a moins d'information que sur les "principaux".

Il y a donc trois sortes de points de vue, selon la connaissance que l'utilisateur en a (figure 1.7): "principaux", "auxiliaires" et "cachés".

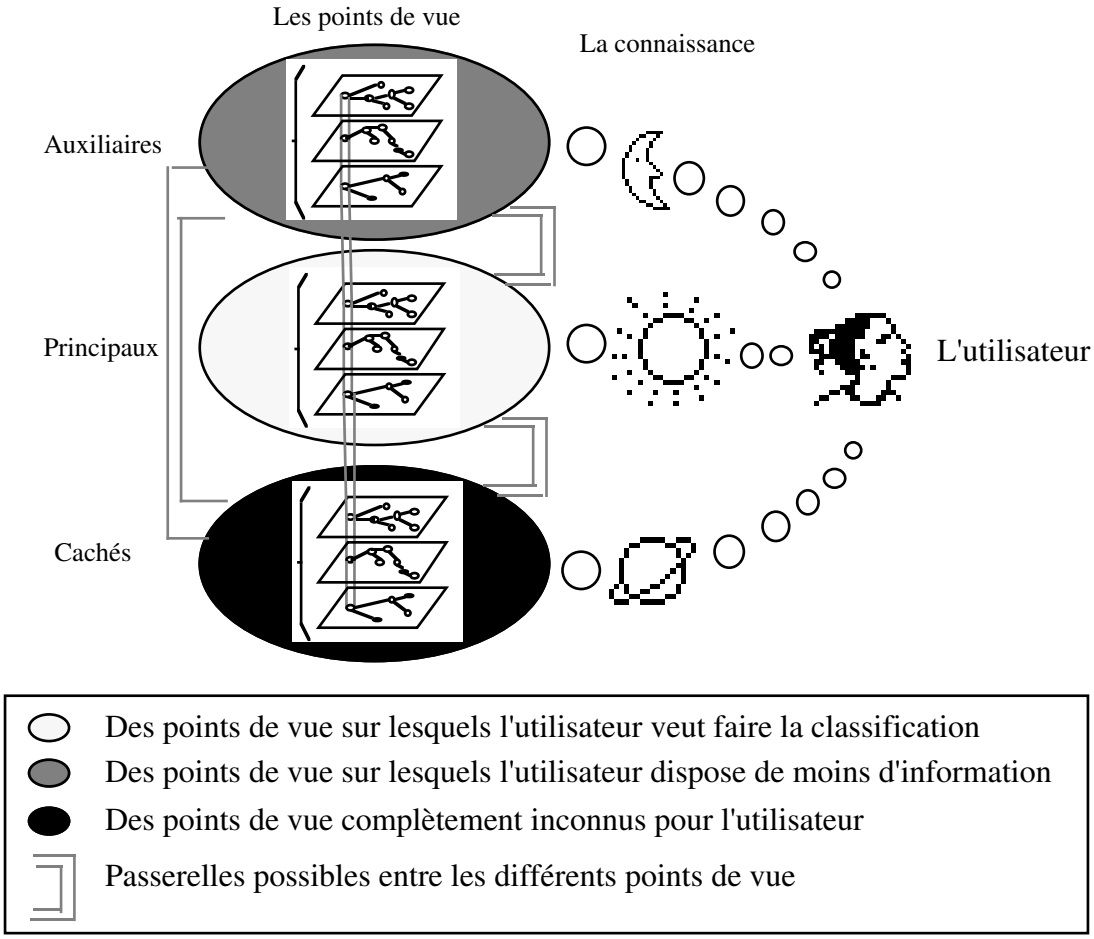


Figure 1.7 : Les connaissances de l'utilisateur sur les points de vue.

- Les points de vue "principaux" sont ceux sur lesquels l'utilisateur veut faire la classification, car ils sont les plus connus pour l'utilisateur. Le système demande toutes les valeurs des attributs définis dans ces points de vue au fur et à mesure qu'il en a besoin pour la classification de l'instance. Le but du système est de faire descendre l'instance le plus bas possible dans des arbres de classes. Ces points de vue sont notés :

  points de vue "principaux" = $[PV_{p1}, \ldots, PV_{pn}]$

- il y a aussi des points de vue sur lesquels l'utilisateur dispose de moins d'information : les points de vue "auxiliaires". Pour faire descendre l'instance dans ces points de vue, le système utilise l'information donnée par l'utilisateur pour les points de vue "principaux" en utilisant les passerelles. Ces points de vue sont donc utilisés par le système lorsque la

classification dans les points de vue “principaux” devient impossible. La classification dans un point de vue devient impossible quand l’utilisateur ne peut plus donner de valeurs pour les attributs définis dans ce point de vue. Néanmoins, il peut exister dans ce point de vue des sous-classes de la classe actuelle (la classe actuelle est une classe “sûre” dans un point de vue PVi pour laquelle le système est en train d’obtenir les valeurs pour tous ses attributs) qui sont possibles. Le système peut alors utiliser les points de vue “auxiliaires” afin de continuer la classification de l’instance et retourner par une passerelle vers les points de vue “principaux”. Donc, un point de vue “auxiliaire” est utile dans le cas où il existe des passerelles pour lesquelles le système peut revenir vers les points de vue “principaux”. Les points de vue “auxiliaires” seront notés :

points de vue “auxiliaires” = [PV_{a1} , … , PV_{ak}]

La classification d’un objet va être faite dans tous les points de vue “principaux” et “auxiliaires”. Pour les premiers, le système va essayer de descendre l’instance dans chacun d’eux tant qu’il existe des classes “possibles”. Pour les deuxièmes, le système ne fera la classification que s’il existe des passerelles “possibles” de ces points de vue vers les points de vue “principaux”. Etant donné que le système peut utiliser les points de vue “auxiliaires” pour faire descendre l’instance dans les points de vue “principaux”, il peut poser des questions à l’utilisateur sur les attributs définis dans ces points de vue.

- enfin, les points de vue “cachés” sont les perspectives complètement inconnues par l’utilisateur. Le système ne pose jamais de questions sur les attributs définis dans ces points de vue. Néanmoins, s’il existe des passerelles entre les points de vue “principaux” ou “auxiliaires” et les points de vue “cachés”, le système peut utiliser ces points de vue comme “ponts” entre les différents points de vue. Par exemple, si pendant la propagation de marques dans un point de vue “caché” le système arrive à une passerelle entre ce point de vue et un autre “principal”, le système peut aussi déclencher une propagation de marques dans ce dernier. Les points de vue “cachés” seront notés :

  points de vue “cachés” = [PV_{c1} , … , PV_{cm}]

En résumé, les points de vue “principaux” sont ceux sur lesquels l’utilisateur veut faire la classification. Les points de vue “auxiliaires” permettent au système d’inférer des connaissances d’un point de vue “principal” dans lequel la classification s’est arrêtée. Ces inférences sont faites grâce aux passerelles. Enfin, les points de vue “cachés” sont les “ponts” entre les différents points de vue pour la propagation de marques (figure 1.8).
Le type d’un point de vue “principal”, “auxiliaire” ou “caché” reste toujours le même pendant la classification d’un objet. L’état de ces points de vue peut changer pendant la classification de l’instance. Par exemple si le système arrive à une feuille d’un arbre de classes, il est évident que le système ne peut plus descendre l’instance dans ce point de vue.

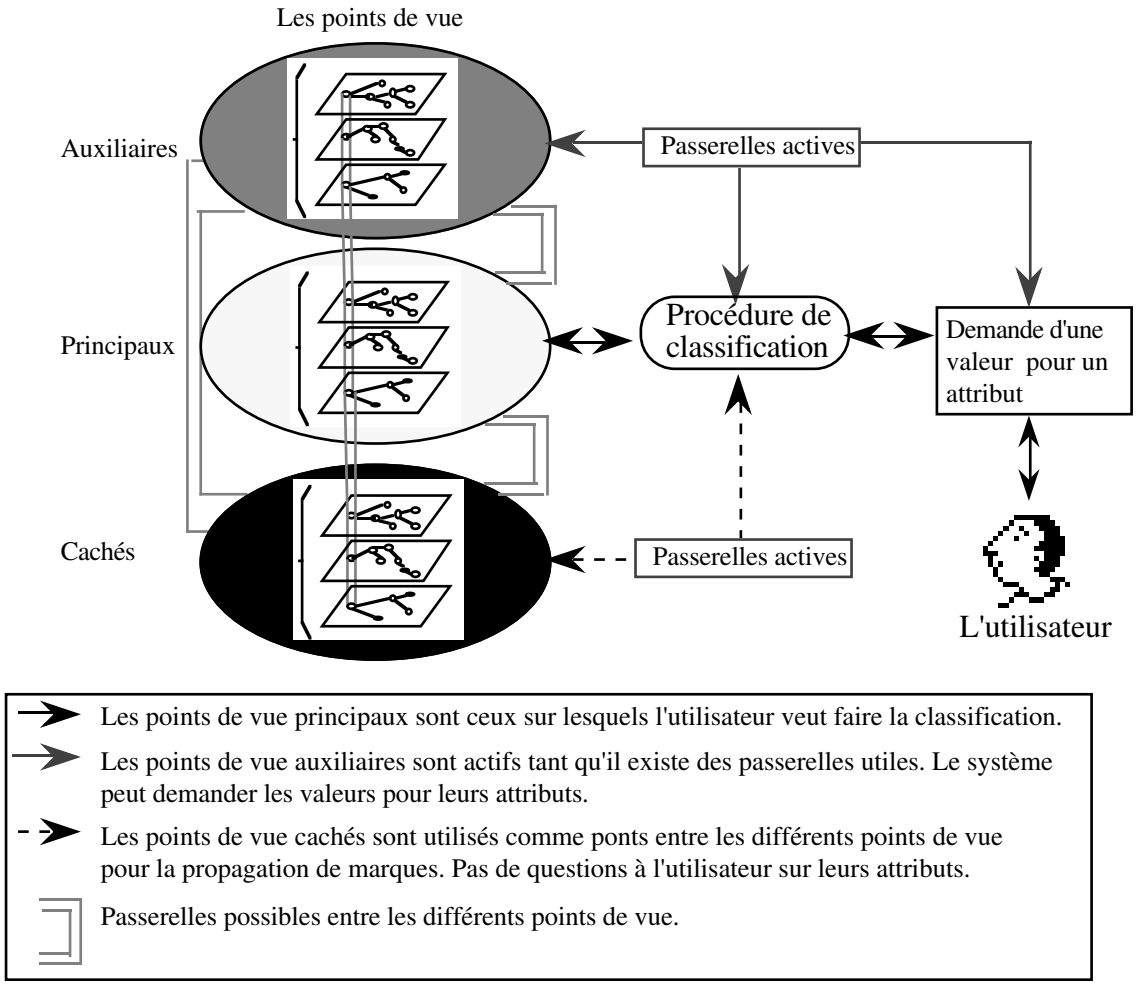


Figure 1.8 : La classification d'un objet dans les différents points de vue.

Une fois que nous avons défini l'état d'une classe et d'une passerelle et les sortes des points de vue selon la connaissances de l'utilisateur, il ne nous reste qu'à définir l'état des points de vue. Il existe deux états pour les points de vue : "actif" et "inactif".
Un point de vue "actif" est un point de vue qui est encore utile pour la classification de l'instance. Les points de vue "principaux" sont "actifs" s'ils ont au moins un sous-arbre non vide "actif" dans le sens que l'état des classes de ce sous-arbre soit toujours "possible"; il est actif car le système peut essayer de faire descendre plus bas l'instance vers ces sous-arbres. Un point de vue "auxiliaire" ou "caché" est "actif" s'il a au moins une passerelle "active" c'est-à-dire dont son état est "possible".
Etant donné que pendant la classification d'un objet plusieurs points de vue peuvent être "actifs", mais que quelques uns d'entre eux peuvent être "en attente" parce que la classification s'avère impossible, nous distinguons deux cas :

- un point de vue "actif courant" sera un point de vue "actif" sur lequel l'algorithme de classification est en train de classer l'instance.
- un point de vue "actif en attente" est un point de vue "actif", mais pour lequel l'utilisateur ne peut plus donner de valeurs pour les attributs, donc il faut attendre pour essayer de revenir à travers une passerelle et continuer la classification en faisant une inférence. Dans ce cas une inférence sert à déduire des nouvelles connaissances en s'appuyant sur les connaissances déjà existantes dans la base.

Les points de vue "actifs courants" par la suite seront simplement appelés "actifs" (figure 1.9).

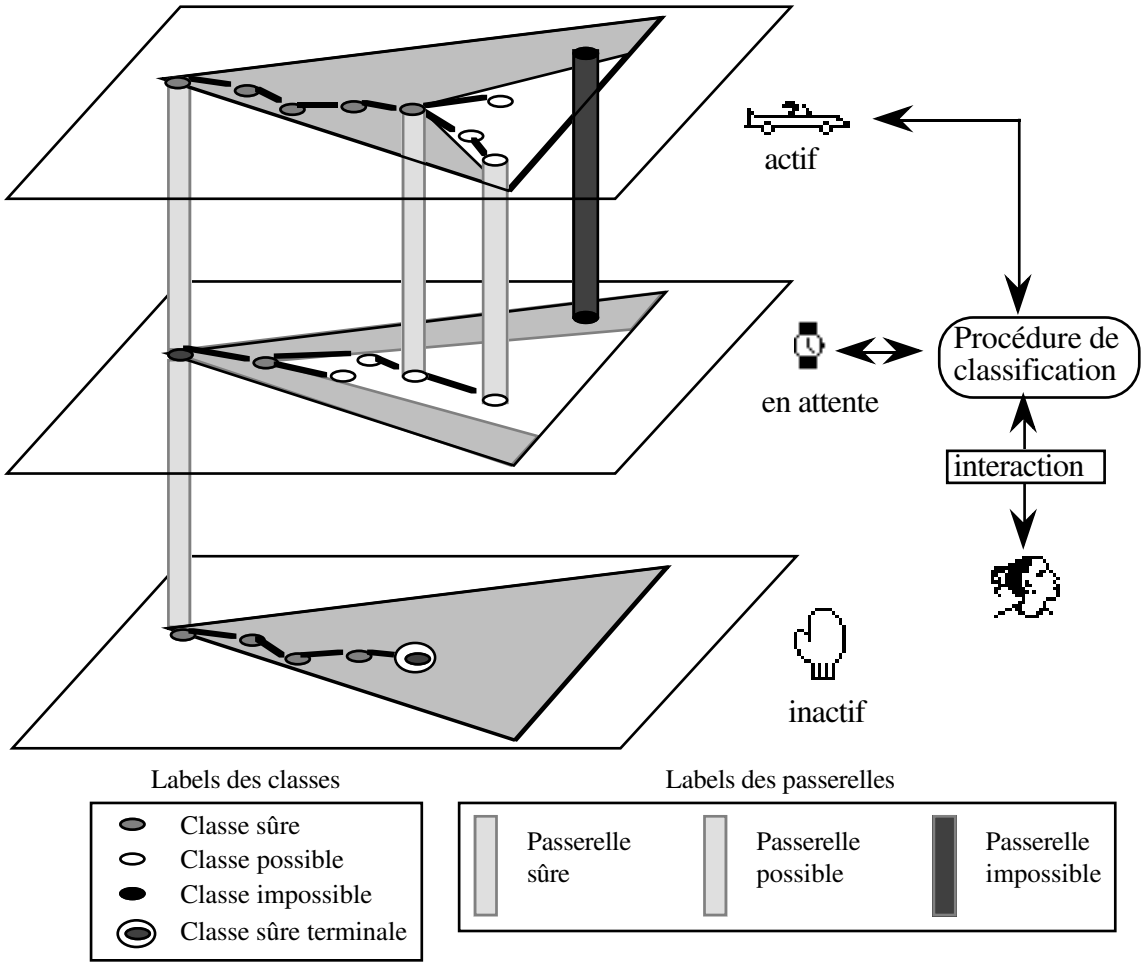


Figure 1.9 : états des points de vue “principaux” :

- état “actif” : il existe toujours des classes marquées “possibles” pour le point de vue.
- état “en attente” : il y a des classes “possibles” dans le point de vue, mais la descente s’avère impossible.
- état “inactif” : il y a une classe “sûre terminale”.

Enfin, un point de vue principal est “inactif” lorsqu’il a une classe “sûre terminale”. Les points de vue “auxiliaires” ou “cachés” sont “inactifs” lorsqu’ils n’ont plus de passerelles “actives”.

1.3.3. Algorithme séquentiel de classification dans TROPES

Le but de l'algorithme de classification d'un objet dans un concept est de faire descendre cet objet le plus bas possible dans les différents points de vue. Cette descente entraîne une interaction permanente avec l'utilisateur afin de lui demander les valeurs pour les différents attributs. Au fur et à mesure que l'algorithme de classification fait descendre l'objet I dans les différents points de vue, la connaissance sur I s'enrichit. Cette descente entraîne aussi qu'à chaque pas, l'ensemble des contraintes des attributs augmente : il est possible d'ajouter des nouveaux attributs (définition d'attributs), ou d'ajouter de nouvelles contraintes pour des attributs déjà existants (attributs affinés). Il faut remarquer qu'une instance ne peut être rattachée à une classe C/PVi que si toutes les valeurs pour les attributs de C/PVi sont connues et satisfont les contraintes correspondantes.
L'algorithme actuel de classification des instances dans TROPES est composé de cinq pas (algorithme 1.1) [MAR 89] : obtention d'information, appariement, propagation de marques, actualisation d'information et choix du prochain point de vue.

```
tant que (non condition pour finir la classification)
        obtention d'information
        appariement
        propagation de marques
        actualisation d'information
        choix du prochain point de vue
fin tant que
```

Algorithme 1.1 : Cet algorithme est composé de cinq pas, qui vont être décrits dans la section suivante : obtention d'information, appariement, propagation de marques, actualisation d'information et choix du prochain point de vue.

1.3.2.1. Obtention d'information

A travers la procédure d'obtention d'information (figure 1.10), le système obtient de l'utilisateur l'information manquante sur la classe actuelle du point de vue à regarder PVi. L'utilisateur n'est autorisé à donner que des valeurs valides pour les attributs dans cette classe.

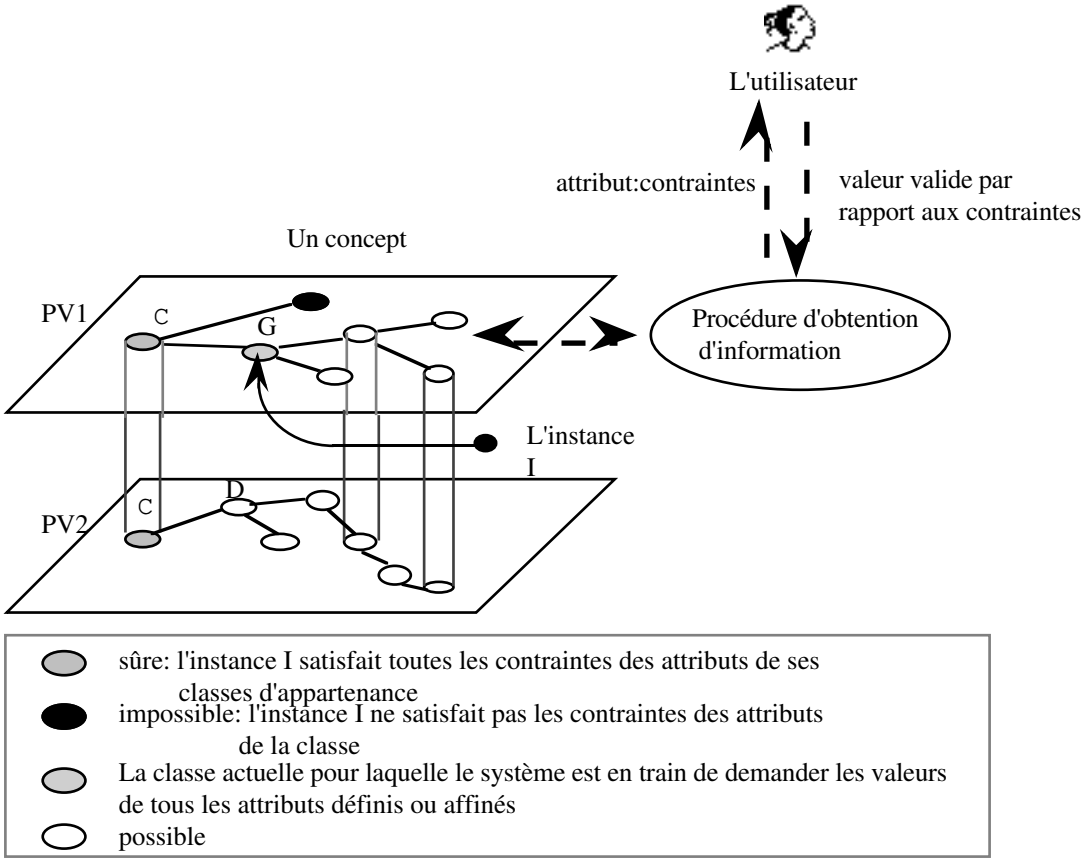


Figure 1.10 : Le système est en train de demander toute l'information manquante sur la classe actuelle.

1.3.2.2. Appariement

Une fois l'information de la classe actuelle C/PVi complétée, le système cherche à descendre l'instance dans son point de vue PVi. Pour cela, la procédure d'appariement fait une comparaison entre les valeurs déjà données par l'utilisateur pour les différents attributs, et les contraintes des attributs affinés dans les sous-classes (figure 1.11). Pour les attributs définis dans les sous-classes, le système demande à l'utilisateur les valeurs des attributs dont il a besoin. Comme résultat de cette comparaison, les sous-classes deviennent "impossibles" ou "sûres" ou elles restent "possibles". Il n'y a qu'une sous-classe qui peut devenir "sûre".

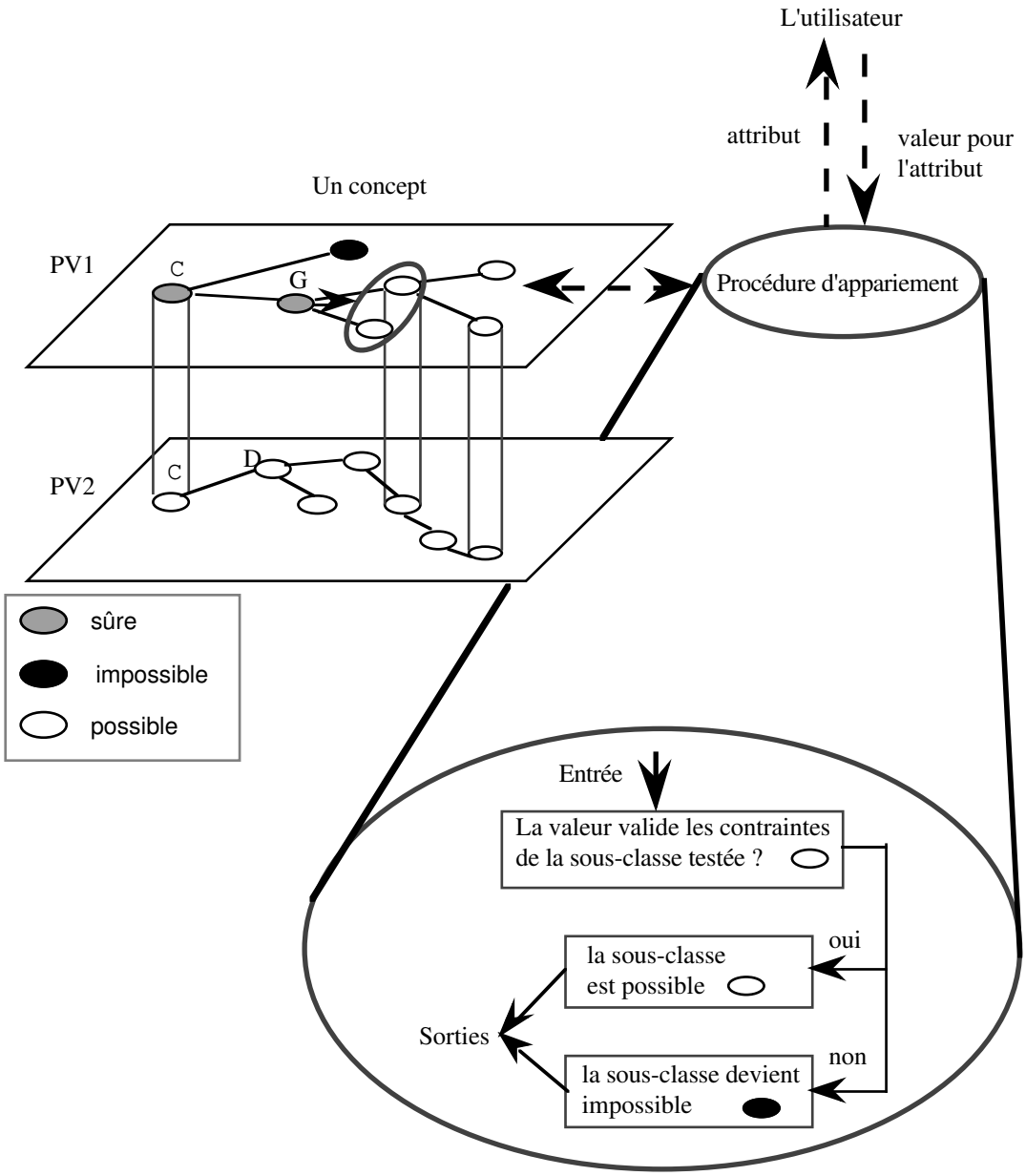


Figure 1.11 : Une fois l'information de la classe actuelle complétée, le système cherche à faire descendre l'instance dans le point de vue actuel. La procédure d'appariement fait une comparaison entre les valeurs des attributs et les contraintes des différentes sous-classes de la classe actuelle. Comme résultat de la comparaison, la procédure marque chacune des classes révisées comme "impossible" ou "sûre".

1.3.3.3. Propagation de marques

Un fois la procédure d'appariement finie, la procédure de propagation de marques met à jour le point de vue PVi en marquant les sous-classes des classes qui sont devenues "impossibles"

comme elles aussi “impossibles”. Dans le cas où au moins une des sous-classes marquées est une passerelle, il faut aussi mettre à jour les autres points de vue qui participent à cette passerelle : il faut propager les marques “sûres” et “impossibles”. Si la marque propagée pour la passerelle C est “impossible”, toutes ses sous-classes vont devenir aussi “impossibles”. Et si la marque propagée est “sûre”, toutes les sur-classes de C vont devenir aussi “sûres” (figure 1.12). Enfin, cette étape met à jour l’état actif ou inactif des points de vue ainsi que leur nombre de passerelles actives.

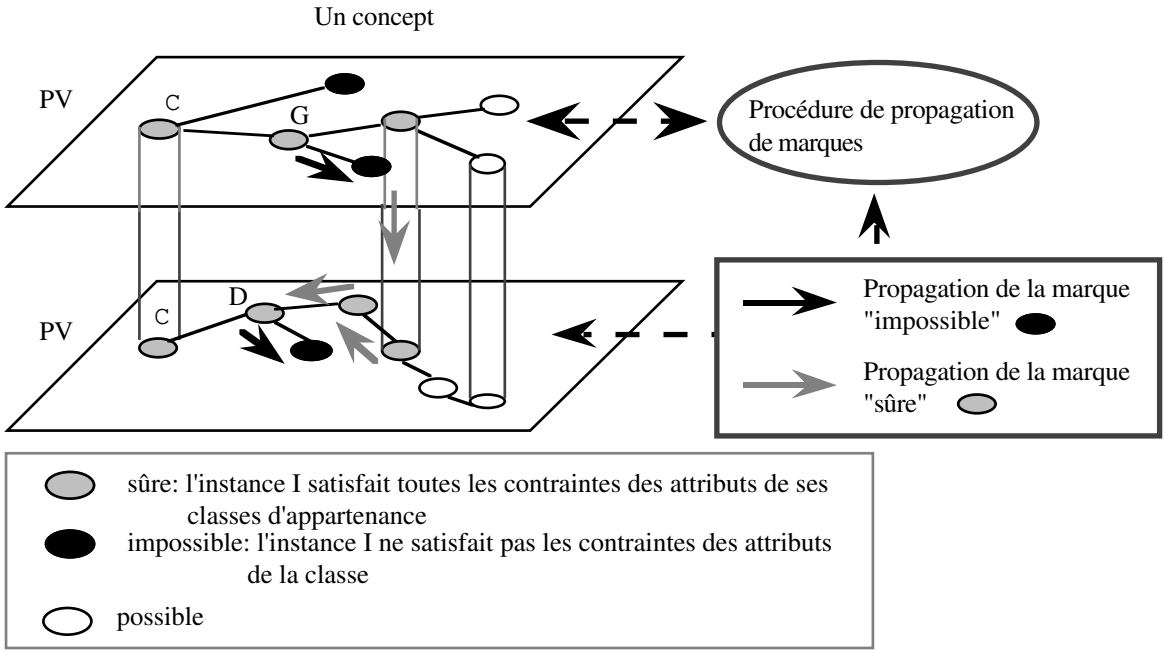


Figure 1.12 : Propagation de marques dans le point de vue actuel, et par les passerelles vers les autres points de vue.

1.3.3.4. Mise à jour et choix du point de vue

La procédure de mise à jour d’information met à jour l’information de la classe “sûre” de chaque point de vue et récupère toute l’information agrégée. Finalement, la procédure “choix du point de vue” calcule le point de vue à prendre comme actuel pour continuer la classification de l’instance (figure 1.13).

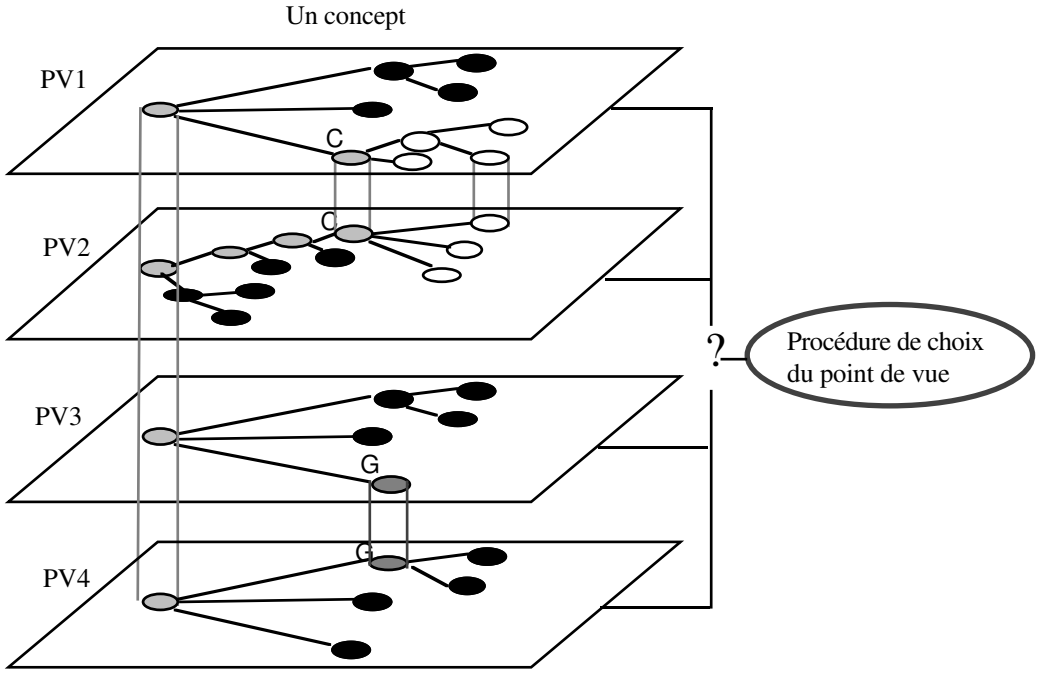


Figure 1.13 : Choix du prochain point de vue.

1.4. Correction de l'algorithme de classification

Cette section présente la formalisation du modèle TROPES et de la classification des instances dans celui-ci. Le formalisme utilisé pour décrire l'algorithme dans cette partie ainsi que les différents théorèmes démontrées sont le résultat d'un travail d'équipe développé avec Rodrigo Cardoso et Olga Mariño de l'Université des Andes (Bogotá)

Définition 1

Le triplet FC = (X,C,f) est une forme de classification pour X ssi :

- X est un ensemble d'individus, appelé l'univers de FC.
- C est une partition de X', $X' \subseteq X$, $C = \{X_1,\ldots,X_h\}$ avec $h \geq 1$, et telle que $\forall i\ [1..h]$, $Xi \subseteq X'$, et $Xi \cap Xj = \varnothing$ pour tout $i \neq j$.
- $f: X \rightarrow \{0,1,\ldots,h\}$ et telle que $\forall i\ [0..h]$, $f(x)=i \Leftrightarrow x \in Xi$. f est appelée la fonction de classification et $X_0 = X$.

Dans la définition précédente, X représente une classe, C représente l'ensemble de sous-classes de X et f représente l'algorithme de classification pour trouver une sous-classe de la classe X (la classe actuelle). Dans le cas d'une taxonomie X' = X.
Si $f(x)=0$ alors l'objet x ne peut pas classifié dans une sous-classe de la classe actuelle et sa classe d'appartenance est toujours X.
Exemple :

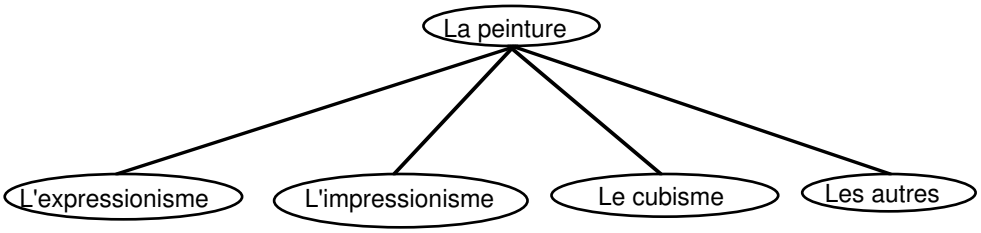
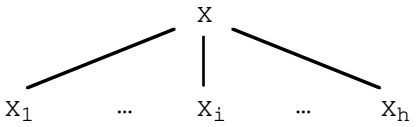


Figure 1.14 : Une classe composée en quatre sous-classes.
Pour la figure 1.14 : X = La classe peinture, C = {l'expressionisme, l'impressionisme, le cubisme, les autres} et f = la fonction qui donne une sous-classe de la classe Peinture. Dans ce cas, C est une partition d'un sous-ensemble de la classe peinture.
Remarques
- Le cas particulier où h=0 est une forme de classification triviale. Dans ce cas nous avons C = { } et $\forall x \in X$, f(x)=0.
- Une FC peut être représentée graphiquement par un arbre de profondeur 1 (ou 0 cas d'une FC triviale) de la façon suivante :



- Un élément x de X est **classifié** pour la forme de classification FC = (X,C,f) lorsque l'appartenance de x à l'un des ensembles de la partition C de X est déterminée en accord avec la valeur de f pour x. Si $h \geq 1$ et $f(x)=0$ alors x ne peut pas être classifié dans l'un des sous-ensembles de X donnés par C, il est alors classifié dans X. Si h=0, x est trivialement classifié dans X.

Définition 2

Un arbre de classification A(X), noté AC, est un arbre fini dont les nœuds sont des FC et tel que :
i. La racine est une FC (X,C,f).
ii. Si un nœud est une FC (Y,D,g) et ses successeurs sont les FCs

$$(Y_1,D_1,g_1),\ldots,(Y_h,D_h,g_h)$$

alors la partition D est donnée par les Y_i : $D = \{Y_1,\ldots,Y_h\}$.

Dans la définition précédente, A(X) représente un point de vue (la classe racine est X) et D est l'ensemble des sous-classes d'une classe.

Exemple

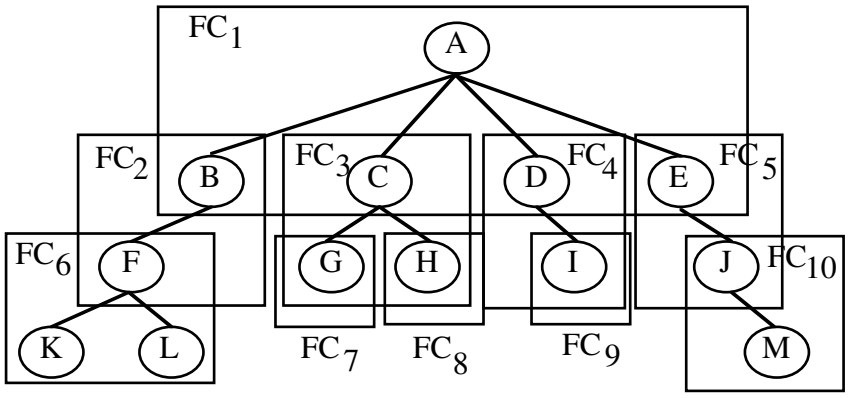


Figure 1.15 : Un arbre (point de vue) AC, dont la racine est A.

Dans la figure 1.15, le point de vue AC est formé par l'ensemble des FC (AC = {FC_1, ..., FC_{10}}). Egalement, FC_1= (A,C,f), où C est la partition {B,C,D,E}.

Un AC est donné par l'ensemble de ses nœuds, des FCs, par exemple

AC = $\{(X_1,D_1,f_1),\ldots,(X_h,D_h,f_h)\} = \{(A,\{B,C,D,E\},f_1),\ldots,(J,\{M\},f_{10})\}$. Un AC peut être vu comme un arbre étiqueté[3] où :

- La racine est (X,C,f).

---

[3] Les définitions suivantes ont été prises dans [COM 88].

Définition de position : N désignera l'ensemble des entiers naturels. N+ est l'ensemble des entiers naturels non nuls. N_+^* est l'ensemble des suites finies de naturels non nuls. $\lambda \in N_+^*$ est la suite vide.

Si $p,q \in N_+^*$, $p.q \in N_+^*$ est la concaténation des séquences p et q.

Si $i \in$ N+, nous identifions i avec la séquence à un élément (i).

N_+^* est muni de la relation d'ordre $\leq_p$ définie par :

$$\forall\, p,q \in N_+^*,\ p \leq_p q \Leftrightarrow \exists \rho \in N_+^*,\ p.\rho = q.$$

Dans ce cas, nous disons que p est un préfixe de q. Nous pouvons remarquer que tout ensemble de positions contient ε, aussi appelé racine.

Un ensemble de positions Pos est un sous-ensemble non vide de N_+^* tel que :

- $\forall p \in$ Pos, $\forall q \in N_+^*$, $q \leq_p p \Rightarrow q \in$ Pos.

- $\forall p \in$ Pos, $\forall i \in N_+^*$, $p.i \in$ Pos $\Rightarrow \forall\ 1 \leq j \leq i$, $p.j \in$ Pos.

Une position $p \in$ Pos est appelée feuille si elle n'est préfixe d'aucune position de Pos autre qu'elle même.

Définition d'arbre étiqueté : Etant donnée une signature (S,F) et un ensemble de variables X, un arbre étiqueté (par (S,F,X)) est une application t d'un ensemble de positions Pos(t) (positions de l'arbre) dans $F \cup X$ qui respecte la cardinalité des symboles de fonction :

- Si $t(p) \in F \cup Xo$, alors p est une feuille de Pos(t).
- Si $t(p) = f : s_1 X \ldots X s_n \longrightarrow s$, alors
  - $\forall i \in$ N, $p.i \in$ Pos(t) $\Leftrightarrow$ $1 \leq i \leq n$.
  - $\forall p.i \in$ Pos(t), t(p.i) est une variable de sorte s_i ou bien un élément $g \in F$ de codomaine s_i.

Un arbre (sous-entendu étiqueté par (S,F)) est fini si son ensemble de positions est fini.

- Pour une FC $(X_\alpha, C_\alpha, f_\alpha)$, sa i-ième fille est notée comme $(X_{\alpha.i}, C_{\alpha.i}, f_{\alpha.i})$.

Donc l'ensemble de positions de A(X), noté Pos(A), est donné par l'ensemble des suites finies d'entiers non nuls des nœuds de l'arbre A. Pour deux positions $\alpha, \beta \in Pos(A)$

$$\alpha <_p \beta :\Leftrightarrow \alpha \text{ est un préfixe de } \beta.$$

et U_A est l'univers de A, $U_A = \{X_\alpha \mid \alpha \in Pos(A)\}$.

Par exemple pour l'arbre suivant,

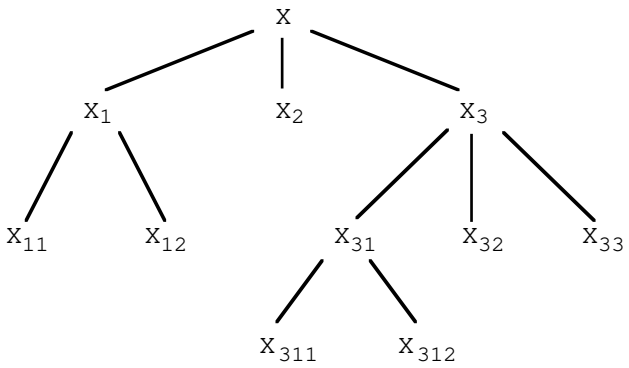


Figure 1.16 : Un point de vue représenté comme un arbre étiqueté. L'ensemble de positions est : $Pos(A) = \{\lambda, 1, 2, 3, 1.1, 1.2, 1.3, 3.1, 3.2, 3.3, 3.1.1, 3.1.2\}$.

Le but de la classification est de faire descendre un objet le plus bas possible dans l'arbre de classes. La classification consiste donc à passer d'un état initial dans lequel l'instance est attachée à la racine de l'arbre, à un état final où elle est attachée à sa classe d'appartenance la plus petite. Pour cela, nous ne pouvons utiliser que l'information contenue dans les fonctions de classification. L'idée est d'utiliser un AC A pour "classifier" les éléments qui appartiennent à la racine. C'est à dire, que pour $x \in X$, l'idée est trouver un $\alpha \in Pos(A)$ tel que $x \in X_\alpha$, et que X_α soit minimal relativement à l'ordre des positions. Dans la suite, nous définissons ces concepts :

Définition 3

Soit $A(X)$ un AC, et $x \in X$.

a La classification de x par rapport à $A(X)$, notée $C_A(x)$, est l'univers $X_{n_1 \ldots n_r}$ tel que:

- Pour $1 \le i < r$: $f_{n_1 \ldots n_i}(x) = n_{i+1}$
- $f_{n_r}(x) = 0$

On dit que x est totalement classifié par rapport à A(X), si la classe d'appartenance pour x est l'univers le plus petit X_α (la classe d'appartenance la plus basse), tel que $x \in X_\alpha$.

Un cas particulier est : si la FC (X, C, f) est triviale, C = { }, nous disons que x est totalement classifié par rapport à $A(X)$.

b La trace de la classification de x par rapport à $A(X)$, notée $A_t(x)$, est l'union du chemin de $A(X)$ qui part de X, et arrive à $C_A(x)$, avec le sous-arbre dont la racine est $C_A(x)$ (figure 1.17).

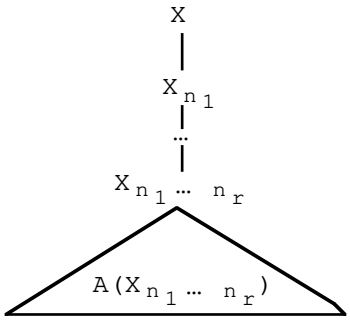


Figure 1.17 : La trace $A_t(x)$ de la classification de x par rapport à A(X).

Par exemple pour la figure 1.16, si un objet x est classifié ($C_A(x)$) dans la classe $X_{3.1.2}$ alors $f_0(X)=3$, $f_3(X_3)=3.1$ et $f_{3.1}(X_{3.1})=3.1.3$. La trace est $A_t(x) = \{X, X_3, X_{3.1}, X_{3.1.2}\}$.

Définition 4

Un concept (un ensemble de points de vue) S(X), noté EPV, est une paire <B,P> où :

- B = {A^1(X),…,A^m(X)} est un ensemble fini de ACs de l'ensemble X.
- La relation de passerelles est une relation binaire définie sur l'ensemble

$$U_B = U_A^1 \cup \ldots \cup U_A^m$$

qui ne met pas en rapport deux éléments du même ensemble U_A^i, i=1,…,m.

L'ensemble de passerelles est noté comme P.

Chaque couple $\langle x_\alpha^i, x_\beta^j \rangle \in P$ est une *passerelle*, et i≠j.

- Un tuple de classification est un ensemble de classes {C_1, ..., C_m}, tel que C_i est la classe d'appartenance de l'objet x pour le point de vue PV_i.

Par exemple, le concept "Les femmes" peut être représenté par 3 points de vue : sports, loisirs et travail.

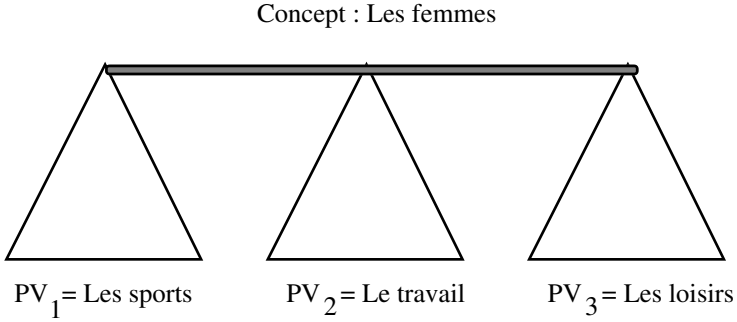


Figure 1.18 : Le concept "Les femmes" où l'ensemble B est composé de trois points de vue : les sports, le travail et les loisirs.

La classification d'un élément par rapport à un EPV consiste à trouver un tuple de classification pour x par rapport aux arbres qui appartiennent au concept (figure 1.18).

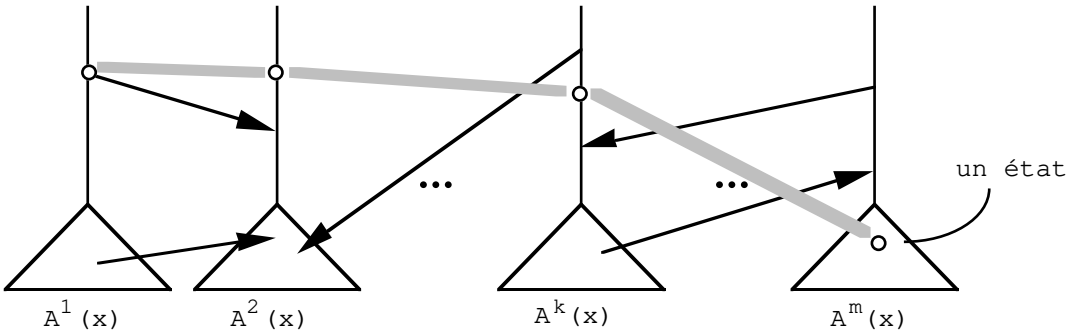


Figure 1.18 : Un tuple de classification dans un concept pour un objet (→ représente les passerelles).

Néanmoins, l'utilisation des passerelles peut aider au processus de classification, car le fait de savoir qu'un objet appartient à une classe, selon un certain AC (point de vue), permet de déduire sa position dans la structure d'un autre AC. C'est-à-dire, si pour le i-ième AC $x \in x_\alpha^i$ et s'il existe une passerelle $\langle x_\alpha^i, x_\beta^j \rangle$, alors on peut conclure $x \in x_\beta^i$.

Définition 5

Soit S = <B,P> un EPV, avec B = {A^1,…,A^m}, x∈X.

a L'ensemble des états de la classification pour x, noté EC, est le produit cartésien :

$$U_{A^1} \times \ldots \times U_{A^m}$$

b L'ensemble E des états "atteignables" à partir d'un état initial ($\rightarrow_R^*$), noté ECA, est un sous-ensemble de EC défini par induction comme :

- <X,…,X> ∈ ECA

- Soient e,e' ∈ ECA, alors e $\rightarrow_R^*$ e' ssi :

  - ou bien, il existe β_i tel que $\beta_i = \alpha_i f_{\alpha_i}(x)$
  - ou bien, il existe une passerelle $<X^j_{\alpha_j}, X^i_{\beta_i}> \in P$, telle que $j \in [1..m]$, $X^j_{\alpha_j} \in e$, $X^i_{\beta_i} \in e'$,

    et β_i n'est pas un préfixe de α_i.

tel que $X^j_{\alpha_j} \in e$ et $X^i_{\beta_i} \in e'$.

c Une S-classification pour x est un état atteignable à partir duquel il n'existe aucun autre état atteignable.

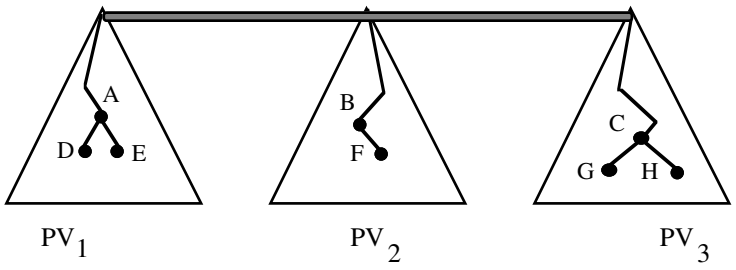


Figure 1.19: Un concept conformé pour trois points de vue.

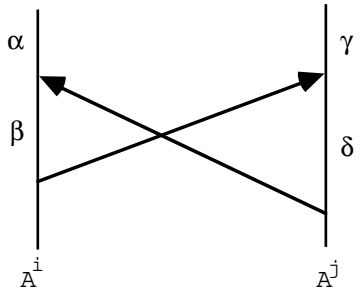
Un état est un ensemble de classes {C_1, ..., C_m} tel que $C_i \in PV_i$. Par exemple, un état possible pour la figure 1.19 (où sont définies les FC) peut être {A,B,C}. Un état atteignable peut être {A,F,C}, ou {D,B,C}, etc. Une S-classification peut être {E,F,G}.

Définition 6

Soit S = <B,P> un EPV.

a S est redondant ssi :

- ou bien, il existe deux passerelles $<X^{i}_{\alpha\beta}, X^{j}_{\gamma}>$, $<X^{j}_{\gamma\delta}, X^{i}_{\alpha}> \in P$, telles que :



- ou bien, il existe une chaîne de passerelles $<X^{i_1}_{\alpha\beta}, X^{i_2}_{\gamma}>$, ..., $<X^{i_k}_{\gamma\delta}, X^{i_1}_{\gamma^{\alpha}}>$,..., $<X^{i_j}_{\gamma^{\varphi}}, X^{i_1}_{\alpha}> \in P$,

tel que : α est un préfixe de $\alpha\beta$.

b S est inconsistant ssi il existe un $x \in X$ pour lequel il y a deux états atteignables e et e', tels que $e = <X^{1}_{\alpha_1},\ldots,X^{k}_{\alpha_\kappa},\ldots, X^{m}_{\alpha_m}>$, $e' = <X^{1}_{\alpha_1},\ldots,X^{k}_{\beta_\kappa},\ldots, X^{m}_{\alpha_m}>$

et $\exists k$ pour lequel α_k n'est pas un préfixe de β_k et β_k n'est pas non plus un préfixe de α_k.

Un EPV consistant est un EPV qui n'est pas inconsistant

c S est (P-) complet (par rapport aux passerelles), si :

$$<X^{i}_{\alpha}, X^{j}_{\beta}> \in P \ \wedge \ \alpha\gamma \in Pos(A^i) \ \Rightarrow \ <X^{i}_{\alpha\gamma}, X^{j}_{\beta}> \in P$$

S est (P-) incomplet si S n'est pas complet. Par exemple dans la figure 1.20, nous avons à droite la complétion du EPV de gauche.

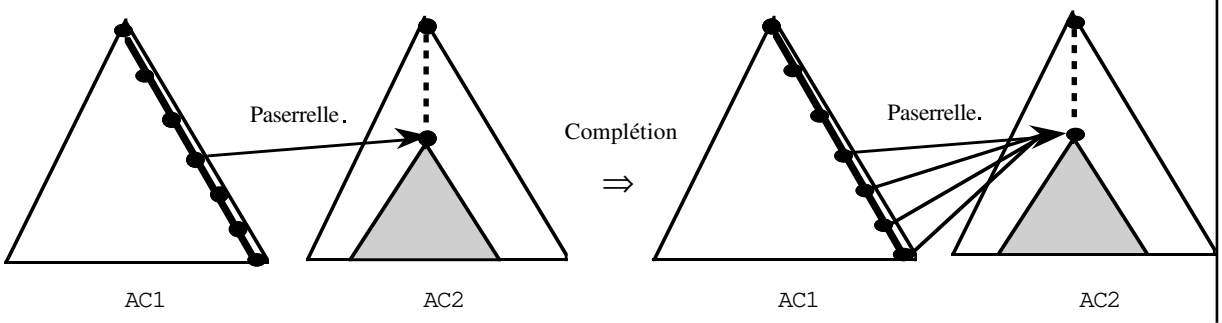


Figure 1.20 : La complétion d'un EPV.

d EPV cmp(S) = <B,P1> est la complétion de S, ssi :

$$P1 = P \cup \{<X^i_{\alpha\gamma}, X^j_{\beta}> \mid <X^i_{\alpha}, X^j_{\beta}> \in P \ \wedge \ \alpha\gamma \in Pos(A^i)\}$$

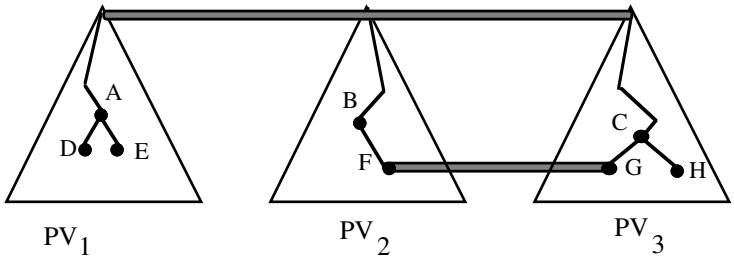


Figure 1.21 : Un concept formé par trois points de vue et deux passerelles. La première passerelle (PA_1) entre les classes racine et la deuxième (PA_2) entre les classes F/PV_2 et G/PV_3.

Pour la figure 1.21, s = <{PV_1,PV_2,PV_3},{PA_1,PA_2}>. s est inconsistant, si par exemple pendant la classification d'un objet x dans le point de vue PV_3, on arrive à la classe H, et à travers de la passerelle PA_2 on arrive en même temps à G.

L'utilisation des passerelles dans un EPV redondant peut entraîner des cycles pendant le processus de classification. Néanmoins, ces cycles sont interdits pour la relation atteignable ($\rightarrow^*_R$), car si le processus de classification se trouve dans l'état atteignable

$$<\ldots, X^i_{\alpha\beta}, \ldots, X^j_{\gamma\delta}, \ldots>$$

et qu'il existe les passerelles $<X^i_{\alpha\beta}, X^j_{\gamma}>$, $<X^j_{\gamma\delta}, X^i_{\alpha}> \in P$, alors les états

$$<\ldots, X^i_{\alpha}, \ldots, X^j_{\gamma\delta}, \ldots>, \quad <\ldots, X^i_{\alpha}, \ldots, X^j_{\gamma}, \ldots>, \quad <\ldots, X^i_{\alpha\beta}, \ldots, X^j_{\gamma}, \ldots>,$$

ne sont pas atteignables, en accord avec la définition 5b.

Une conclusion des définitions précédentes est : la classification d'un objet dans un AC ne peut jamais arriver à un état ambigu. Par contre, la classification d'un objet dans un EPV inconsistant peut arriver à un état ambigu. Par exemple, cette ambiguité peut être produite pour l'utilisation des passerelles pendant la classification.

Remarques

- Un système complet utilise toujours l'information des passerelles.
- Si l'ensemble de passerelles d'un EPV est vide, alors le EPV est trivialement complet.
- La complétion d'un EPV consistant est aussi consistante.
- Il est clair que la complétion d'un EPV est toujours un EPV complet, car le fait d'ajouter de nouvelles passerelles pour la complétion n'ajoute pas de nouvelle connaissance au EPV.
- La classification d'un objet dans une base de connaissances consistante finit toujours en un tuple.

Le théorème suivant montre l'importance de la complétion et de la consistance d'un EPV. Le théorème montre que le choix du prochain point de vue pendant la classification n'a aucune importance.

Théorème 1

Soit $S = <B,P>$ un EPV consistant et complet, alors $\forall x \in X$ il n'existe qu'une S–classification.

Preuve

Soit $x \in X$:

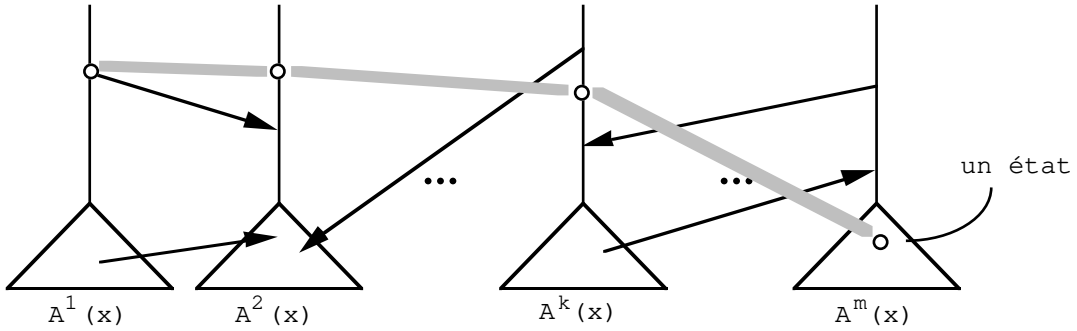


Figure 1.22 : Pour chaque AC^k, la trace est $A^k_\tau(x)$ et les flèches sont les passerelles. ($\rightarrow$ est une passerelle)

Un état e atteignable est représenté pour un tuple d'univers (de classes) des FCs de chaque AC du EPV (figure 1.22). C'est-à-dire, un tuple est un ensemble des classes d'appartenance (la classe actuelle) de x dans chaque point de vue du concept.
Un état atteignable à partir de l'état e, correspond :

- soit à descendre verticalement dans une ou plusieurs composantes du tuple. La descente verticale veut dire que dans un AC l'instance est descendue à une sous-classe de la classe actuelle.
- soit à utiliser une passerelle ou plusieurs pour changer les composantes du tuple. Dans un SC consistant, l'utilisation d'une passerelle entraîne un changement d'une des composantes du tuple.
- soit aux changements simultanés par la descente verticale et par l'utilisation des passerelles.

Soit Q un ensemble fini, $Q = \{<\alpha_1,\ldots,\alpha_m> \mid <X_{\alpha_1},\ldots,X_{\alpha_m}> \in ECA\}$, et soit le graphe $G(Q,.\rightarrow.)$, tel que :

1. $<\alpha_1,\ldots,\alpha_m> \rightarrow <\beta_1,\ldots,\beta_m> \Leftrightarrow <X_{\beta_1},\ldots,X_{\beta_m}>$ est atteignable en partant de $<X_{\alpha_1},\ldots,X_{\alpha_m}>$.

2. La relation $.\rightarrow.$ est définie comme :

$$<\alpha_1,\ldots,\alpha_m> \rightarrow <\beta_1,\ldots,\beta_m> \quad \Rightarrow \quad <\alpha_1,\ldots,\alpha_m> <_p <\beta_1,\ldots,\beta_m>.$$

où $.<_p.$ est l'ordre de préfixes sur $(N_+^*)^m$ ($\forall i$, α_i est un préfixe de β_i). C'est à dire que le graphe G ne peut pas être cyclique, donc chaque séquence d'états atteignables est finie. Les séquences maximales en partant de l'état $<X,\ldots,X>$ finissent toujours en S-classifications.
D'autre part, si nous considérons le graphe G comme un graphe de réduction, la confluence locale entraîne la confluence et la canonicité. Etant donné que pour un système de réécriture R noethérien : R est confluent si et seulement si il est localement confluent. Donc, l'ensemble de S-classifications doit être unitaire, car il correspond aux formes normales de la réduction. La confluence locale d'un système S peut être montrée pour la convergence des paires critiques de S. Après le théorème de G. Huet, un système de réécriture est localement confluent si et seulement si toutes ses paires critiques convergent.
Soit $a,b,c \in Q$, tels que $a \rightarrow b$, $a \rightarrow c$. Il faut montrer que b et c convergent, c'est à dire qu'il existe un w tel que $b \rightarrow_R^* w$ et $c \rightarrow_R^* w$. La figure 1.23 illustre les différentes situations :

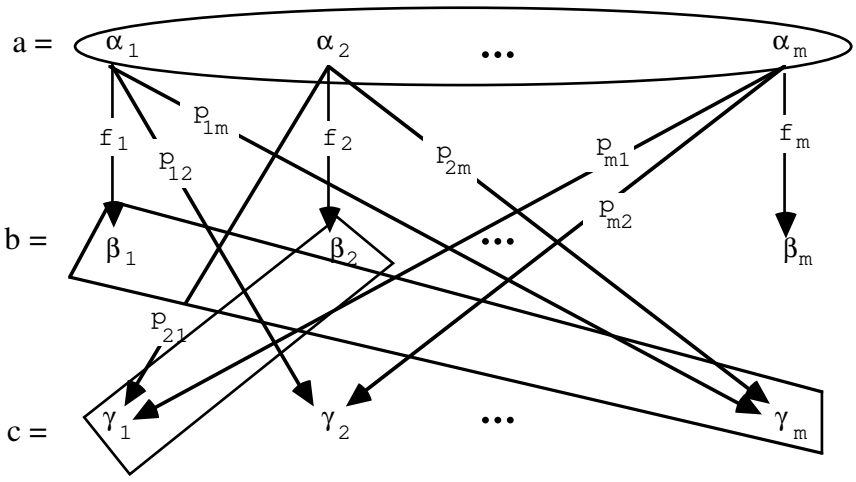


Figure 1.23 : Les paires critiques pendant la classification. Par exemple a → b et a → c.

En partant de $a = \langle \alpha_1, \ldots, \alpha_m \rangle$, les réductions possibles de a,b et c sont décrites pour les tuples dont la composante α_k est changée pour β_k ou pour γ_k, $\forall k=1,\ldots,m$. La composante α_k change à β_k ($\alpha_k \xrightarrow{f_k} \beta_k$) si on utilise la fonction de classification f_k (plus précisément, la fonction f_{α_k}); le changement à γ_k peut être justifié par l'existence d'une passerelle p_{ik}, avec $i \neq k$. Dans la figure 1.24, il y a des passerelles qui ont été rajoutées lors de la complétion. Finalement, s'il n'y a pas de changements en la composante k, leur justification est la fonction identité id_k.

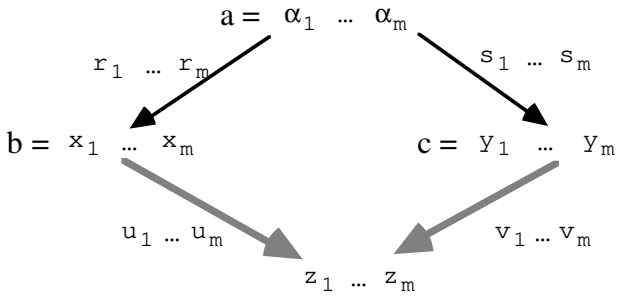


Figure 1.24 : Convergence des paires critiques b et c.

Donc, b est égal à $<x_1,\ldots,x_m>$ et c est égal à $<y_1,\ldots,y_m>$. Les étiquettes $<r_1,\ldots,r_m>$ qui nomment les arcs connectant a et b sont les justifications des changements dans les composantes. Elles sont définies comme :

r_k $= id_k$, si $x_k = \alpha_k$

$= f_k$, si $x_k = \beta_k$

$= p_{jk}$, si $x_k = \gamma_k$. Dans ce cas, p_{jk} existe (une passerelle).

De la même façon sont définies les étiquettes $<s_1,\ldots,s_m>$, qui justifient les changements de a à c.

Maintenant, il faut trouver un tuple $d=<z_1,\ldots,z_m>$ pour lequel b et c convergent. Pour cela, nous exprimons comment se transforment, à travers des réductions, les composantes des deux tuples. Les étiquettes $<u_1,\ldots,u_m>$ et $<v_1,\ldots,v_m>$ sont utilisées pour montrer les transformations possibles (table 1.2).

x_k	y_k	u_k	v_k
α_k	α_k	id_k	id_k
α_k	β_k	f_k	id_k
α_k	γ_k	p_{jk}	id_k
β_k	α_k	id_k	f_k
β_k	β_k	id_k	id_k
β_k	γ_k	p_{jk}	id_k
γ_k	α_k	id_k	p_{jk}
γ_k	β_k	id_k	p_{jk}
γ_k	γ_k	id_k	id_k

Table 1.2 : La définition de u_k et v_k, pour $k=1,\ldots,m$,, pour les possibles valeurs de x_k y y_k.

Il faut noter l'utilisation de passerelles, qui n'ont pas été dessinées dans la figure 1.23. Cette utilisation est justifiée par l'ajout des passerelles lors de la complétion de EPV. C'est le cas de la passerelle p_{1m}, qui part de β_2 vers γ_m.

Il est trivial de montrer que $b \rightarrow_R^* d$ y $c \rightarrow_R^* d$. Donc, G est localement confluent, et étant donné que le EPV est noethérien, EPV est confluent (le lemme de Newmann pour les systèmes de réécriture). □

Nous avons analysé la correction et la complexité des algorithmes qui font la classification des instances dans un EPV. La définition suivante ne concerne que les algorithmes n'utilisant que les fonctions de classification définies pour EPV.

Définition 7

Soit S(X) un EPV et le graphe $G(ECA, .\rightarrow_R^* .)$.

- Pour $x \in X$, une S-trace de classification pour x, notée S(x), est un chemin de longueur maximal en G et dont l'état initial est <X,…,X>. Si S(x) est fini, alors S(x) finit en une S-classification.
- soit H un ensemble de S-traces : H est un algorithme de classification pour S, ssi $\forall x \in X$, il existe une S-trace $S(x) \in H$.
- Un résultat H(x) est une S-classification pour x. Donc, H(x) est un état terminal d'une S-trace $S(x) \in H$.
- Un algorithme de classification H est déterministe si pour chaque $x \in X$, il n'existe qu'une S-trace $S(x) \in H$.

Théorème 2

La complexité d'un algorithme de classification H est le maximum des longueurs des traces qui appartiennent à H : complexité(H) = maximum{|t| | t∈H , t est une trace}.

Preuve :

C'est une conséquence du Théorème 1. []

Cette complexité est la plus petite possible. L'algorithme parallèle donné dans le chapitre 5 a cette complexité.

Théorème 3

Pour un EPV consistant et complet, un algorithme de classification ne produit qu'une S-classification.

Preuve :

C'est une conséquence du Théorème 1. []

La conséquence de ce théorème est que : la classification d'un objet dans une base de connaissances est toujours la même, si l'information initiale sur l'objet est la même.

Théorème 4

Pour un EPV consistant, il existe un algorithme de classification qui produit comme résultat la S-classification de x, qui est produit pour n'importe quel algorithme sur le EPV complet.

Preuve :

L'algorithme doit utiliser pour la classification d'abord les passerelles, s'il peut le faire. Cette utilisation est équivalente à simuler la complétion. []

1.5 Conclusions

Dans ce chapitre ont été présentés d'abord le modèle de connaissances TROPES, puis une formalisation du modèle. Le résultat le plus important de cette formalisation est la convergence des résultats des différents algorithmes de classification d'une instance en TROPES, en partant de la même connaissance et avec la même base de connaissances pour tous. Ainsi, ces classes minimales, déterminées lors la classification d'une instance dans les différents points de vue ne dépendent pas de l'algorithme mais bien de la connaissance sur l'instance. Donc, le choix du point de vue n'a aucune importance pour la classification, sauf pour le modèle séquentiel dans lequel la performance de la classification peut être améliorée selon le point de vue choisi. Nous avons montré aussi que l'utilisation de passerelles pendant la classification n'entraîne pas un changement de la classification en elle-même. Elle peut améliorer la performance du processus de classification et aider l'utilisateur à classer l'instance grâce aux inférences produites quand une passerelle est utilisée pour passer d'un point de vue à un autre.

2. MODELE PARALLELE DE TROPES

Dans le modèle de TROPES, il existe plusieurs sortes d'opérations parmi lesquelles : la classification de classes[4], le filtrage, l'exécution de tâches, la classification des objets, etc. Dans ce travail nous ne traitons que la classification en parallèle des objets.
Dans ce chapitre nous présentons un modèle de classification en parallèle dans un modèle multi-points-de-vue. Le but de cette classification est la localisation en parallèle d'un objet dans les différentes perspectives d'un concept, dans le modèle TROPES.
Le chapitre commence par montrer que la classification des objets dans TROPES est parallélisable. Puis, un algorithme parallèle pour la classification est défini en partant de l'algorithme séquentiel de classification de TROPES présenté dans le chapitre précédent.
Dans la première version du modèle de classification en parallèle le raisonnement hypothétique n'est pas considéré. Celui-ci est pris en compte dans une extension du modèle.

2.1. Classification en parallèle d'un objet dans TROPES

Le but de cette section est de montrer qu'il est possible de paralléliser la classification multi-points-de-vue dans TROPES. Pour cela, nous montrons d'abord que le calcul des contraintes pour les différents attributs définis peut être fait de façon "indépendante" (ce concept d'indépendance sera détaillé plus tard.) dans les différents points de vue d'un concept. Puis, nous décrivons les différentes parties de l'algorithme lorsqu'elles s'exécutent en parallèle.
Par la suite, nous allons montrer que le calcul de contraintes pour les différents attributs dans un point de vue, lors de la classification en parallèle d'un objet, peut être fait de façon indépendante par rapport aux autres points de vue, même s'il existe des passerelles entre eux. Il y a indépendance en le sens qu'une fois que l'utilisateur a donné dans un point de vue la valeur v_X pour un attribut a_X, et que cette valeur satisfait les contraintes t_X de l'attribut, la valeur sera toujours la même. Cela veut dire qu'au fur et à mesure que les autres points de vue demandent la valeur pour le même attribut a_X, le système rend toujours la même valeur : la valeur v_X donnée par l'utilisateur lors de la première requête.
Il existe deux cas pour le calcul de contraintes dans un point de vue lors de la classification en parallèle d'un objet : soit il n'existe que la classe racine comme passerelle entre les points de vue du concept, soit il existe des passerelles différentes de la classe racine entre les différents points de vue.
La demande de valeurs n'est faite que pour les procédures d'obtention d'information et d'appariement. Pour la procédure d'obtention d'information, l'utilisateur n'est autorisé à donner comme valeur d'un attribut que des valeurs qui satisfont les contraintes établies pour cet attribut dans la classe actuelle du point de vue actuel. Cette demande est faite afin d'obtenir toutes les valeurs pour les attributs définis dans la classe actuelle. Pour l'appariement, l'utilisateur peut donner une valeur quelconque et elle est acceptée si celle-ci satisfait les contraintes d'une des sous-classes de la classe actuelle. Sinon, la sous-classe que le système est en train de regarder va devenir "impossible".
Dans la suite, la classification ne se fait que pour un objet à la fois, donc le système ne peut avoir qu'une interaction à la fois. Il doit donc exister un contrôle du dialogue pour gérer les requêtes à l'utilisateur et les réponses de celui-ci. Ce contrôle peut être très simple ou très complexe. Un exemple du contrôle très simple est : le contrôle se réduit à demander les valeurs à l'utilisateur au fur et à mesure que les différents points de vue en ont besoin. Puis les valeurs sont retournées sans vérifier si elles satisfont ou non les contraintes des attributs.
Avec un contrôle aussi simple que celui-ci, nous pouvons arriver à des incohérences si deux points de vue demandent la valeur pour le même attribut, et que l'utilisateur donne deux réponses différentes. D'autre part, plusieurs demandes d'une valeur pour un même attribut peuvent être ennuyeuses pour l'utilisateur. Donc, il faut faire une interaction beaucoup plus intelligente entre l'utilisateur et le système.
Pour développer le contrôle de dialogue il faut prendre en compte différents aspects :

- au fur et à mesure que l'utilisateur donne des valeurs pour les différents attributs, le contrôle de dialogue peut les envoyer à tous les points de vue, même s'ils n'ont rien

---

[4] La classification de classes est une opération de construction ou maintien de la base de connaissances, qui travaille au niveau de la structure de classes; classer une classe exige la modification des liens entre les classes, le contrôle de la cohérence de la base et éventuellement la modification des instances déjà classées [CAP 93].

demandé. Il existe une raison pour cette diffusion : la procédure d'appariement ordonne les questions à poser à l'utilisateur de façon à optimiser les inférences que le système peut tirer des réponses données. Donc, la procédure peut utiliser d'abord les valeurs déjà données par l'utilisateur lors de la demande pour un autre point de vue. De plus, cette diffusion va éviter des communications ultérieures entre les points de vue et le contrôle de dialogue : les points de vue n'ont plus besoin de demander au contrôle de dialogue (CD) la valeur d'un attribut qui a été déjà donnée par l'utilisateur.

- Lors de la demande d'une valeur pour un attribut, le CD valide cette valeur par rapport aux contraintes : si la valeur satisfait les contraintes, le CD retourne la valeur au point de vue qui l'a demandée. Si la valeur ne satisfait pas les contraintes, le point de vue va demander encore une fois la valeur au contrôle de dialogue, et celui-ci à l'utilisateur. Ce cycle sera fait jusqu'à ce que la valeur satisfasse les contraintes. Cette façon de mener le contrôle de dialogue entraîne l'augmentation de la communication entre le CD et les points de vue. Si le contrôle de dialogue lui-même fait la validation de la valeur pour un attribut par rapport à ses contraintes, il n'y aura qu'une communication entre le CD et le point de vue qui a demandé la valeur, une fois que la valeur a été acceptée.
- à partir du moment où le système a demandé une valeur pour un attribut a à l'utilisateur et que celui-ci l'a donnée, chaque fois qu'un autre point de vue demande la valeur pour le même attribut, le système doit retourner la même valeur initiale sans rien demander à l'utilisateur. Il faut rappeler que lorsque deux classes d'appartenance d'un objet font référence au même attribut a, sa valeur dans l'objet doit satisfaire les contraintes des deux classes.
- le système n'a pas toujours besoin de valider les valeurs que rend l'utilisateur lors d'une demande. C'est le cas si la demande de la valeur pour un attribut d'une classe de PVi a été faite pour la procédure d'appariement. Dans ce cas-là, l'utilisateur peut donner une valeur qui ne satisfait pas les contraintes pour l'attribut dans cette classe qui deviendra alors "impossible".
- dans le cas où au moins deux demandes arrivent en même temps au contrôle de dialogue, le système ne fait qu'une requête à l'utilisateur. Pour la satisfaction des contraintes, le système valide la valeur donnée par l'utilisateur avec les contraintes de toutes les demandes faites par une procédure d'obtention d'information.
- d'autre part, lorsqu'une procédure d'obtention d'information demande une valeur pour l'attribut a, et que cette valeur a déjà été donnée, il est impossible que cette valeur ne satisfasse pas les nouvelles contraintes, étant donné que la base de connaissances sur laquelle se fait la classification doit être cohérente.
- enfin, le contrôle de dialogue garde pour chaque attribut dont le système a demandé la valeur, toutes leurs contraintes, afin que l'utilisateur puisse voir tous les résultats; il peut ainsi suivre pas-à-pas la descente de l'objet dans chaque point de vue.

Le contrôle de dialogue est visualisé dans la figure 2.1 :

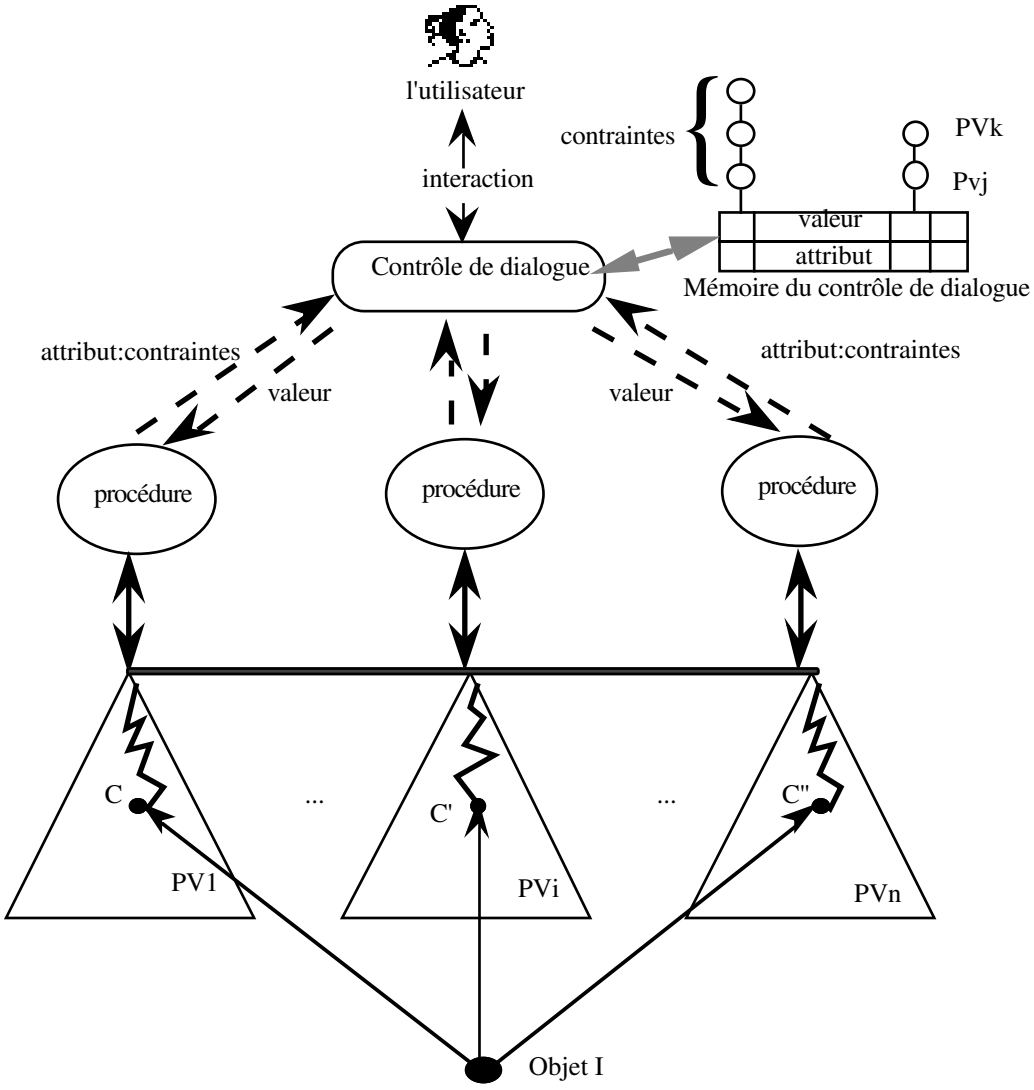


Figure 2.1 : Les procédures d'obtention d'information et d'appariement peuvent demander les valeurs pour les différents attributs dont ils ont besoin, afin de faire descendre l'objet le plus bas possible dans les différents points de vue.

Pour montrer que le calcul de contraintes lors de la classification en parallèle d'un objet peut être fait de façon indépendante entre les différents points de vue, nous allons partir de trois faits :

- la base de connaissances définie par l'utilisateur est cohérente.
- une fois que l'utilisateur a donné une valeur pour un attribut, cette valeur reste toujours la même. Celle-ci peut être utilisée pour les procédures d'obtention d'information et d'appariement dans les différents points de vue. Si la procédure d'obtention d'information demande la valeur déjà connue pour un attribut dans une classe C du point de vue PVi, cette valeur doit valider leur contraintes, étant donné que la classe C dans PVi est actuelle, (donc "sûre").
- tous les points de vue connaissent les valeurs des attributs qui ont déjà été demandées à l'utilisateur. Nous avons dit qu'au fur et à mesure que l'utilisateur fournissait des valeurs pour les différents attributs, celles-ci étaient diffusées par le CD.

Donc, pour le calcul des contraintes d'un attribut de façon indépendante dans les différents points de vue, il existe trois possibilités:

- la procédure qui demande la valeur est l'obtention d'information : dans ce cas-là, le système est en train de compléter l'information pour la classe actuelle, et les valeurs pour les différents attributs doivent satisfaire leurs contraintes. Ces contraintes sont calculées

en parcourant les liens sorte-de (du point de vue actuel) à partir de la classe actuelle et jusqu'à la racine. Si la valeur est déjà connue, elle satisfait les contraintes de l'attribut. Si la valeur n'est pas connue, elle est demandée à l'utilisateur. Cette valeur doit être valide, et à partir de ce moment tous les autres points de vue vont la connaître.

- la procédure qui demande la valeur est l'appariement : dans ce cas-là, le système est en train de choisir une classe "sûre" parmi les sous-classes de la classe actuelle C. Donc les valeurs pour les différents attributs peuvent ou non satisfaire les contraintes des attributs dans les sous-classes. Ces contraintes sont calculées en ajoutant aux contraintes déjà calculées pour C les nouvelles contraintes des attributs dans les sous-classes de C. Si la valeur ne satisfait pas les contraintes de l'attribut, alors la sous-classe devient "impossible". Si la valeur n'est pas connue, elle est demandée à l'utilisateur, et à partir de ce moment tous les autres points de vue vont la connaître.
- enfin, si le système arrive à un point de vue PVj par une passerelle bi-directionnelle C dont l'origine est le point de vue PVi, les valeurs déjà connues pour les attributs dans PVi doivent satisfaire toutes les contraintes des attributs dans C/PVj étant donné que la classe représentée pour C/PVi et C/PVj est la même. A partir de ce moment, toutes les valeurs sont fixées pour continuer la classification dans PVj. Donc, le calcul de contraintes dans PVj peut être toujours effectué de façon indépendante. Il faut noter qu'il n'est pas possible d'arriver à une contradiction, si le système descend dans le point de vue PVj vers une sous-classe D et demande les valeurs manquantes des attributs dont il a besoin. Car les sur-classes de D dans PVj sont "sûres" et leurs sur-classes dans PVi à travers la passerelle C sont aussi "sûres".

  Si la passerelle est unidirectionnelle, nous pouvons faire la même analyse que dans le cas précédent, étant donné que la classe à laquelle le système arrive est du point de vue "ensembliste" un sous-ensemble de la classe origine de la passerelle.

Le fait que le calcul de contraintes dans un point de vue puisse être fait de façon indépendante permet d'éviter trop de communication entre les différents points de vue à cause de la diffusion des contraintes d'un attribut dans C/PVi à tous les autres points de vue.
Ci-dessus nous avons développé une façon de calculer les contraintes dans les différents points de vue lors de la classification d'un objet. Dans la prochaine section nous montrons que la classification en parallèle d'un objet dans TROPES est possible en utilisant les résultats précédents. Pour cela nous partons de l'algorithme de classification multi-points-vue développé par Mariño [MAR 89].

2.2. Algorithme de Classification en parallèle

Dans cette section est présenté l'algorithme parallèle de classification d'objets dans TROPES à partir de l'algorithme séquentiel (algorithme 2.1). Le but de cette classification est la localisation de façon parallèle d'un objet dans les différents points de vue d'un concept.

```
tant que (non condition pour finir la classification)
        obtention d'information
        appariement
        propagation de marques
        actualisation d'information
        choix du prochain point de vue
fin tantque
```

Algorithme 2.1 : Cet algorithme est composé de cinq pas, qui ont été décrits dans la section précédente : obtention d'information, appariement, propagation de marques, actualisation d'information et choix du prochain point de vue.

La première partie présente le problème du choix du prochain point de vue pour la classification. La deuxième partie montre l'exécution en parallèle de plusieurs processus de la même sous-procédure de l'algorithme de classification (un sur chacun des points de vue du concept). Dans ce cas, nous pouvons dire que dans tous les points de vue s'exécute, par exemple, la procédure de propagation de marques. Enfin, la dernière partie présente l'exécution en parallèle de plusieurs processus de toutes les sous-procédures de la classification, un ou plus

sur chacun des points de vue. Dans cette partie, contrairement à la section précédente, la procédure de propagation de marques peut s'exécuter dans un point de vue tandis que la procédure d'obtention d'information s'exécute dans un autre. De même, la procédure d'obtention d'information et la procédure de propagation de marques peuvent s'exécuter simultanément dans le même point de vue.

2.2.1. Le choix d'un point de vue

Dans l'algorithme séquentiel de classification de TROPES, le choix du prochain point de vue est important, étant donné que celui-ci calcule le point de vue à prendre comme point de vue actuel pour la prochaine itération du cycle. Après la mise à jour de l'information, le système choisit le nouveau point de vue à considérer. Ce choix dépend du type de point de vue actuel (principal ou auxiliaire), et de son état (actif ou inactif).
Il existe plusieurs critères pour le choix d'un point de vue [MAR 89], parmi lesquels :

- choix du point de vue ayant la passerelle la plus basse vers un point de vue principal.
- choix du point de vue permettant de retourner le plus vite possible à un point de vue principal.
- choix à partir des listes des points de vue données pour l'utilisateur. C'est-à-dire que l'utilisateur ordonne les points de vue selon sa connaissance sur eux. Ce dernier critère a été choisi pour l'algorithme de classification séquentielle, et il prend en compte l'ordre pour choisir le "meilleur" point de vue.

Ces critères sont utiles lors de la classification "séquentielle", étant donné qu'il ne peut exister qu'un point de vue actuel à la fois : le point de vue sur lequel s'exécute à un moment déterminé l'algorithme de classification.
Nous montrons que le choix du point de vue n'a aucune importance pour le processus de classification. En effet pour classer un objet dans un concept et avec une connaissance fixée, l'algorithme arrive toujours aux mêmes classes dans les différents points de vue du concept (en concordance avec le théorème 1) indépendamment de l'arbre de choix des points de vue. Bien que, le choix du point de vue puisse améliorer la performance de l'algorithme de classification séquentiel, cette amélioration ne se présente pas dans le cas de l'algorithme parallèle.
Dans l'algorithme parallèle, le choix d'un point de vue devient très simple, car tous les points de vue "principaux" et "auxiliaires" sont des points de vue actuels. Autrement dit, la classification se déroule de façon parallèle dans tous les points de vue à la fois. Si à un certain moment, cette classification s'avère impossible sous un point de vue, ce point de vue est placé "en attente", et il doit attendre que le système revienne sur lui par une passerelle (figure 2.2).

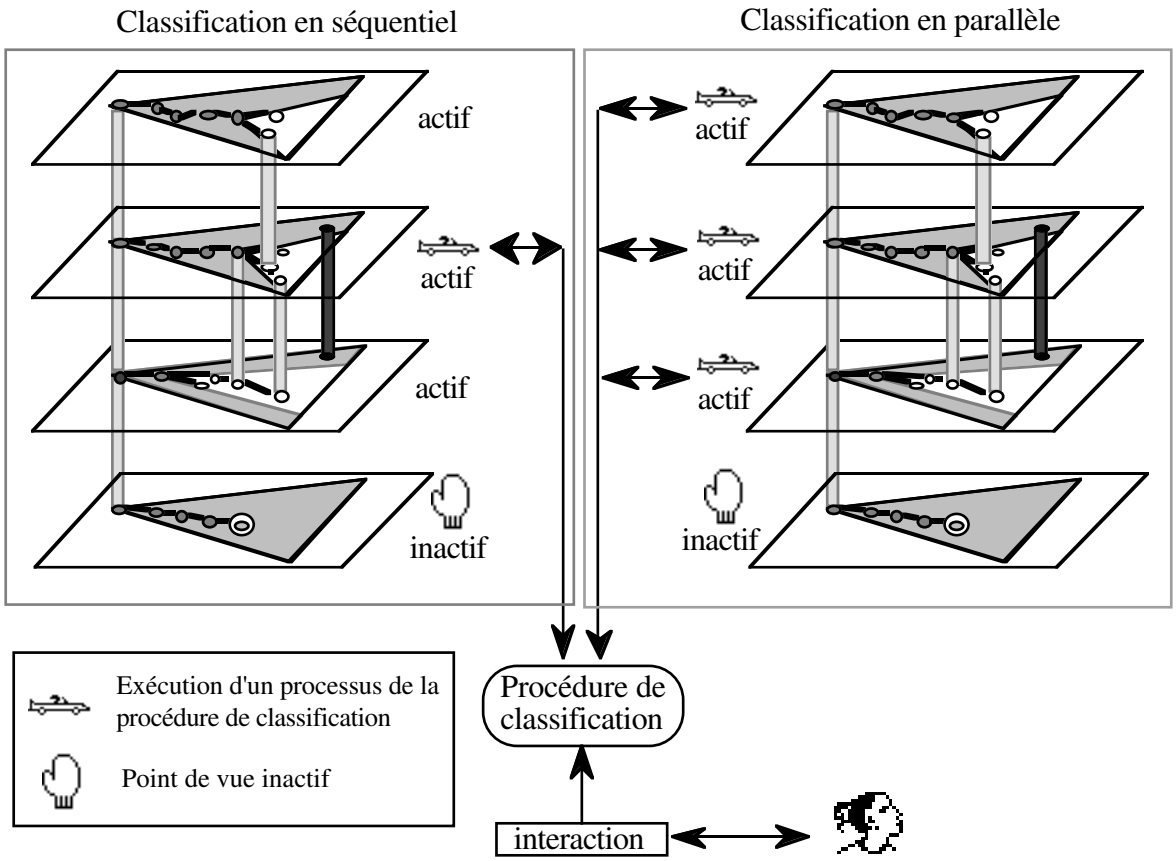


Figure 2.2 : Exécutions en séquentiel et en parallèle de la procédure de classification d'un objet dans TROPES. Pour le modèle séquentiel, l'objet n'est classifié que dans un point de vue à la fois, tandis que pour le modèle parallèle, l'objet est classifié dans tous les points de vue "actifs" simultanément.

D'autre part, le problème de terminaison dans l'algorithme séquentiel est très simple. Il consiste à vérifier les conditions de sortie du cycle, parmi lesquelles se trouve la décision de l'utilisateur d'arrêter la classification ou le fait que tous les points de vue deviennent "inactifs". Par contre, cette tâche dans l'algorithme parallèle devient plus complexe que celle de l'algorithme séquentiel.
Dans le cas du déroulement en parallèle de la classification, le problème de terminaison devient complexe. Il s'agit ici d'un problème de terminaison dans une architecture parallèle, dans laquelle tous les processeurs sont probablement asynchrones. Il y a aussi un problème de communication entre les différents processus-processeurs afin d'arriver à une solution cohérente du problème. Autrement dit, le choix de la méthode de synchronisation et de communication entre les processeurs est un problème de contrôle [QUI 89].
Ce problème de contrôle est fondamental pour les architectures parallèles. Il est tout d'abord possible de placer un processeur chef, appelé unité de contrôle, comme dans l'organisation SIMD (Annexe 3). Ce mode de contrôle est hérité du modèle Von Neuman et souffre d'un défaut : la lenteur.
C'est pour cette raison que beaucoup de systèmes prônent un contrôle décentralisé: où chaque processeur est maître de ses décisions, qu'il prend en général en consultant ses plus proches voisins. Cette approche présente quelques difficultés: la première est que tout programme que l'on veut exécuter de cette façon doit être conçu d'une façon très différente de celle d'un programme séquentiel. La raison en est qu'aucun processeur ne dispose d'une vue d'ensemble de ce qui se passe, c'est-à-dire de l'état global du problème. Ainsi, la simple détermination de l'instant où le programme est terminé doit être obtenue par une succession de décisions à caractère local.
Pour notre problème, un contrôle décentralisé pose plusieurs sous-problèmes :

- Les problèmes de synchronisation: en général, on va traiter les machines à caractère asynchrone. Il faudra donc avoir une synchronisation entre tous les processeurs afin de savoir lesquels sont occupés dans la réalisation d'une tâche et lesquels sont libres et peuvent être utilisés pour réaliser une autre tâche.
- La synchronisation entre les différentes sous-procédures de l'algorithme de classification.
- Le maintien de la cohérence de la classification de l'objet en même temps dans différents points de vue, et la mise à jour de l'objet avec ses classes de rattachement. Pour cela il faudra avoir des mécanismes permettant de gérer les problèmes de concurrence sur les mêmes données.
- L'exclusion mutuelle entre les processeurs: ce problème se pose lorsque deux processeurs ont besoin d'une même ressource.
- La détection de la terminaison des tâches des différents processeurs.

L'autre problème qui se pose est la communication entre les différents processus (qui font partie de la tâche pour classer un objet dans le concept). Ceci peut se faire de deux façons différentes :

- Centralisée, où un processus superviseur reçoit tous les messages et se charge de les faire parvenir à leur destinataire.
- Distribuée, où chaque processus fait parvenir son message à son destinataire. Nous faisons ici abstraction du routage des messages.

Nous reviendrons sur ces sujets, le contrôle pour la terminaison de tâches et le problème de la communication, dans le chapitre 6.

2.2.2. Exécution en parallèle d'instances de la même procédure

Cette section présente la classification en parallèle d'un objet entre les différents points de vue. Le principe est d'exécuter la même sous-procédure dans tous les points de vue "actifs" de façon parallèle, sans risque d'avoir une incohérence dans le résultat. A la fin de la classification l'objet doit être rattaché dans chacun des points de vue à une classe d'appartenance. Cette classe doit être minimale au sens où elle est la classe la plus basse dans la hiérarchie de classes du point de vue, pour laquelle l'objet satisfait toutes les contraintes.
La section est composée de quatre parties, une pour chacune des sous-procédures de la classification : obtention d'information, appariement, propagation de marques et mise à jour d'information. Par exemple, le système peut exécuter la procédure d'obtention d'information dans tous les points de vue à la fois.

2.2.2.1. Obtention d'information

D'après la section 2.1, le calcul des contraintes pour les attributs dans les différents points de vue est fait de façon indépendante, car il est possible d'exécuter en parallèle la procédure d'obtention d'information dans tous les points de vue "actifs".
Comme nous l'avons dit au début du chapitre, grâce à la procédure d'obtention d'information, le système obtient de l'utilisateur l'information manquante sur la plus basse classe "sûre" d'un point de vue PVi. Etant donné que la classe est "sûre", l'utilisateur n'est autorisé à donner que des valeurs valides pour les attributs de cette classe.
La procédure d'obtention d'information s'exécutant en parallèle dans tous les points de vue "actifs", le système peut demander plusieurs valeurs pour des attributs différents ou plusieurs fois la valeur pour le même attribut en même temps.
Etant donné qu'il n'y a qu'un utilisateur à la fois qui peut faire la classification d'un objet dans TROPES, le système ne peut poser qu'une question à la fois. Donc, au fur et à mesure que les processus de la procédure d'obtention d'information (par la suite, le processus de cette procédure d'obtention d'information agissant sur le point de vue PVj va être noté POIj) demandent des valeurs pour les différents attributs, le système, à travers un contrôle de dialogue (CD) centralisé, va poser les questions une à une à l'utilisateur.
Si le système demande à l'utilisateur plusieurs fois la valeur pour le même attribut, l'utilisateur peut donner des valeurs différentes à chaque fois, et le système risque d'arriver donc à une incohérence. Car, il ne peut exister qu'une valeur possible pour chaque attribut, étant donné que le concept dans lequel se fait la classification de l'objet doit être cohérent. Il faut rappeler que si deux classes d'appartenance de l'objet font référence au même attribut, sa valeur doit satisfaire les contraintes des deux classes.

Donc, pour résoudre ces problèmes, le CD garde les questions posées pour les différents POI dans une liste selon leur ordre d'arrivée. Chaque nœud de la liste de questions est composé de cinq parties: attribut, contraintes, la classe, PVi et POIj. S'il existe déjà une entrée dans cette liste pour le même attribut, alors le CD calcule l'ensemble des contraintes à partir des contraintes déjà existantes et les contraintes de la nouvelle requête. Ce résultat est laissé à la place des contraintes déjà existantes dans la liste afin que, lors de la demande de la valeur à l'utilisateur, le CD lui montre toutes les contraintes de l'attribut et valide la valeur donnée par l'utilisateur par rapport à ce nouvel ensemble de contraintes.
Lors de la demande d'une valeur à l'utilisateur par le CD, celui-ci montre à l'utilisateur les contraintes propres à l'attribut afin qu'il donne une valeur valide. Le CD valide donc la valeur donnée par l'utilisateur en la comparant avec les contraintes de l'attribut :

- si la valeur ne les satisfait pas, le système demande encore la valeur pour le même attribut jusqu'à ce que l'utilisateur donne une valeur valide.
- si la valeur est valide, le CD la retourne au POIj qui l'avait demandée, et fait une diffusion de la valeur vers tous les POI pour gagner du temps, si jamais un autre POIi a besoin de la valeur pour le même attribut. Cette diffusion est faite parce que le concept est cohérent. Et ainsi une fois que l'utilisateur a donné une valeur pour l'attribut, cette valeur reste fixe.
- après la diffusion de la valeur v_X pour un attribut a_X, si un POIj a besoin de la valeur pour a_X, il doit utiliser la valeur fixée v_X, qui doit satisfaire les contraintes de l'attribut a_X. Cette validation de la valeur v_X par rapport aux contraintes de l'attribut a_X est faite au niveau local, sans rien demander au CD.

Il faut remarquer que le CD a une structure de l'objet dans laquelle existent toutes les contraintes dérivées pour chacun des attributs dans toutes les classes auxquelles a été rattachée l'objet. Cette structure contient également toutes les valeurs données par l'utilisateur pour les différents attributs et les classes d'appartenance de l'objet dans tous les points de vue. Toute cette information est mise à jour au fur et à mesure que les différents OPI demandent des valeurs au CD.
Cette structure permet donc d'une part à l'utilisateur de suivre pas-à-pas la descente de l'objet dans chaque point de vue et au système d'expliquer le raisonnement suivi, et d'autre part de simplifier la procédure de mise à jour d'information.
Pour finir, nous allons présenter un résumé de l'exécution en parallèle de la procédure d'obtention d'information.

- Exécution en parallèle des instances de la procédure d'obtention d'information dans les différents points de vue "actifs". Au début tous les points de vue "principaux" et "auxiliaires" sont "actifs", tandis que les points de vue "cachés" ne sont utilisés que comme "ponts" pour les autres points de vue (figure 2.3).

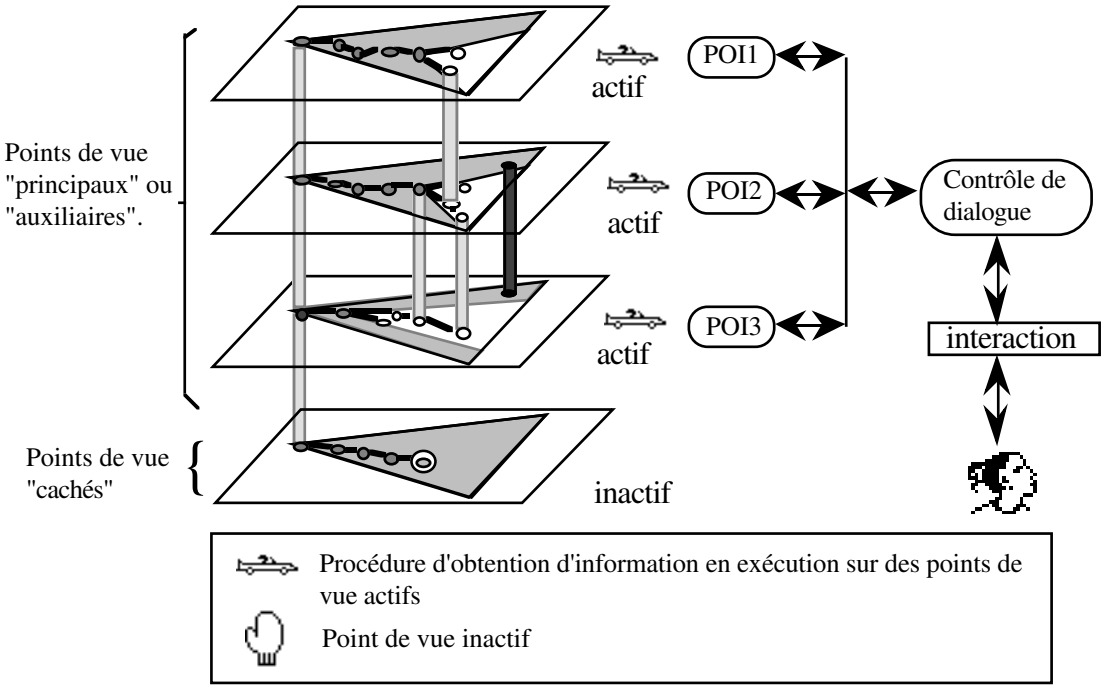


Figure 2.3 : Exécution en parallèle de plusieurs instances de la procédure d'obtention d'information POIj.

- Le contrôle de dialogue CD se fait de façon centralisée. La structure de données du CD a deux sous-structures :
    - la première sous-structure contient une liste avec les requêtes des différents POI (figure 2.4), où chaque nœud de cette liste contient l'attribut, les contraintes pour l'attribut, la Ca/PVi, le PVi et le POIi.

Structure de données : liste de demandes des valeurs

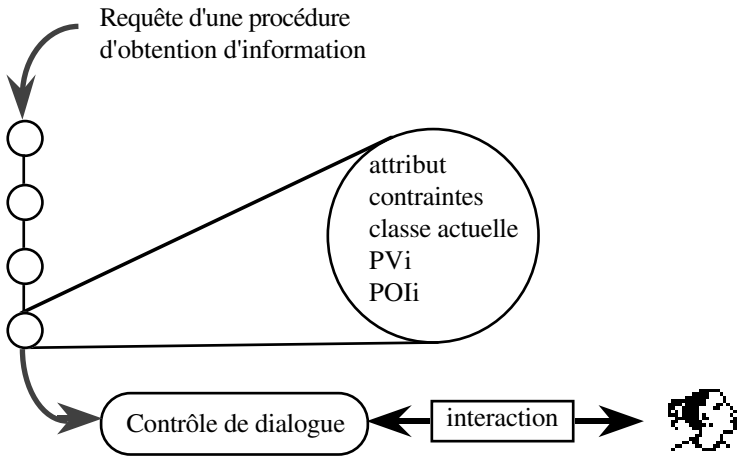


Figure 2.4 : Structure de données pour les requêtes de valeurs pour les différents attributs dont les POIi ont besoin.

- La deuxième sous-structure contient l'objet avec les attributs, les valeurs pour ces différents attributs, les contraintes dérivées pour ces attributs, et les classes d'appartenance de l'objet dans les différents points de vue (figure 2.5).

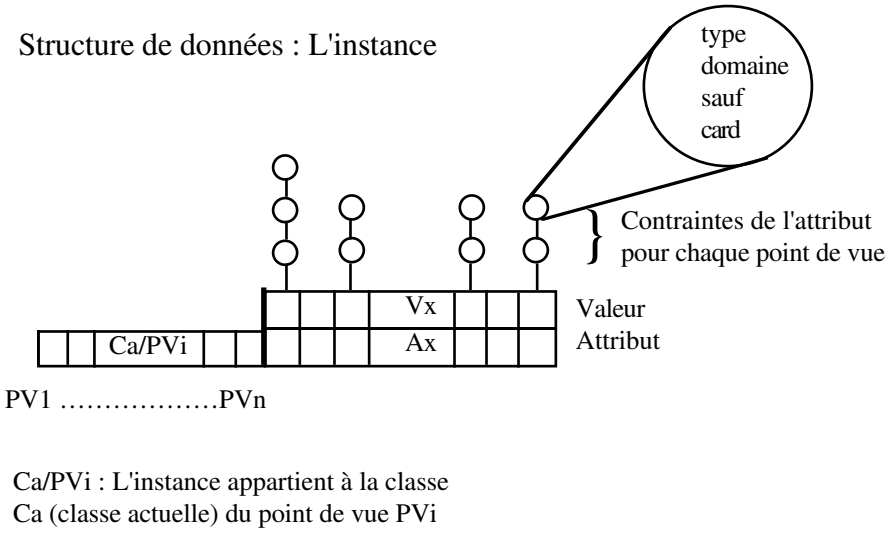


Figure 2.5 : Structure de données pour l'objet. Le contrôle de dialogue garde les valeurs données par l'utilisateur pour les différents attributs, leurs contraintes, etc.

- La procédure POIi demande la valeur encore inconnue pour l'attribut a_X. Pour cela, POIi fait une requête au CD, où la requête a la forme : attribut a_x, contraintes pour a_X, classe d'appartenance Ca (la classe actuelle), point de vue PVi et origine POIi (figure 2.6). Si la valeur est déjà connue, POIi met à jour l'attribut avec la valeur sans rien demander au CD.

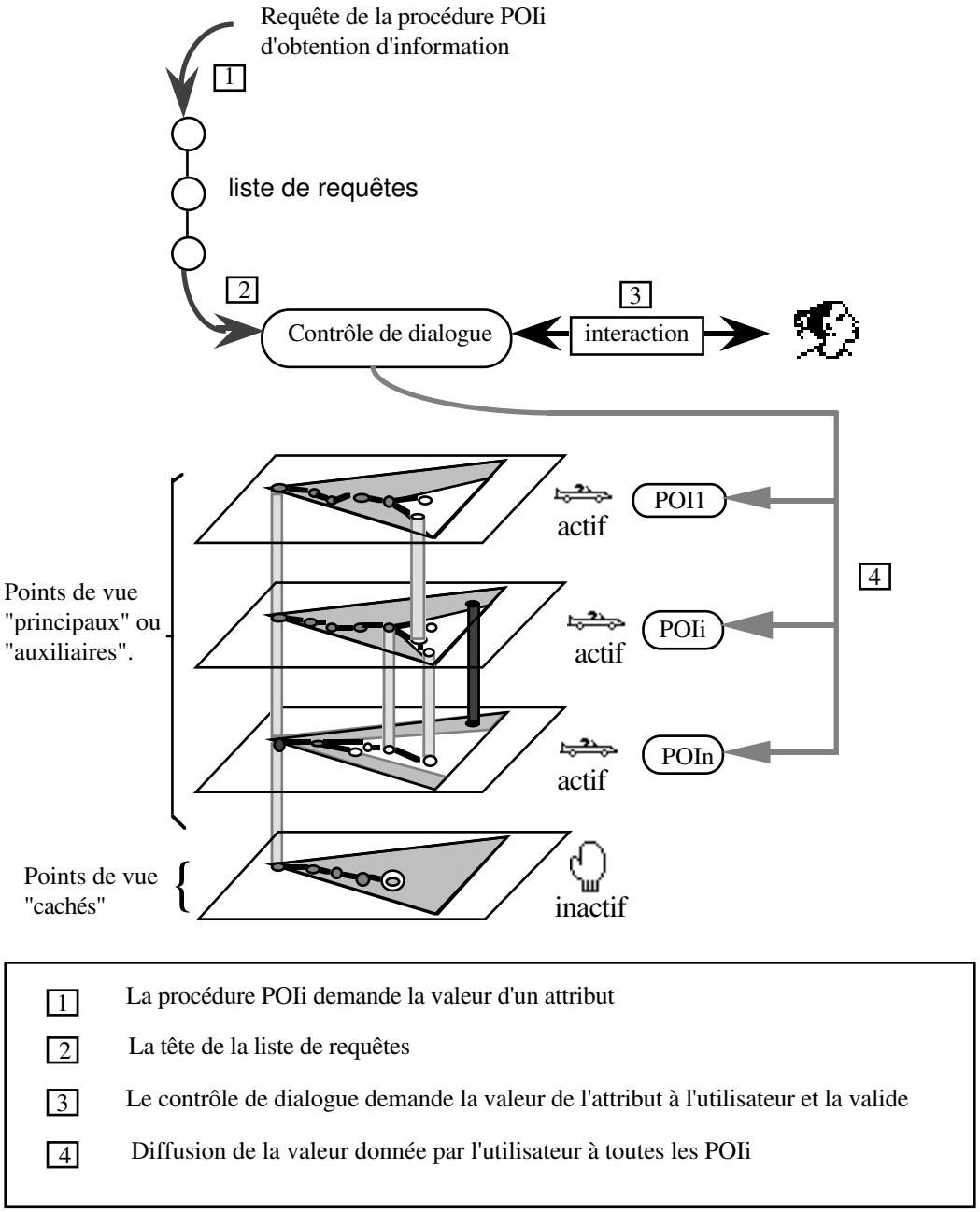


Figure 2.6 : Cycle de demande d'une valeur pour un attribut et diffusion de la valeur donnée par l'utilisateur à toutes les procédures d'obtention d'information.

- Le CD demande la valeur d'un attribut à l'utilisateur en signalant les contraintes. L'utilisateur doit donner une valeur valide. Une fois qu'il donne la valeur, le CD fait une diffusion de cette valeur à tous les POI (figure 2.6).
- Si une POIi a toutes les valeurs pour tous les attributs définis ou affinés dans la classe actuelle C_a, elle s'achève. A ce moment la procédure d'appariement peut choisir entre les sous-classes de C_a, la "meilleure" afin de continuer la classification, c'est-à-dire essayer de faire descendre l'objet le plus bas possible. Si la classe C_a n'a aucune sous-classe, elle devient donc une classe "sûre terminale", et le point de vue devient "inactif".

2.2.2.2. Appariement

Le but de la procédure d'appariement est de trouver une classe "sûre" parmi les sous-classes de la classe actuelle C_a dans un point de vue PVi (l'algorithme détaillé se trouve dans les annexes). Autrement dit, la procédure essaie de trouver une sous-classe de C_a pour laquelle l'objet satisfait toutes les contraintes afin de descendre le plus bas possible l'objet. Pour cela, le système valide toutes les valeurs déjà connues de l'objet, afin d'éliminer les classes qui ne les valident pas. Ces sous-classes deviennent donc "impossibles".
Puis la procédure commence à poser des questions à l'utilisateur, à travers le contrôle de dialogue, pour les attributs dont elle a besoin, les valeurs d'attributs inconnus jusqu'à présent, afin d'inférer les marques des sous-classes. Au fur et à mesure que la procédure reçoit les valeurs pour les différents attributs qu'elle a demandée afin de trouver la "meilleure" sous-classe de C_a, elle fait la validation de la valeur par rapport aux contraintes de définitions de l'attribut dans les différentes sous-classes. Lorsque cette validation donne "impossible", la sous-classe correspondante devient aussi "impossible". Par contre, si la validation donne "sûre" la sous-classe reste toujours "possible".
Le cycle de demande de valeurs de la procédure d'appariement termine lorsque :

- toutes les sous-classes de la classe actuelle C_a sont marquées comme "impossibles". Dans ce cas, la classe actuelle devient "sûre terminale" et le point de vue PVi "inactif".
- il ne reste qu'une sous-classe SC_a comme "possible" et toutes les autres classes ont été marquées comme "impossibles". Dans ce cas, SC_a devient la nouvelle classe actuelle du point de vue PVi (pour le cas d'une taxonomie).

Nous venons d'exprimer l'exécution d'un processus de la procédure d'appariement dont le contexte est le point de vue PVi. Or, l'exécution en parallèle de plusieurs processus de la procédure d'appariement n'entraîne une interaction entre eux qu'au niveau de la demande des valeurs pour les attributs manquants pour le choix de la "meilleure" sous-classe de la classe actuelle. Cette interaction se déroule de la même façon que pour la procédure d'obtention d'information. Il n'y a qu'une différence : l'utilisateur peut donner une valeur qui ne valide pas les contraintes, et ainsi éliminer certaines sous-classes.
Il faut rappeler que la procédure d'appariement marque comme "impossible" ou "sûre" les sous-classes de la classe actuelle du PVi, mais c'est la procédure de propagation de marques qui met à jour les marques des autres classes :

- du point de vue PVi : tous les sous-arbres d'une classe marquée comme "impossible" deviennent aussi "impossibles".
- des autres points de vue à travers des passerelles.

2.2.2.3. Propagation de marques

Les procédures d'appariement produisent des changements sur les états des classes et des passerelles lors de leur exécution. L'état d'une passerelle est défini à partir des classes : s'il y a des changements des états de ces classes alors ces changements entraînent les changements des états des passerelles. Ces changements peuvent générer des incohérences au niveau des points de vue. Par exemple une classe C/PVi devenue "impossible" et qui est une passerelle bi-directionnelle entre C/PVi et D/PVj $i \neq j$ entraîne que la classe D/PVj devient également "impossible". Le but de la procédure de propagation de marques est donc la mise-à-jour des états des classes dans tous les points de vue. Autrement dit, la procédure de propagation de marques rétablit la cohérence en marquant "impossible" les sous-classes des classes "impossibles", et en propageant quand il faut les marques "sûre" et "impossible" par les passerelles. Le premier type de propagation s'appelle "propagation interne" (figure 2.7) et le dernier s'appelle "propagation externe" (figure 2.8).

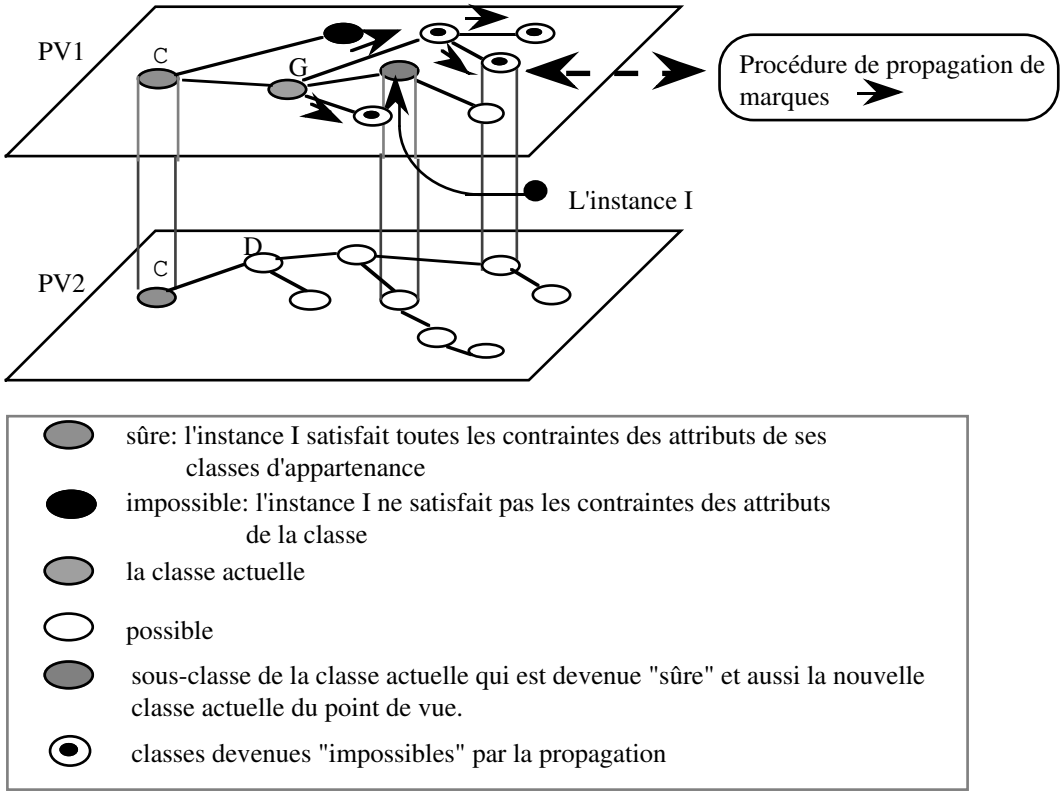


Figure 2.7 : Une des sous-classes de la classe actuelle G est devenue “sûre”, donc toutes ses classes sœurs deviennent “impossibles” de même que les sous-classes de ces classes.

Cette propagation des changements par les passerelles dépend du type de passerelle, bi-directionnelle ou orientée, et du type de la marque sûre ou impossible :
- pour les passerelles bi-directionnelles (définies entre C/PVi et D/PVj i≠j), les deux marques “impossible” et “sûre” doivent être propagées. Si la marque propagée vers la classe D/PVj est “impossible”, toutes leurs sous-classes deviennent aussi “impossible”. Par contre, si la marque propagée est “sûre”, il faut remonter vers les sur-classes de D/PVj à travers des liens “sorte-de”, en marquant toutes ces sur-classes comme “sûres”. Lorsqu’une sur-classe SC est marquée comme “sûre”, les classes sœurs de cette classe SC sont marquées comme “impossibles”. Dans le cas d’une classe sœur marquée comme “impossible”, toutes ses classes descendantes deviennent également “impossibles” (figure 2.8).

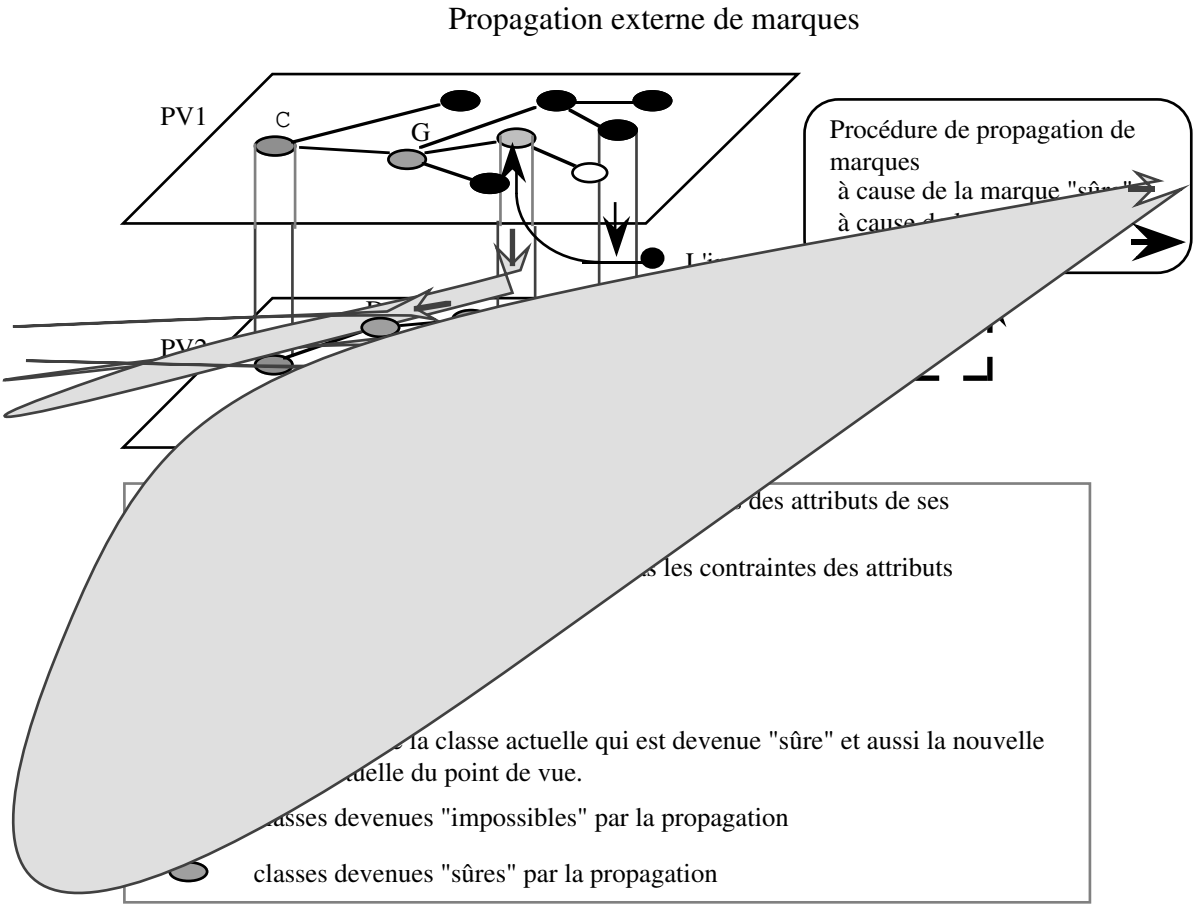


Figure 2.8 : Au fur et à mesure que la propagation interne dans un point de vue marque les différentes classes, le système vérifie si elles sont des passerelles. Si elles sont des passerelles le système déclenche une procédure de propagation dans les autres points de vue de ces passerelles.

Le fait de savoir qu'un objet appartient à une classe C dans un point de vue PVi et que cette classe est une passerelle vers un autre point de vue PVj, permet de faire des inférences sur la position de l'objet dans PVj. Autrement dit, le système infère la classe d'appartenance de l'objet dans le point de vue PVj. C'est cette possibilité d'inférence qui rend plus puissant le modèle TROPES et la classification multi-points de vue.
- pour les passerelles orientées (définies de C/PVi vers D/PVj i≠j), la propagation dépend du type de la marque et de la "direction" de la passerelle. L'origine de la passerelle orientée, la classe C/PVi, peut être vue comme un sous-ensemble de la classe D/PVj. Le fait que l'objet appartienne à l'une de deux classes entraîne :

- si l'objet I appartient à C, il doit aussi appartenir à D. Dans ce cas, le système propage de C vers D, par la direction de la passerelle, la marque "sûre".
- si la classe C est "impossible" alors l'objet n'appartient pas à C, et le système ne peut rien déduire sur l'appartenance de l'objet à la classe D, car C n'est qu'un sous-ensemble de D, et l'objet peut appartenir ou non à D. Pour la propagation, dans ce cas, la classe C étant "impossible", le système ne la propage pas par la passerelle, laquelle devient "impossible".
- si la classe D est "impossible" alors l'objet n'appartient pas à la classe C, car C est toujours un sous-ensemble de D. La procédure de propagation propage la marque "impossible" par la passerelle dans le sens contraire de sa définition : de D vers C.
- enfin si l'objet appartient à la classe D, le système ne peut rien déduire sur l'appartenance de l'objet à la classe C.

Lors de la propagation d'une marque par une passerelle, le système déclenche la procédure de propagation au niveau du point de vue, "propagation interne". Dans la table suivante se trouve le résumé de la propagation des marques (table 2.1).

état classe C	passerelle bi-direction. C <---> D	passerelle orientée C ---> D	propagation	état classe D
sûre (1)	---->		oui	sûre (2)
sûre (2)	<---		oui	sûre (1)
sûre (1)		--->	oui	sûre (1)
impossible (1)	---->		oui	impossible (2)
impossible (2)	<---		oui	impossible (1)
impossible		<---	oui	impossible (1)
impossible (1)		--->	non	----------
----------		<---	non	sûre (1)

le la propagation des marques par les passerelles selon leur type et le type de la marque. La propagation et le numéro entre parenthèses l'ordre de la propagation.

relles dépend de l'état des classes qui définissent la passerelle, d'une de ces classes entraîne le changement de l'état de la t les passerelles d'un point de vue ont été mises à jour, et nchée, celle-ci considère le type de point de vue et s'il point de vue "actif" ou le changer en "inactif". Si cune autre passerelle "active", le point de vue

parallèle de plusieurs processus de la ntraîne aucune incohérence dans le définir la zone d'influence d'un

- une propagation interne e". Ces sous-classes bre est la "zone

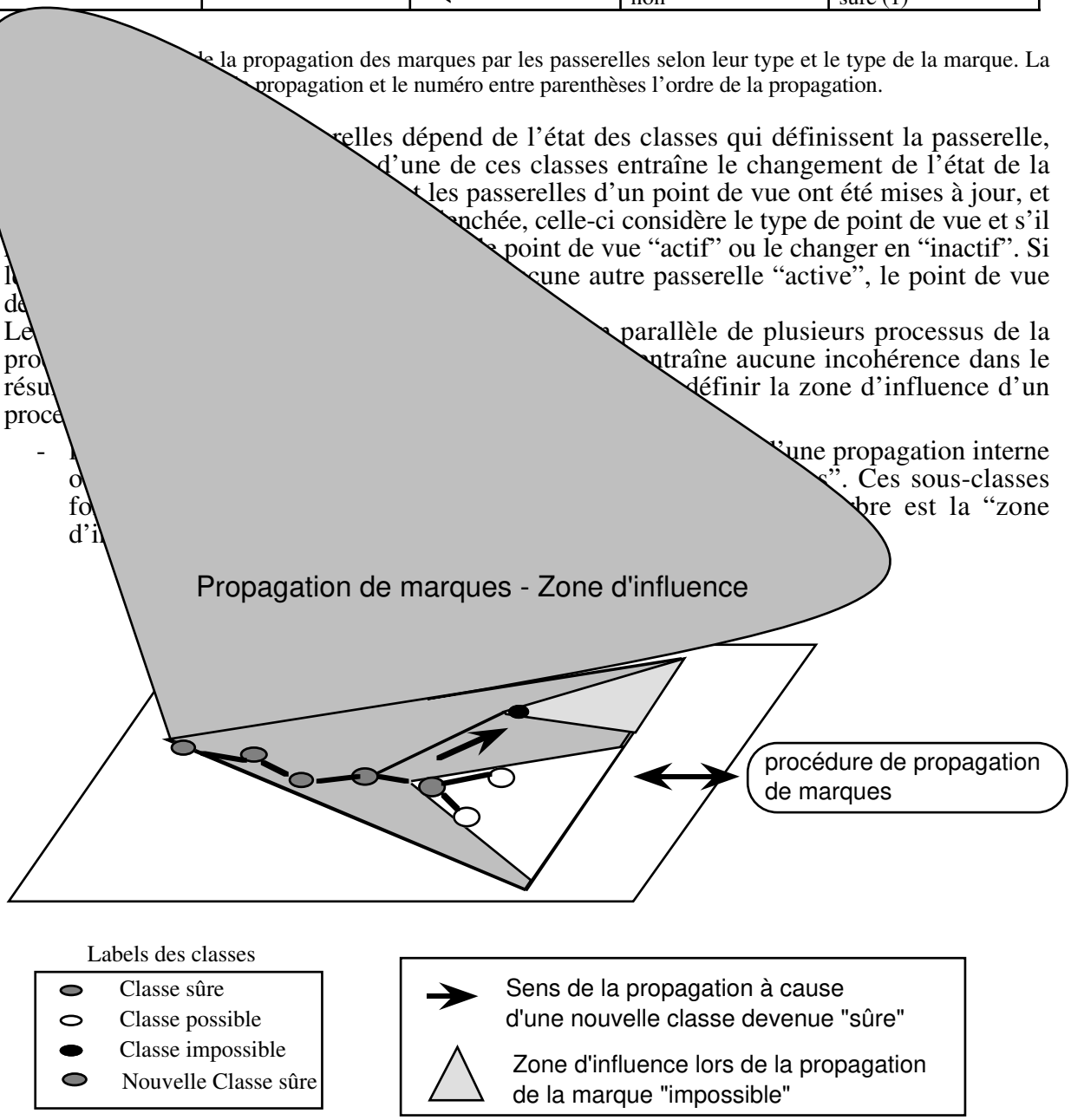


Figure 2.9 : Zone d'influence pour la propagation de la marque "impossible"

- s'il existe une passerelle entre C/PVi et D/PVj ou de C/PVi vers D/PVj, et que la classe C devient "sûre" alors la classe D devient également "sûre". Une fois que cette classe D est marquée "sûre" toutes ses sur-classes deviennent aussi "sûres". Cet ensemble de classes est la "zone d'influence" pour la propagation de la marque "sûre".

Pour montrer qu'il est possible d'exécuter en parallèle plusieurs processus de la procédure de propagation, nous décrirons deux cas : soit il n'existe qu'une passerelle entre les points de vue, soit il existe au moins deux passerelles entre les points de vue. Dans le premier cas, cette passerelle est la passerelle entre les classes racines des points de vue. Dans le dernier cas la propagation dans un point de vue PVi peut entraîner le déclenchement des processus de la procédure de propagation dans d'autres points de vue.

- pour le premier cas, l'exécution en parallèle de plusieurs processus de la procédure de propagation n'entraîne aucune incohérence au niveau des marques des classes, étant donné qu'il n'existe pas de communication entre les points de vue, sauf la racine. Donc chaque processus a un contexte complètement indépendant des contextes des autres processus. En effet, étant donné que le contexte est formé de l'ensemble des classes qui font partie du point de vue, et qu'il n'y a aucune passerelle sauf la racine, l'intersection de ces ensembles sans la racine donne l'ensemble vide. Donc, à la fin, l'état des classes qui appartiennent aux différents points de vue est cohérent. Dans ce cas, toutes les propagations sont internes, et une propagation externe n'est pas possible.
- pour le deuxième cas il faut montrer qu'il est possible que plusieurs processus de la procédure de propagation s'exécutent simultanément sur le même point de vue. Il y a des processus qui sont déclenchés à cause des propagations externes et d'autres à cause des propagations internes. La démonstration sera faite en deux parties : quand les zones d'influence des différentes processus sont indépendantes et quand les zones d'influence se superposent.
    - chaque processus de la procédure de propagation, dont le contexte est le même point de vue PVi, a une zone d'influence différente de celle des autres processus. Ces zones sont alors des sous-arbres indépendants de la hiérarchie de classes qui n'ont aucune classe en commun; la hiérarchie de classes étant un arbre, deux sous-arbres indépendants sont nécessairement disjoints et deux procédures ne peuvent pas marquer les mêmes classes avec des marques différentes. Donc, après l'exécution de toutes les procédures de propagation sur le même PVi, celui-ci reste cohérent (figure 2.10).

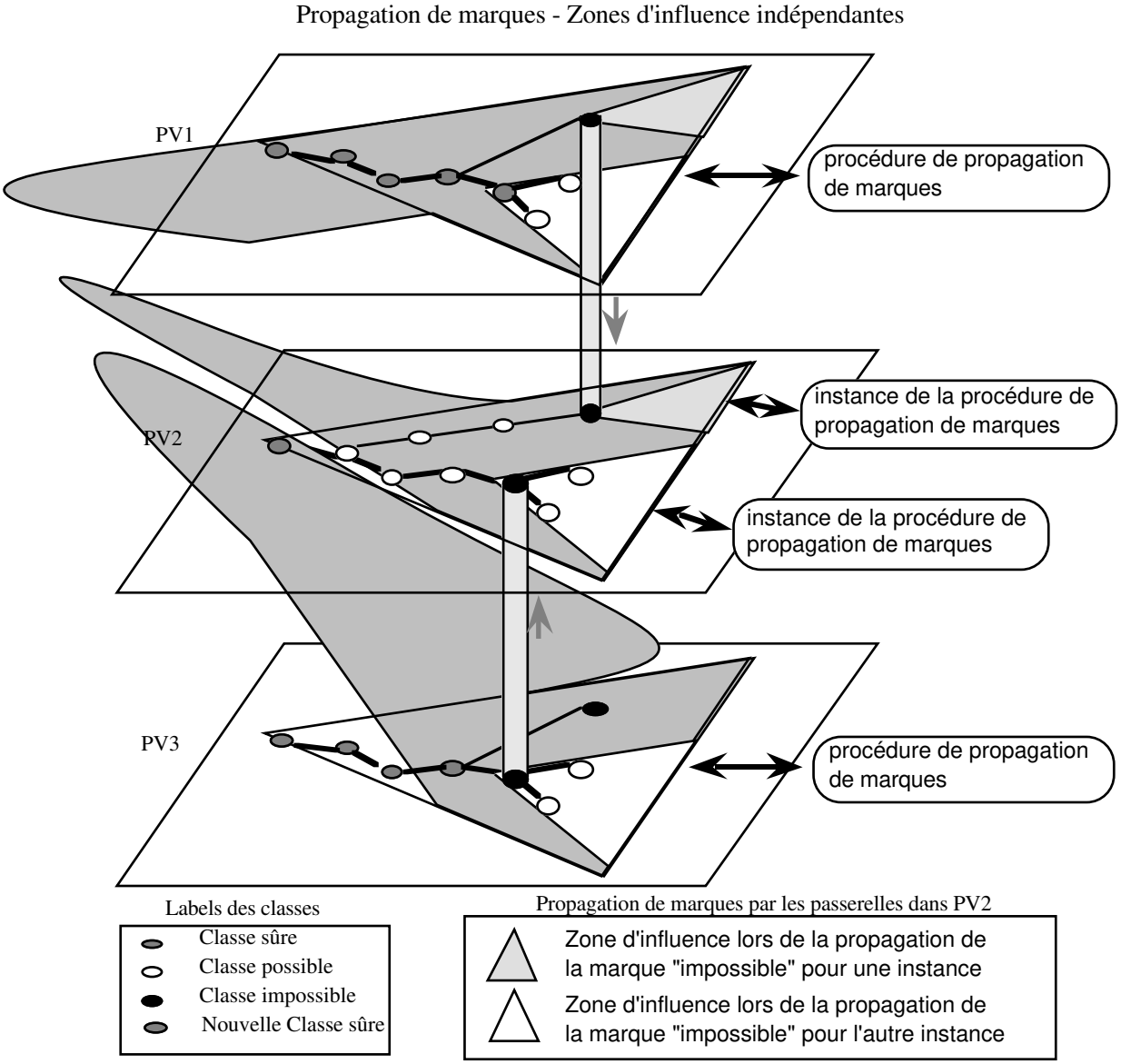


Figure 2.10 : Exécutions simultanées de deux instances de la procédure de propagation sur le point de vue PV2 avec des zones d'influence indépendantes.

- pour montrer que l'exécution simultanée de plusieurs processus de la procédure de propagation, dont les zones d'influence se superposent, n'entraîne aucune incohérence, il faut partir du fait que le concept défini pour l'utilisateur est cohérent et que la hiérarchie des classes est un arbre. Donc, s'il existe une superposition entre deux zones d'influence Z1 et Z2 de deux processus P1 et P2 respectivement alors, ou bien Z1 est un sous-arbre de Z2 ou bien Z2 est un sous-arbre de Z1.
  Une superposition entre Z1 et Z2 veut dire qu'il existe au moins une classe commune entre les deux zones d'influence, et étant donné que chaque zone d'influence est un arbre, il peut y avoir trois cas :
  a) Pour le cas où Z1 = Z2 (Z1 est le même sous-arbre que Z2) et en partant de la cohérence du modèle, la propagation des marques faite par les processus P1 et P2 sur Z1 et Z2 respectivement est exactement la même. Il suffit donc qu'un seul des deux processus P1 ou P2 s'exécute pour mettre à jour le sous-arbre avec les marques respectives pour leurs classes. Enfin, dans ce cas la "faisabilité" du deuxième processus en relation à l'état global du point de vue n'est plus valable, étant donné que leur tâche

a été déjà réalisée par un autre processus, et il n'est pas nécessaire de le déclencher. Donc la cohérence des points de vue est assurée.

b) Pour le cas où Z2 est un sous-arbre de Z1, la démonstration est aussi simple. Si les deux procédures P1 et P2, dont les contextes sont Z1 et Z2 respectivement, sont déclenchées simultanément ou P2 avant P1, et étant donné que Z2 est un sous-arbre de Z1, P2 doit marquer avant P1 la classe racine C2 de Z2 (par exemple, si les deux procédures s'exécutent en parallèle dans un environnent multi-processus, dans lequel par exemple chacune s'exécute un instant, et après l'autre, etc. P1 peut finir avant P2 ou vice-versa), et lorsque P1 arrive à C2, elle sera déjà marquée, donc la "faisabilité" de P1 dans le sous-arbre Z2 n'est plus valable et P1 continue la propagation dans le reste de Z1. Il faut noter que la marque de la classe C2 doit être la même pour P1 et P2 grâce à la cohérence du modèle.

D'autre part, si P1 a déjà marqué la racine de Z2 la "faisabilité" du processus P2 n'est plus valable car le sous-arbre Z2 est déjà marqué.

c) Le dernier cas (Z1 est un sous-arbre de Z2) est le symétrique du cas b.

Comme nous l'avons montré ci-dessus, il est possible d'exécuter en parallèle plusieurs processus de la procédure de propagation sur le même point de vue. Pour le cas où les zones d'influence de ces processus se superposent, alors il existe une coopération entre les différents processus pour la propagation de marques dans les points de vue.

2.2.2.4. Mise à jour d'information

Cette procédure est la dernière du cycle principal de l'algorithme de classification d'un objet dans un concept. La descente de l'objet dans un point de vue fait changer la classe actuelle ainsi que l'ensemble des contraintes pour les différents attributs. Pour rétablir complètement la cohérence au niveau de chacun des points de vue modifiées, il faut mettre à jour les contraintes et changer la classe actuelle pour la nouvelle sous-classe "sûre".

L'exécution en parallèle de l'algorithme de classification, où plusieurs processus s'exécutent simultanément sur des contextes différents, entraîne la descente en parallèle dans plusieurs points de vue en même temps.

La plupart de la mise à jour de l'information globale est déjà faite par les procédures d'obtention d'information et d'appariement, car au fur et à mesure qu'elles ont demandé des valeurs pour les différents attributs, elles ont aussi donné leurs contraintes à la procédure de contrôle de dialogue. Donc, il ne reste qu'à mettre à jour les classes d'appartenance de l'objet dans les différents points de vue. Pour cela, il suffit, après avoir fini la procédure d'appariement, d'envoyer au contrôle de dialogue le nom de la nouvelle classe actuelle si celle-ci a changé. D'autre part, une fois que toutes les procédures de propagation de marques sont finies sur un point de vue, et si celui-ci devient "inactif", il faut aussi envoyer au contrôle de dialogue le nom du point de vue qui est devenu "inactif". Il ne reste qu'à mettre à jour l'information locale et cette mise à jour au niveau local n'entraîne pas d'incohérence, car pour chaque point de vue il existe une structure de données indépendante.

En résumé, nous avons montré qu'il est possible de faire la classification en parallèle d'un objet dans un modèle multi-points de vue avec des passerelles, comme c'est le cas de TROPES.

2.2.3. Exécution en parallèle d'instances de toutes les procédures

Dans la section précédente, nous avons montré qu'il est possible d'exécuter plusieurs processus d'une même sous-procédure de l'algorithme de classification simultanément et où le contexte de chacun de ces processus est un point de vue. Dans cette section, nous allons montrer qu'il est possible aussi de paralléliser les différentes parties de l'algorithme proprement dit : exécuter en parallèle des processus de la procédure d'obtention d'information avec des processus de la procédure d'appariement, etc.

Pour cela nous partons de l'exécution en parallèle de plusieurs processus de la procédure d'obtention d'information.

Par chaque processus dont le contexte est le point de vue PVi, le système obtient de l'utilisateur l'information manquante sur la classe actuelle Ca du point de vue PVi, et tous les autres processus reçoivent cette même information grâce à la diffusion des valeurs faite par le contrôle de dialogue (CD). La procédure d'appariement fait une comparaison entre les valeurs des attributs de l'objet et les contraintes des différentes sous-classes de Ca. Il est possible que les processus de ces deux procédures s'exécutent en parallèle, l'un en travaillant sur la classe

actuelle et l'autre sur les sous-classes de la classe actuelle. De cette façon, une fois la procédure d'obtention d'information achevée, la procédure d'appariement a déjà fait la comparaison entre toutes les valeurs connues de l'objet et les contraintes des sous-classes de Ca, et il ne lui reste qu'à demander à l'utilisateur, à travers le CD, les valeurs dont elle a besoin pour choisir une sous-classe comme la nouvelle classe actuelle du PVi. Comme résultat de ces comparaisons, l'appariement marque les sous-classes révisées comme "impossible", sûre" ou "possible". Néanmoins, il n'y a qu'une sous-classe qui peut devenir "sûre". Dans le cas où aucune classe ne devient sure, la classe actuelle Ca devient une classe "sûre terminale".
Lorsqu'une sous-classe est marquée comme "impossible" pour la procédure d'appariement, le sous-arbre SA correspondant à cette sous-classe devient également "impossible". Un processus de la procédure de propagation de marques est donc déclenché afin de marquer toutes les classes et les passerelles de cet arbre SA comme "impossibles". Dans ce cas la propagation de marques est interne. Ce processus s'exécute en parallèle avec des processus de la procédure d'appariement et de la procédure d'obtention d'information.
Il existe également la possibilité qu'un processus de la propagation de marques soit déclenché dans un point de vue PVi à partir d'une classe C comme conséquence de la propagation dans un autre point de vue PVj. Ce processus peut propager la marque "sûre" ou "impossible" et sa zone d'influence peut se superposer ou non avec la classe actuelle Ca ou les sous-classes de Ca du PVi :

- si la marque propagée est "impossible" et la passerelle est une des sous-classes de Ca, alors cette sous-classe n'est plus considérée par aucun autre processus, y compris le processus d'appariement.
- si la marque propagée est "impossible" et la passerelle n'est pas une sous-classe de la classe actuelle du PVi, alors le processus de propagation de marques s'exécute sans interaction avec aucun processus d'appariement. Il peut exister d'autres processus de propagation de marques et leurs zones d'influence peuvent être indépendantes ou pas. Cela a été déjà considéré dans la section précédente.
- si la marque propagée est "sûre", alors le processus de propagation va marquer une classe parmi les sous-classes de la classe actuelle Ca du PVi comme "sûre". Dans le cas où il existe un processus d'appariement, il s'arrête étant donné que le système a déjà trouvé la "meilleure" sous-classe de Ca.

Il nous reste à dire que la procédure de "mise-à-jour" d'information peut s'exécuter en parallèle (le contexte pour la procédure mise-à-jour est le même que pour la procédure de propagation : le point de vue PVi) avec la procédure de propagation des marques, et une fois que tous les processus de "mise-à-jour" sont finis, le système arrive à un état stable, donc les points de vue sont cohérents. Dans cet état stable :

- le système a choisi une nouvelle classe "sûre" comme actuelle.
- la classe actuelle est devenue "sûre terminale".

Enfin, nous avons montré qu'il est possible d'exécuter en parallèle le processus d'obtention d'information, d'appariement et de propagation de marques dont le contexte est le même point de vue PVi. De la même manière un processus "mise-à-jour" peut s'exécuter en parallèle avec des processus de propagation de marques, et après tous les processus d'obtention d'information et d'appariement (algorithme 2.2a et 2.2b).

h = hauteur maximale de l'arbre de classes (point de vue).

n = nombre maximal de sous-classes de la classe actuelle.

pa = nombre de passerelles actives dans le point de vue actuel.

p = nombre de points de vue du concept.

pour chaque point de vue PV (i=1,...,p)

appeler procédure classification (PVi,l'objet `In`) **en parallèle**

algorithme 2.2a

pour chaque point de vue PV (i=1,...,p) **en parallèle**

/* La classe sûre du point de vue PV_i est C_j^i */

- **appeler** procédure d'obtention d'information (PV_i,C_j^i, In) **en parallèle**
- **faire en parallèle**
  - **pour chaque** sous-classe C_k^i de C_j^i (k=1,...,n)

    appeler procédure d'appariement (PV_i,C_k^i, In) **en parallèle**

    pour chaque attribut $At_{k_j}^i$ de la classe C_k^i **en parallèle**

    pour chaque contrainte $Ct_{k_{j_L}}^i$ de l'attribut $At_{k_j}^i$ **en parallèle**

    évaluer($At_{k_j}^i$, $Ct_{k_{j_L}}^i$, valeur)

    marque = "impossible"
    Pa = ensemble de passerelles du point de vue PV_i
  - **si** une sous-classe C_k^i devient "impossible" **alors**

    appeler propagation de marques(PV_i,C_k^i,marque,Pa) **en parallèle**

    sinon

    si une sous-classe C_k^i devient "sûre" **alors**

    marque = "sûre"

    appeler propagation marques(PV_i,C_k^i,marque,Pa) **en parallèle**

    pour chaque sous-classe sœur C_l^i de C_k^i (l=1,...,n; l≠k) **en parallèle**

    marque = "impossible"

    appeler propagation marques(PV_i,C_l^i,marque,Pa) **en parallèle**

    fin-si

    fin-si
- **appeler** procédure de mise à jour (PV_i,C_j^i, In)

Algorithme 2.2b : Algorithme de classification en parallèle d'un objet dans TROPES

2.3 Conclusions

Dans ce chapitre, nous avons montré que la classification d'un objet dans le modèle TROPES est un problème hautement parallélisable.

L'objet peut être classifié en parallèle dans tous les points de vue; dans ces points de vue, la procédure de classification entraîne une comparaison entre les classes qui appartiennent au point de vue et un objet particulier (l'objet). Cette comparaison peut être faite en parallèle. La comparaison d'un objet avec une classe entraîne l'évaluation des différentes valeurs par rapport aux contraintes de tous les attributs de la classe. L'évaluation des différents attributs et l'évaluation des valeurs par rapport aux différents contraintes peuvent être faites en parallèle.

3. MODELES PARALLELES DE RESOLUTION DE PROBLEMES EN INTELLIGENCE ARTIFICIELLE

En intelligence artificielle, il existe plusieurs modèles qui utilisent le parallélisme pour la résolution de problèmes[5]; on distingue parmi ceux-ci : le "tableau noir", le modèle "acteur" et le modèle "multi-agents".

Le modèle "acteur" a été proposé par Hewitt [HEW 77]. Les acteurs sont des objets actifs et concurrents, qui communiquent entre eux par échange de messages. Chaque acteur a une boîte aux lettres dans laquelle il peut recevoir un nombre fini de messages. Il a aussi une adresse unique, qui est son identification, un ensemble d'accointances (les acteurs connus par lui) et un scénario qui décrit son comportement. Comme réponse à un message, un acteur peut exécuter un nombre fini d'actions. Dans ce modèle le paradigme utilisé pour la résolution d'un problème est la *délégation*.

Le modèle du "tableau noir" est une architecture de système à base de connaissances, qui se compose d'un ensemble de sources de connaissances, d'un contrôleur (superviseur) qui surveille le bon déroulement d'une résolution et d'une zone de données partagées (appelée le "tableau noir"). Les sources de connaissances partagent l'information à travers le "tableau noir". Les sources de connaissances coopèrent entre elles afin de résoudre le problème. La communication entre les sources se fait par le "tableau noir" où chacune peut lire les données d'un sous-problème du problème général, et peut écrire les solutions trouvées lors de la résolution d'un des sous-problèmes indiqués sur le tableau. Ce processus est surveillé par le contrôle jusqu'à qu'une solution soit trouvée ou réfutée. Dans ce modèle le paradigme utilisé pour la résolution d'un problème est la *coopération*. Le premier système de "tableau noir" a été développé par [ERM 75]. Bien que les paradigmes utilisés pour les deux modèles précédents soient différents, ils ne sont ni contradictoires, ni incompatibles. Au contraire, il est possible de construire un système en utilisant les deux modèles. Par exemple dans TrActors [HAG 92], le contrôleur d'un système de "tableau noir" a été développé en utilisant le modèle "acteur".

Le modèle "multi-agents" est un système qui utilise la *négociation* pour la résolution de problèmes [KON 80]. Les agents communiquent entre eux : des actions, la division de tâches, la façon de partager une ressource. Cette communication est faite afin de trouver une solution au problème (mécanisme explicite de communication). Les agents peuvent réquisitionner, répondre, conseiller, ou permettre l'exécution de certaines actions. Du fait de leur travail simultané, chacun des agents voit son "monde" changer sans qu'il puisse le prédire. Leurs plans doivent donc être sensible aux changements produits par les autres intervenants. Les agents doivent aussi réduire les interactions mutuelles et établir un comportement coopératif entre eux afin d'atteindre celui d'une équipe. Le problème est alors de savoir quels sont les éléments à prendre en compte pour permettre à un groupe d'individus de formuler un plan, constitué d'un ensemble de parties élaborées indépendamment. Pour la mise en œuvre d'un système multi-agents, les problèmes liés à la communication entre agents sont cruciaux. Néanmoins, jusqu'à présent, il n'existe aucune théorie formelle sur la sémantique de cette communication [MUN 91][KRA 91]. Pour cette raison, nous n'avons pas choisi ce modèle pour résoudre le problème de classification parallèle dans TROPES.

Dans le reste du chapitre nous présentons l'architecture de "tableau noir" et le modèle "acteur". Pour chacun de ces deux types de système, nous présentons notre propre mise en œuvre, développée à l'Université des Andes, et l'équipe HIDRA au sein du Laboratoire d'informatique (Bogotá-Colombie).

3.1. Le "tableau noir"

Le modèle du "tableau noir" est un modèle de résolution de problèmes, qui décrit l'organisation de la connaissance, des données et le fonctionnement de la résolution de problèmes dans l'organisation générale. Le modèle de "tableau noir" permet d'appréhender la résolution d'un problème au travers d'un processus incrémental et opportuniste de construction de la solution à partir de l'espace solution selon différentes hiérarchies correspondant à différents niveaux d'analyse du problème" [BOI 90]. Le modèle de "tableau noir" a été utilisé dans de nombreuses applications, en utilisant différentes combinaisons de représentation de la connaissance, des

---

[5] Un modèle de résolution de problèmes est un schéma pour l'organisation des étapes du raisonnement et du domaine de connaissance afin de construire une solution à un problème [NII 79].

schémas de raisonnement et des mécanismes de contrôle. C'est dans ce cadre que nous allons utiliser le modèle. Pour une description plus détaillée du modèle, voir [ENG 88] [NII 86b][QUI 92a].

3.1.1. Historique

Les premières personnes ayant parlé de la notion de "tableau noir" en Intelligence Artificielle sont Newell et Simon. Le premier [NEW 72] s'intéresse au problème du stockage de la connaissance et à son accès global. Il présente ainsi la notion de mémoire de travail sous la forme d'un "tableau noir" sur lequel viennent s'inscrire les éléments de solution. Simon [SIM 77] par contre présente sa conception de la résolution de problème comme le processus effectué par un ensemble de démons, chacun apportant sa contribution à la résolution.
Entre 1971 et 1976, ces deux notions ont été réunies et développées pour donner au "tableau noir" sa définition actuelle. Cette évolution s'est produite à travers la construction d'un système de compréhension de la parole: Hearsay-II [ERM 75].
Le modèle de "tableau noir" constitue un des premiers systèmes introduisant la notion de coopération entre systèmes à bases de connaissances [GAS 88] [BOI 90]. Son utilisation a été reprise et développée dans cette direction avec des systèmes répartis sur plusieurs sites et a donné lieu à la construction de nombreuses architectures [COR 83]. Ainsi ont été introduits les problèmes de communication de connaissances, de cohérence de la résolution, de contrôle de la résolution [HAY 86], et de comportement coopératif entre plusieurs agents [COR 83].
Actuellement les systèmes de "tableau noir" s'orientent vers des architectures parallèles ou systèmes répartis mettant ainsi en valeur l'indépendance des spécialistes [FEN 77]. Pour plus de détails sur leur histoire, voir [ZIE 90] [ENG 88].

3.1.2. Métaphore

Nous allons présenter le comportement du modèle avec un exemple imagé introduit par Nii [NII 86a].
Cet exemple considère plusieurs personnes placées autour d'une table pour réaliser un puzzle. Chacune des personnes possède un certain nombre de pièces du puzzle.
La résolution est initialisée par des volontaires qui viennent placer leurs meilleures pièces sur la table. Chaque membre du groupe regarde alors si les pièces qu'il possède peuvent compléter la solution. Ceux qui ont des pièces appropriées les déposent, et ainsi améliorent la solution en cours. Ceci permettra à d'autres personnes de placer leurs pièces à leur tour. Il faut souligner:

- que chaque personne s'active elle-même en regardant les changements intervenus sur le puzzle.
- qu'il n'y a aucune communication entre les personnes.
- qu'elles ne savent pas a priori quelles pièces possèdent les autres, c'est-à-dire qu'elles ignorent la connaissance que les autres ont du problème.
- qu'il n'y a pas d'ordre établi a priori pour aller poser les pièces.
- que la solution est construite de manière incrémentale et opportuniste (§ 3.1.3.3) selon l'état de la résolution en cours.

Si maintenant le puzzle n'est assemblé que par une seule personne à la fois, un moniteur est nécessaire pour choisir cette personne. Pour établir son choix, le moniteur utilise différents critères: la personne qui s'est manifestée la première, la personne qui a le plus de pièces à placer, etc. Ces critères constituent des stratégies de résolution que le moniteur choisit à l'initialisation du système ou qu'il développe en cours d'exécution.
Conceptuellement, il existe un parallélisme important dans ce modèle. Les personnes peuvent participer de manière parallèle et asynchrone. Nous montrerons dans la suite du chapitre que cela permet d'envisager une implantation matérielle du modèle sur des systèmes multiprocesseurs.

3.1.3. Le modèle des systèmes de "tableau noir"

Dans cette section, nous allons exposer les différents composants du modèle de façon générale, et dans le chapitre suivant, nous ferons la correspondance entre le modèle de "tableau noir" et la classification dans le modèle TROPES.
Le modèle de "tableau noir" est constitué de trois composants indépendants, qui sont les sources de connaissances, que nous appelerons SC, la structure de données globale (le "tableau") et le contrôle (figure 3.1).

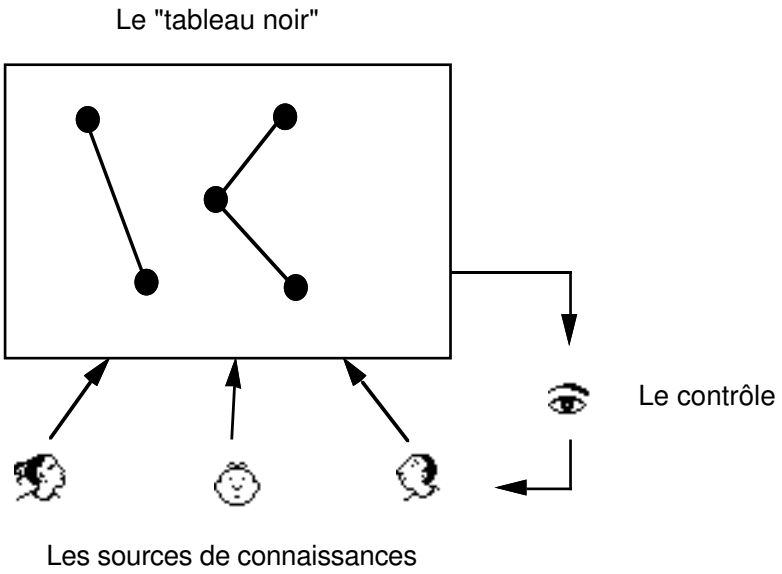


Figure 3.1 : Chaque spécialiste apporte ses connaissances afin de résoudre le problème. Ces spécialistes sont indépendants.

3.1.3.1. Les sources de connaissances

Le domaine de la connaissance nécessaire pour résoudre un problème est divisé en sources de connaissances, qui sont séparées et indépendantes. Celles-ci représentent les différents spécialistes, qui coopèrent pour arriver de façon incrémentale à la solution du problème, chaque source apportant ses connaissances spécialisées sur le problème (figure 3.2). Les sources de connaissances sont les seules qui peuvent modifier l'état du "tableau noir". Toutes les modifications conduisant vers la solution sont visibles et explicites.

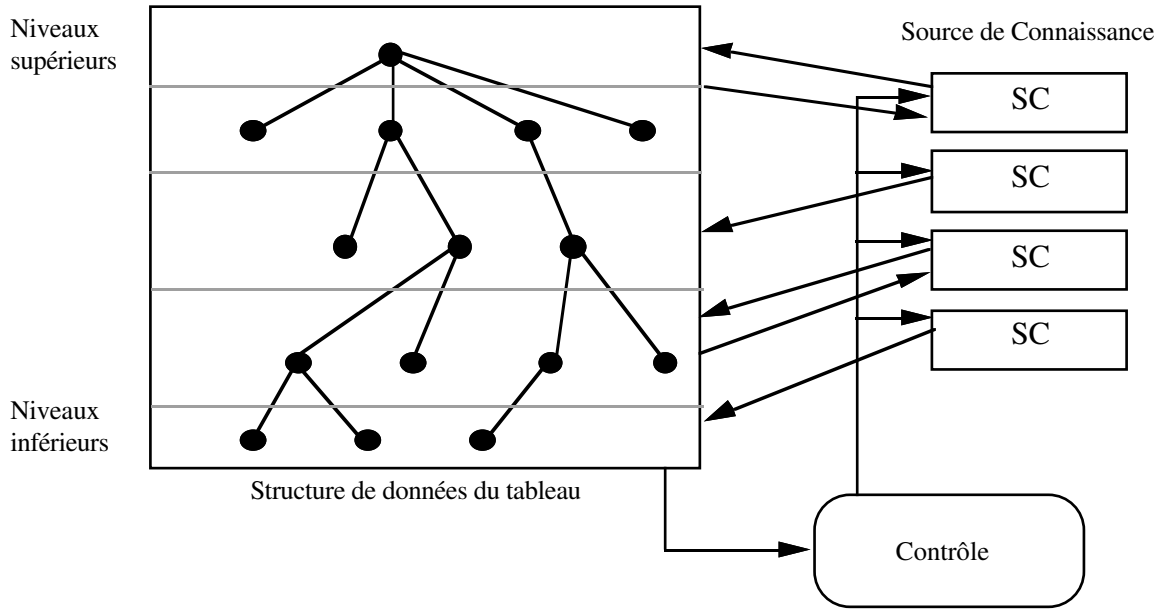


Figure 3.2 : Les données sont organisées hiérarchiquement, les sources de connaissances sont logiquement indépendantes et elles seules peuvent modifier le "tableau noir". A partir des derniers changements sur le "tableau noir", le contrôle sélectionne et exécute la prochaine source de connaissances.

Chaque source de connaissances doit savoir les conditions sur lesquelles elle peut contribuer à la solution, et les préconditions nécessaires pour son déclenchement. Donc, une source de connaissances est composée de deux parties: la précondition pour son déclenchement et le corps qui constitue la partie action. Ces connaissances sont représentées de façon déclarative comme un ensemble de procédures, un ensemble de règles ou d'assertions logiques. Néanmoins, la

plupart des sources de connaissances qui ont été développées utilisent des procédures ou des ensembles de règles. Hearsay-II, le premier modèle de “tableau noir” qui a été développé utilise des procédures [ERM 80].
Les sources de connaissances présentent les caractéristiques suivantes [BOI 90]:

- elles sont indépendantes et séparées.
- elles sont les seuls éléments intervenant sur le “tableau noir”, en le modifiant ou en effectuant des entrées-sorties.
- elles permettent la structuration de la connaissance à travers la définition de leurs niveaux d’intervention sur le “tableau noir”.
- elles coopèrent pour le partage d’information.

Pour résumer, les sources de connaissances renferment la connaissance nécessaire à la résolution du problème et elles sont indépendantes. Appartenant à un domaine d’expertise, une source de connaissances correspond à une compétence précise.

3.1.3.2. Structure de données

Le “tableau noir” est une structure de données globale. Les différentes sources de connaissances produisent des changements de façon incrémentale sur le “tableau noir”, qui permettent d’arriver à la solution du problème. L’interaction entre les sources de connaissances ne se fait qu’à travers le “tableau noir”, car il n’existe pas de communication directe entre les différentes sources.

Le “tableau noir” est composé d’objets qui peuvent être : des données d’entrée, des solutions partielles, différentes possibilités et bien sûr la solution finale. Il peut aussi exister des données de contrôle utilisées par le contrôle. Ces objets et ces propriétés définissent l’espace de solution du problème.

Le “tableau noir” est structuré en une hiérarchie de niveaux de résolution de problèmes, qui correspondent à différents degrés d’analyse du problème [LEN 91]. En général les niveaux supérieurs (par rapport à sa position dans la hiérarchie) contiennent des objets plus abstraits que les niveaux inférieurs (par rapport à sa position dans la hiérarchie). Donc, au fur et à mesure que se fait une descente dans la hiérarchie, la connaissance sur la solution du problème augmente, et s’approche de la solution du problème (figure 3.2).

Les objets sont reliés par des liens sémantiques. Ces relations peuvent être entre objets de différents niveaux ou objets de même niveau.

Enfin, dans certains cas, l’espace solution peut être divisé en plusieurs hiérarchies formant des sous-espaces ou panneaux. Chaque sous-espace a ses propres propriétés. Les différentes opérations faites par les sources de connaissances sont réalisées de façon indépendante des autres sous-espaces. Lors de la production d’un événement, celui-ci peut être considéré comme local aux sous-espaces. Donc, dans ce cas, le “tableau noir” peut être vu comme un ensemble de panneaux.

Pour résumer, le “tableau noir” est une base de données globale contenant l’état courant de résolution du problème. Cette base de données est divisée en niveaux hiérarchiques. Les sources de connaissances déclenchées par l’apparition d’un événement dans le “tableau noir” créent, modifient ou détruisent une hypothèse et participent à la construction incrémentale de la solution. Elles répondent aux changements du “tableau noir”. Les communications et les interactions entre les sources de connaissances passent exclusivement par le “tableau noir”.

3.1.3.3. Le contrôle

Au fur et à mesure que les différentes sources de connaissances modifient le “tableau noir”, d’autres sources peuvent être déclenchées comme conséquence de ces changements. Il existe des modules de contrôle, qui se chargent de surveiller ces changements et de décider quelles sont les actions à suivre. Ces actions indiquent sur quoi appliquer le prochain processus: quelle source de connaissance est déclenchée ou quelle structure de données est mise-à-jour.

La solution est construite pas à pas (de façon incrémentale). A chaque pas, les différentes sources peuvent utiliser différentes formes de raisonnement. Il est clair que dans les systèmes de “tableau noir”, le déclenchement des sources de connaissances est fait de façon dynamique et opportuniste (ce qui permet d’exprimer explicitement des heuristiques de résolution. Les heuristiques de contrôle s’adaptent dynamiquement à l’état courant de la solution.), et non de façon fixe.

Le cycle de contrôle des systèmes de “tableau noir” est composé de trois opérations (figure 3.3): détection, sélection et exécution [ENG 88] [BOI 90] [JAG 89].

- *Détection*: lorsqu’une source de connaissances fait une modification sur les objets du “tableau noir”, un mécanisme de contrôle détecte ces modifications (certaines d’entre elles sont la contribution de la source de connaissances à la solution du problème). Le mécanisme de détection est fondé sur les événements. Un événement permet de donner à la résolution une image résumée en termes de modification, destruction, création (ces changements sont faits sur les objets du tableau noir, et en particulier ils peuvent être faits sur des structures de contrôle). Il est possible de raisonner sur une abstraction de la solution sans se perdre dans les détails.
- *Sélection*: une fois que les mécanismes de détection ont relevé un événement, le système sélectionne la prochaine action à exécuter. L’opération de sélection vise à déterminer la prochaine action à exécuter pour faire progresser la résolution. Les sources de connaissances peuvent entrer en compétition pour différentes raisons [BOI 90] :

- elles ont des niveaux d'entrée en commun dans le tableau.
- l'état actuel du tableau présente une situation telle que plusieurs régions de données différentes constituent des îlots de données compétitifs : la modification d'un îlot peut invalider ou modifier la valeur d'un autre îlot.

La phase de sélection permet de réaliser une conduite opportuniste de la résolution, avec plus ou moins de flexibilité dans la définition des stratégies de contrôle. Le choix porte généralement sur les sources de connaissances en utilisant, selon les applications du modèle, différents critères relatifs notamment aux objets concernés par les sources de connaissances et l'action de celles-ci [ROC 84].

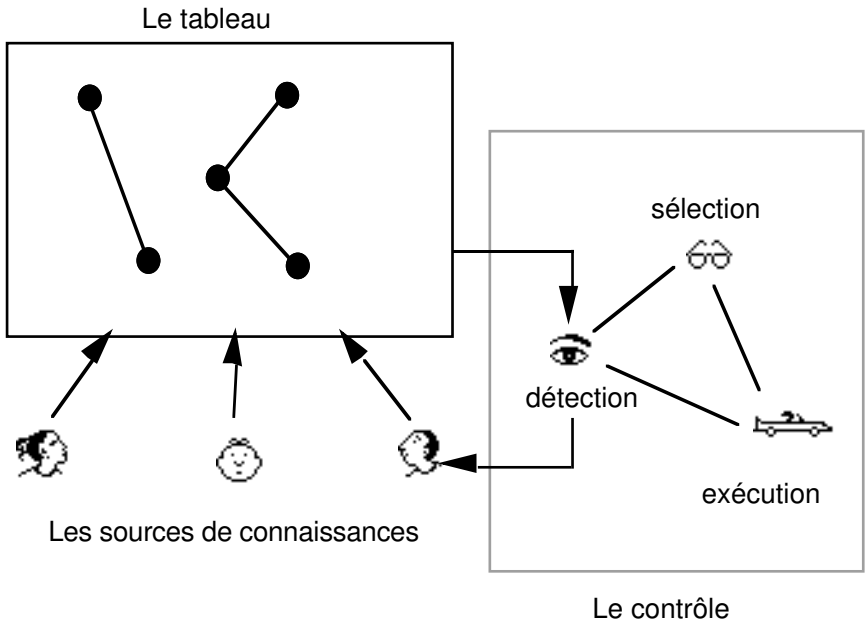


Figure 3.3 : Le cycle du contrôle est composé de la détection d'un événement, de la sélection d'une source de connaissances et d'un objet, et finalement de l'exécution de cette source avec le contexte de l'objet.

- *Exécution*: la phase d'exécution consiste à mettre à jour l'état du tableau au travers de l'exécution du corps de la source de connaissance, les autres structures de données servant à la résolution. Une fois que le système a sélectionné une source de connaissances, un objet, ou les deux, le système l'exécute ou le met à jour selon le cas. Il existe trois cas différents:
  - Une source de connaissances a été sélectionnée, le système sélectionne un objet, qui sert de contexte à la source de connaissances pour son exécution.
  - Un objet a été sélectionné, le système sélectionne une source de connaissances et l'exécute.
  - Enfin, si le système a déjà sélectionné l'objet et la source, il ne reste qu'à mettre à jour l'objet et à déclencher l'exécution de la source.

Il existe trois types de contrôle différents [KOE 89]: procédural, hiérarchique et opportuniste.

- Le contrôle procédural est constitué par un programme intégrant toutes les connaissances sur le contrôle. Cette structure est efficace, mais elle est aussi rigide, et dépend de l'application. Elle ne permet pas d'adapter les heuristiques de contrôle à l'état courant de la solution. Cette structure est adaptée pour des problèmes où les connaissances de contrôle sont uniformes et structurées.
- Le contrôle hiérarchique est constitué d'un ensemble de méta-sources de connaissances organisées en plusieurs niveaux de hiérarchie, agissant sur les sources de connaissances de domaine. Cette structure impose de connaître un ordre de séquencement des sources de connaissances, ce qui confère une certaine rigidité au système.
- Le contrôle opportuniste est composé d'un ensemble de sources de connaissances de contrôle agissant sur le "tableau noir" de contrôle (il existe un "tableau noir" indépendant

pour le contrôle et un autre pour la zone de données partagées), ce qui permet d'exprimer explicitement des heuristiques de résolution. Les heuristiques de contrôle s'adaptent dynamiquement à l'état courant de la solution. Le contrôle de la résolution fait l'objet d'un raisonnement propre.

Pour résumer, le contrôle gère les sources de connaissances; il contrôle doit déterminer à tout instant la source de connaissances la plus pertinente à activer selon l'état du "tableau noir".

3.1.4. Outils

Dans cette section nous décrivons deux réalisations du modèle de "tableau noir": Hearsay-III développé à Carnegie Mellon [BAL 80] [ERM 81] et BB1 développé à l'université de Stanford [HAY 84].

Le but des concepteurs de Hearsay III était de réaliser une architecture indépendante du domaine d'application, permettant de raisonner sur des solutions partielles et pouvant assurer la modularité de l'application.

Le "tableau noir" est divisé en "tableau noir" du domaine d'application et "tableau noir" du contrôle. Chacun d'eux est décomposé en niveaux hiérarchiques. Pour faciliter la maintenance et le raisonnement sur des hypothèses alternatives, les éléments du "tableau noir" ont été généralisés à des objets structurés plus complexes appelés UNITS. Un mécanisme de contexte a été introduit pour permettre des interprétations simultanées de la solution.

Les sources de connaissances peuvent êtres comparées de façon générale à des règles de production. La condition d'une source de connaissances est constituée de deux parties: le déclencheur et le code immédiat.

Le contrôle est de type opportuniste.

Pour BB1, le but était de construire un outil de développement de systèmes de "tableau noir" qui planifient explicitement et dynamiquement leur résolution.

Le système contient deux tableaux noirs: le domaine et le contrôle. Le "tableau noir" du domaine est structuré en niveaux d'abstraction: il peut être éclaté en plusieurs tableaux noirs. Un objet donné du "tableau noir" peut être relié à d'autres objets par des liens d'héritage. Le "tableau noir" de contrôle est propre à BB1, indépendant des applications.

BB1 différencie les sources de connaissances du domaine et du contrôle. Toutes les sources de connaissances ont la même structure. La condition d'une source de connaissances est composée d'un déclencheur qui teste sa pertinence et d'une précondition qui teste sa faisabilité. La partie action des sources de connaissances est un système de production local dans lequel tous les attributs de déclenchement des sources sont conservés.

Le contrôle est de type opportuniste.

3.1.5. Parallélisation du modèle

Comme nous l'avons vu, un système de "tableau noir" est un ensemble de spécialistes, qui travaillent en utilisant le "tableau noir" comme moyen de communication. Le modèle a été initialement conçu pour que les spécialistes puissent raisonner de façon parallèle et opportuniste, et arriver incrémentalement à la solution. Néanmoins la plupart des systèmes actuels de "tableau noir" ont été développés de façon séquentielle étant donné le mode séquentiel d'exécution des machines du modèle Von Neuman [JAG 89].

Une architecture de "tableau noir" parallèle entraîne l'exécution des sources de connaissances et des composantes du module de contrôle sur le même "tableau noir"[6]. Le plus grand avantage en utilisant des architectures parallèles est la possibilité de réaliser des modifications asynchrones sur le "tableau noir" et les données du contrôle, pour les spécialistes qui s'exécutent en parallèle. Les plus grandes contraintes des systèmes de "tableau noir" sur les architectures parallèles sont la sérialisation et le blocage, qui sont le résultat du mécanisme qui assure la cohérence des objets.

Nous présentons tout d'abord ce qui peut être traité en parallèle dans le modèle de "tableau noir" du point de vue logique, puis différentes possibilités pour la réalisation matérielle du modèle sur une machine parallèle.

---

[6] Les objets du tableau noir utilisent le même espace d'adresses.

3.1.5.1. Aspects parrallélisables dans le modèle de “tableau noir”

En général, le parallélisme peut être introduit dans les modèles de “tableau noir” à différents niveaux. Nii en introduit trois [NII 89] :

- Parallélisme sur les objets: les opérations de création, sélection, requête et modification peuvent être réalisées en parallèle (figure 3.4). Dans ce cas, il faut faire attention à la cohérence des objets, lorsque le système réalise différentes opérations en parallèle.

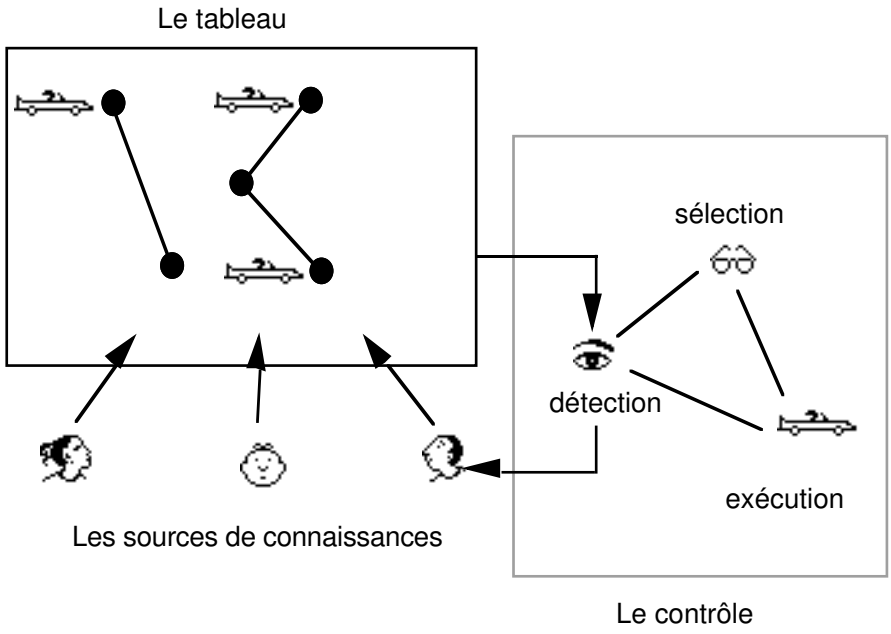


Figure 3.4 : Les différentes opérations sont exécutées en parallèle ( ) sur différents objets.

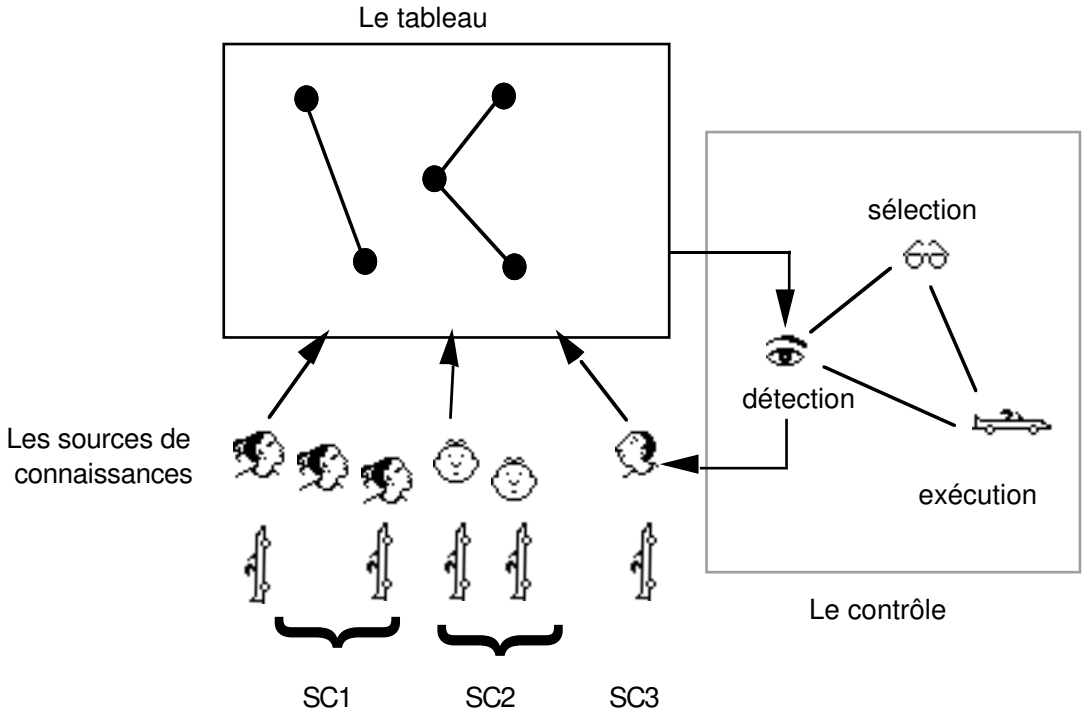


Figure 3.5 : Il existe plusieurs instances des sources SC1 et SC2, qui s’exécutent en parallèle ( ). Il n’est pas nécessaire que toutes les instances s’exécutent simultanément en parallèle.

- Exécution en parallèle de plusieurs sources de connaissances: cette approche reflète l'idée de base des systèmes de "tableau noir", qui est l'exécution en parallèle de plusieurs sources de connaissances, chacune avec son propre mode de raisonnement. Ces sources peuvent être toutes différentes, mais il est aussi possible que plusieurs instances de la même source s'exécutent en parallèle (figure 3.5).
- Parallélisme à l'intérieur des sources de connaissances : ce parallélisme dépend des tâches de chacune des sources de connaissances. Une source peut être parallélisable ou non; certaines sources peuvent travailler en "pipeline" (figure 3.6), et il y a des sources dans lesquelles certaines règles ou procédures peuvent être exécutées en parallèle [COR 89b].

D'autre part, divers auteurs, [COR 89a] [BIS 89] et [RIC 89], parlent aussi de la possibilité d'avoir le parallélisme au niveau du contrôle: certaines opérations peuvent être réalisées en parallèle, comme par exemple la sélection des différentes sources, le déclenchement des différentes sources de connaissances avec leurs contextes respectifs, etc. (figure 3.7).

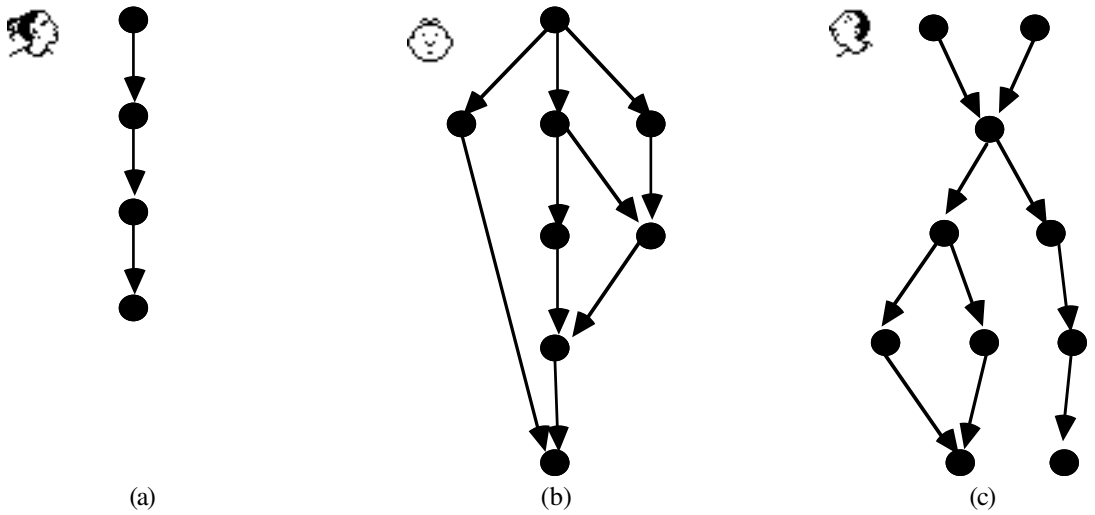


Figure 3.6 : Les sources de connaissances qui s'exécutent en "pipeline" (a). Sources dans lesquelles certaines tâches peuvent être exécutées en parallèle (il existe un certain ordre dans ces tâches) (b). Sources où la plupart des tâches peuvent être réalisées en parallèle (c).

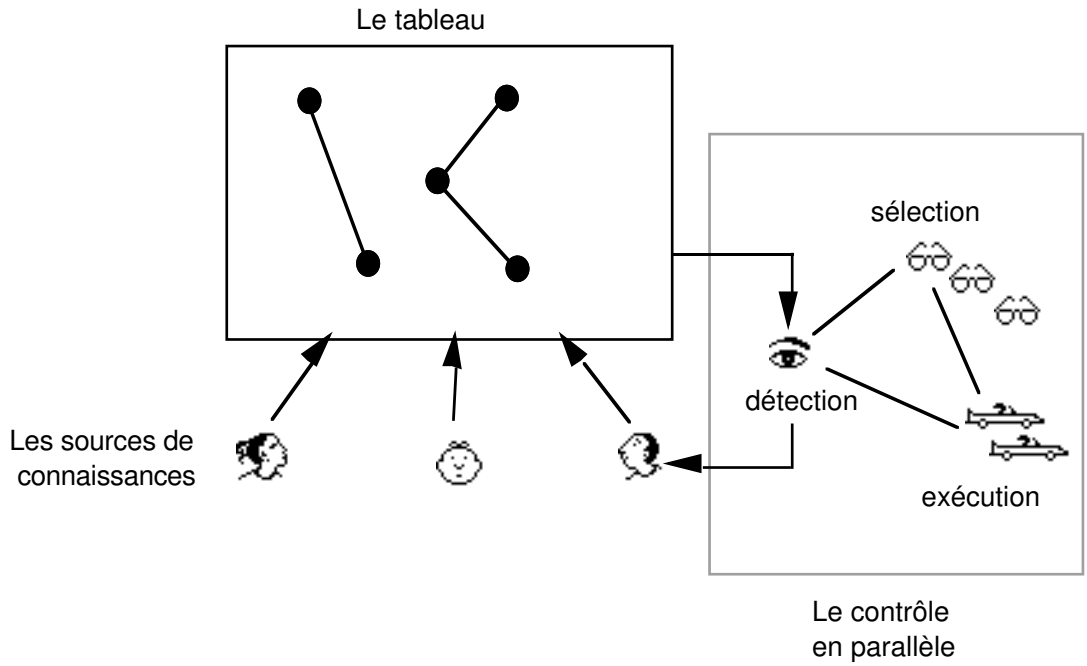


Figure 3.7 : Certaines opérations du contrôle peuvent être exécutées en parallèle. Par exemple, le système peut sélectionner plusieurs événements en même temps, ou lancer l'exécution des différentes sources de connaissances simultanément.

Dans les quatre cas précédents, une partie importante des différents systèmes est le contrôle. Si nous voulons, par exemple, une utilisation efficace des processeurs, il faut trouver la façon d'éviter la surcharge associée au placement de tâches dans les différents processeurs. Une autre source de grandes surcharges dans les systèmes parallèles est la communication entre les processeurs lorsque le contrôle est centralisé. Une solution à ces problèmes est proposée dans [RIC 89], et elle consiste à distribuer le contrôle. Cette solution entraîne une meilleure performance, mais elle crée des problèmes au niveau de la cohérence des objets et du comportement pour la résolution du problème, étant donné que cette solution est trouvée de façon incrémentale.
Une autre solution a été proposée dans [CHA 92]. Elle consiste à utiliser l'approche client/serveur conduisant à organiser la gestion du "tableau noir" (le client) autour du flot de données entre le "tableau noir" et les serveurs de traitement.

3.1.5.2. Réalisation matérielle du modèle sur une machine parallèle ou système réparti

Dans cette section, nous présentons différentes possibilités pour la réalisation matérielle du modèle de "tableau noir" proposées par Corkill [COR 89b].
Dans la suite, nous supposons qu'il existe la possibilité d'avoir plusieurs instances d'une même source de connaissances, et aussi que nous disposons d'une machine parallèle. Au fur et à mesure que les différentes instances sont créées, elles attendent qu'un processeur devienne libre pour pouvoir s'exécuter. Donc, il existe une file d'attente d'instances de sources de connaissances. Nous allons exposer trois possibilités différentes pour la parallélisation des systèmes de "tableau noir".

- Le "tableau noir" partagé: cette aproche permet aux différents processeurs l'accès direct au "tableau noir" pour réaliser les opérations dont ils ont besoin. Dans ce cas, la synchronisation entre les différents processeurs est très importante pour maintenir la cohérence des objets du "tableau noir" (figure 3.8).

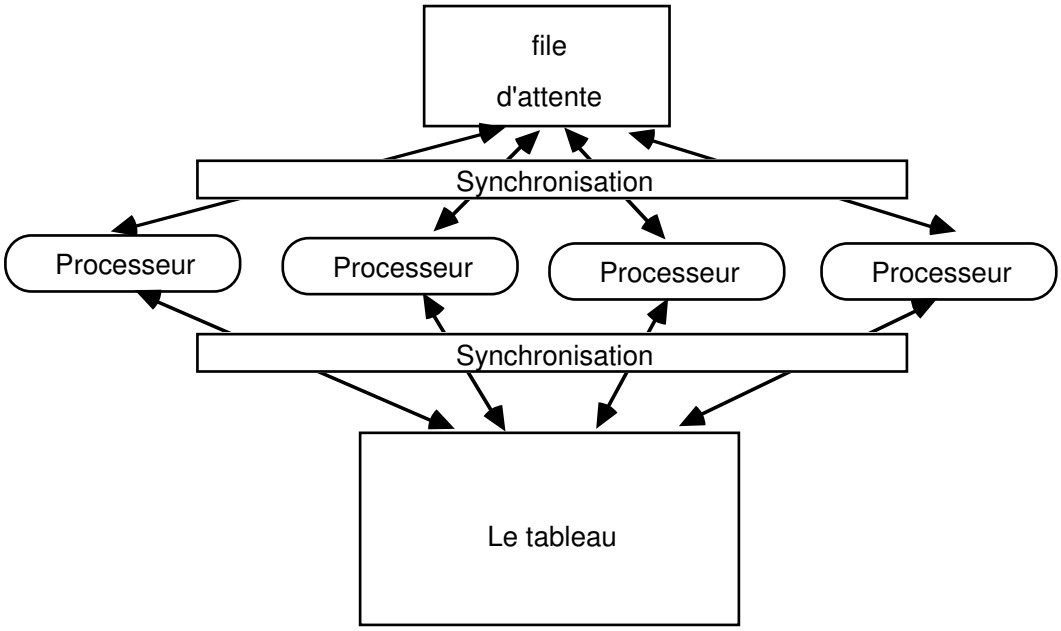


Figure 3.8 : Il existe plusieurs processeurs, qui accèdent en même temps au “tableau noir”. La synchronisation est indispensable.

Il existe aussi deux possibilités pour le placement des instances des différentes sources dans les processeurs:

- L’approche fonctionnelle, où chaque processeur est chargé d’une source de connaissances spécifique: toutes les instances de cette source doivent être exécutées par ce processeur.
- L’approche serveur, dans laquelle lorsqu’un processeur est libre, le système place la première instance de la file, sans prendre en compte le type de cette instance.

Cette possibilité pour la parallélisation du modèle de “tableau noir” est peut-être la plus simple à implémenter étant donnée la similarité entre le modèle et l’architecture :

- Un panneau de “tableau noir” pour chaque processeur: dans cette approche, chaque processeur traite un panneau différent de “tableau noir”, et le processeur peut réaliser, lorsqu’il exécute une source de connaissances quelconque, les différentes opérations sans risque d’avoir des problèmes de cohérence entre les objets de “tableau noir” (figure 3.9).

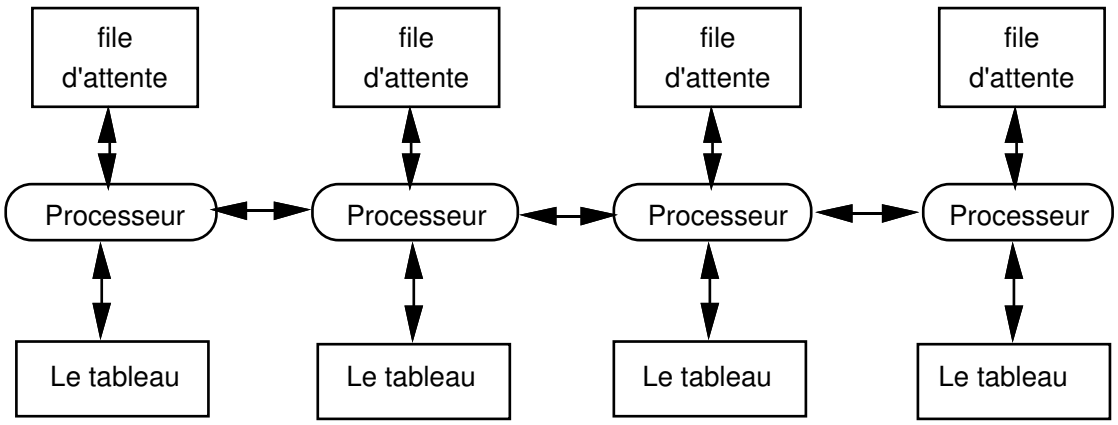


Figure 3.9 : Chaque processeur a sa propre file d’attente et son panneau de “tableau noir”.

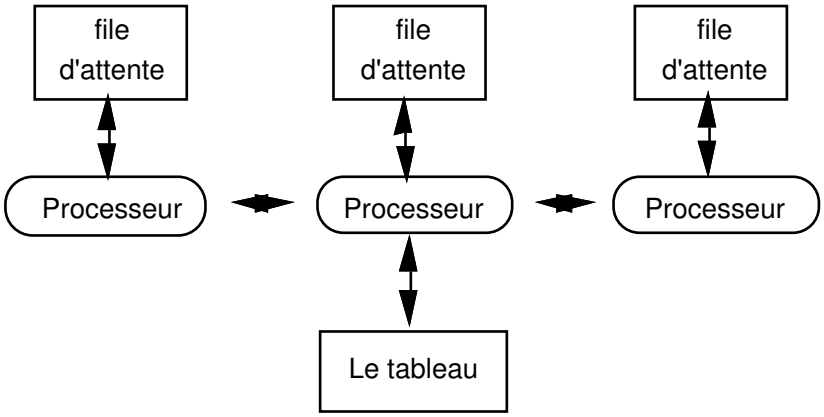


Figure 3.10 : Seul un processeur peut modifier le "tableau noir", et les autres envoient des requêtes à celui-ci.

- Un serveur commun: dans cette dernière approche, il n'y a qu'un processeur qui peut modifier le "tableau noir". Lorsque les autres processeurs ont besoin de se servir du "tableau noir", il font une requête au serveur (figure 3.10).

Nous avons présenté dans cette section différentes possibilités pour la réalisation matérielle du modèle de "tableau noir". Dans la section suivante nous présentons une réalisation matérielle du modèle existant déjà sur un système reparti : DSBB.

3.1.6. La mise en œuvre de DSBB : un "tableau noir" concurrent

DSBB a été développé à l'université des Andes [QUI 92c] dans le cadre du projet GEMINIS [QUI 92e]. L'objectif de ce projet est de développer un modèle qui exploite la concurrence dans un système réparti basé sur le système d'exploitation MACH [GOS 91].
DSBB reprend les idées de Poligon, qui a été développé à l'université de Stanford [RIC 86] [RIC 89] dans le cadre du projet AAP.
Dans la suite, nous présentons les différences entre les systèmes de "tableau noir" traditionnels et DSBB. Celui-ci a été conçu pour s'exécuter dans un système réparti dont la communication entre les processeurs ne se fait qu'à travers des messages [QUI 92f]. Les principales différences sont:

- DSBB n'a pas de planificateur ("scheduler") central. Cela permet d'éliminer les surcharges dérivées de l'exécution séquentielle du planificateur et de l'exécution en parallèle de plusieurs processus asynchrones, qui produisent des événements sur la file du planificateur.
- Comme conséquence de cette décision, il n'est pas nécessaire d'avoir séparé les sources de connaissances du "tableau noir". Les sources de connaissances sont donc associées aux éléments du "tableau noir" (figure 3.11) qui peuvent les déclencher lors de la production d'un événement. Cette association entraîne une diminution de la communication entre les différents processeurs.

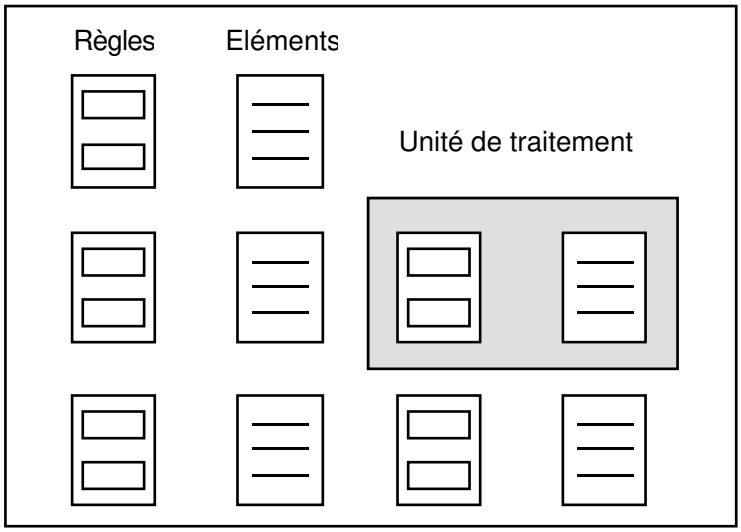


Figure 3.11 : Les sources de connaissances sont attachées aux éléments du “tableau noir”, et ils deviennent des unités de traitement. Cette association permet aux sources de modifier, si nécessaire, les valeurs des attributs des éléments (d’après [QUI 92c]).

- La communication entre les différents contrôles (associés chacun d’eux à un “tableau noir”) se fait de façon asynchrone.
- Le mécanisme de messages pour la communication entre les différents processeurs permet de voir chaque élément du “tableau noir” comme un processus [GOS 91] [TAN 92].
- Le coût de communication entre les différents processeurs a été ignoré, étant donné que le système a été développé sur un réseau à diffusion (figure 3.12).

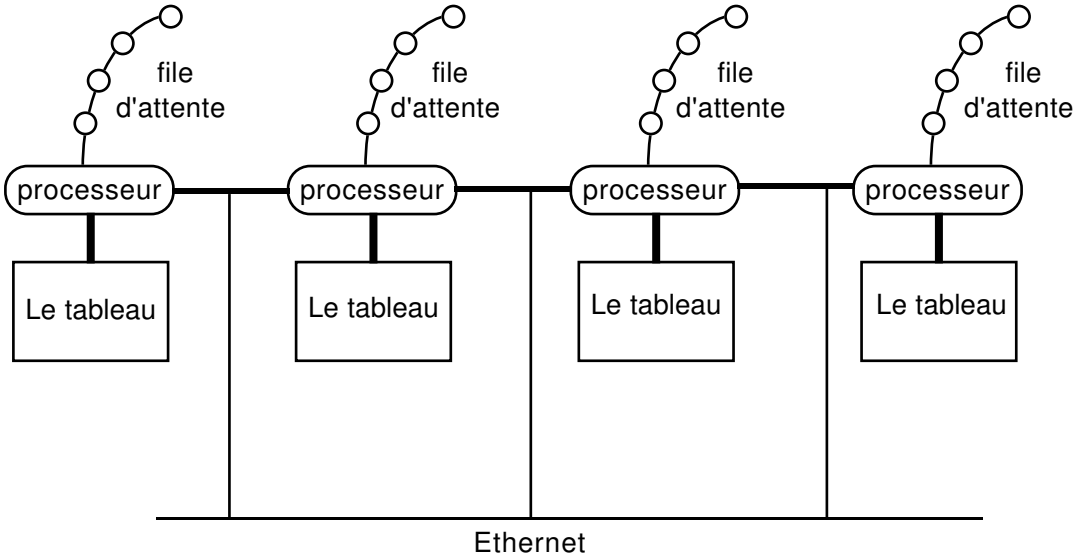


Figure 3.12 : Chaque processeur logique a sa propre file d’attente et son panneau de “tableau noir”. Il n’existe pas de “tableau noir” centralisé. La communication entre les processeurs est faite à travers un réseau à diffusion

Pour la mise en œuvre du système DSBB, nous avons utilisé le système d’exploitation MACH (annexe 1). Mach est un système intégré dont le noyau a été conçu comme une sorte de noyau Unix auquel a été rajouté le support nécessaire à l’exécution de plusieurs processus légers (“threads”) au sein d’un même processus Unix (programme 1). La raison de ce choix a été d’éviter la surcharge des machines lors de la création de beaucoup de processus égaux pour résoudre un même problème. Ce problème se présente en le système acteur que nous avons développé en utilisant Unix (§ 3.2).

D'abord, nous allons exprimer les fonctions dont nous avons besoin pour la gestion du tableau noir pendant les accès concurrents sur les mêmes objets. Puis, les sources de connaissances et finalement le tableau.

- Dans un système de tableau noir, les processus qui fonctionnent ensemble partagent un espace de données où chacun des processus peut lire ou écrire. Cet espace partagé est normalement situé dans la mémoire principale ou dans un fichier. L'emplacement de cet espace mémoire partagé n'influe pas sur la nature de la communication et sur les problèmes qui en découlent. Par exemple, des situations où plusieurs processus lisent ou écrivent des données partagées et où le résultat dépend de l'ordonnnacement des processus peuvent conduire à une incohérence du système.

  Pour éviter ces problèmes, qui peuvent produire des situations d'incohérence dans les données, il faut utiliser un moyen d'interdire la lecture ou l'écriture des données partagées à plus d'un processus à la fois : une exclusion mutuelle. Dans Mach, il existe des fonctions pour la gestion des sections critiques : mutex_lock et mutex_unlock (dans Mach C Threads Package). Néanmoins, l'utilisation de ces fonctions peut entraîner d'autres problèmes : les interblocages ("deadlocks"). Pour éviter les interblocages, nous avons réalisé une fonction VérificationCycle, qui est décrit dans la section 3.2.3.7.

```
#include <stdio.h>
#include <cthreads.h>

int             nombre;         /* nombre de processus légers : esclaves */
mutex_t         verrou;         /* variable pour l'exclusion mutuelle */
condition_t     fini;           /* signe à chaque fois qu'un processus léger termine */

void init()
{
        nombre = 0;
        verrou= mutex_alloc();   /* mutex assigné à la variable "fini" */
        fini = condition_alloc();
}

/* chaque esclave compte un numero "n". Lorsqu'il a fini exécute le signe "fini" */
void esclave(int n)
{
        int i;
        for (i=0;i<n;i+=1)
                cthread_yield();
        /* section critique */
        mutex_lock(verrou);
        nombre -=1;
        condition_signal(fini);
        mutex_unlock(verrou);
}
/* le maître crée un nombre "nesclave" de processus légers */
void maître(int nesclaves)
{
        int i;
        for (i=0;i<=nesclaves;i+=1) {
                mutex_lock(lock);
                nombre +=1;
                cthread_detach(cthread_fork(esclave,1000);
                mutex_unlock(lock);
        }
        mutex_lock(verrou);
        while (nombre != 0)
                condition_wait(fini,verrou);
        mutex_unlock(verrou);
        cthread_exit(0);
}
main()
{
        init();
        maître(16);
}
```

Programme 1 : Exemple d'un programme d'utilisation de processus légers. Il y a un processus maître, qui déclenche plusieurs processus esclaves et il attend que tous ces processus finissent. Dans ce cas particulier, les processus esclaves partagent entre tous une variable commune "nombre". Pour éviter les conflits d'accès à cette variable ou à un objet en général, il nous faut une exclusion mutuelle (dans la section critique) qui empêche les autres processus d'accéder à un objet partagé si cet objet est train d'être utilisé par un processus.

Il existe un autre problème lors de l'exécution répartie : la mise-à-jour des objets éloignés et partagés. Dans ce cas, nous ne pouvons pas utiliser les fonctions d'exclusion mutuelle de Mach : mutex_lock et mutex_unlock. Nous avons donc développé un ensemble de fonctions pour la mise-à-jour d'un ensemble d'objets distribues. On dit qu'on sérialise les mises à jour pour que, même si elles sont simultanées, elles soient équivalentes à une exécution séquentielle. Les techniques mises en œuvre pour cette sérialisation sont appelées algorithmes de contrôle des accès concurrents. Ces fonctions ont été faites à

partir de l'algorithme de verrouillage à deux étapes et de l'algorithme de validation à deux étapes [GOS 91].

- Dans DSBB, chaque source de connaissances doit savoir les conditions sur lesquelles elle peut contribuer à la solution, et les préconditions nécessaires pour son déclenchement. Donc, une source de connaissances est composée de deux parties: la précondition pour son déclenchement et du corps qui constitue la partie action. Ces connaissances sont représentées de façon déclarative comme un ensemble de règles.

  Une régle a une forme :

```
if "condition de déclenchement-1" then
   corps : la partie action-1
elsf "condition de déclenchement-2" then
   corps : la partie action-2
elsf "condition de déclenchement-3" then
   corps : la partie action-3
…
else
   corps : la partie action-N
endif
```

  Cette régle est traduite pour être compilée en un programme C de la façon suivante :

```
Regle(structures de données associées à la régle) {
if ("condition de déclenchement-1") {
   corps : la partie action-1;
} else {
        if  ("condition de déclenchement-2") {
        corps : la partie action-2;
        } else {
                if ("condition de déclenchement-3") {
                corps : la partie action-3;
                } else {
                corps : la partie action-N;
                }
}
```

- Finalement, les structures de données du tableau : nous avons attaché aux structures de données pendant la compilation beaucoup d'informations, afin de réduire le travail du contrôle. Une des informations la plus importante qui est attachée à un nœud du tableau est une liste de régles. Ces régles peuvent être déclenchées lors d'un changement du nœud. Cette implémentation nous permet de placer facilement un tableau sur une machine quelconque et de réduire considérablement la communication entre les différents contrôles.

3.2. Le modèle acteur

Intuitivement, un Acteur est un agent actif, et celui-ci joue un rôle où les répliques sont conformes à un scénario. Notre formalisme montre comment tous les modes de comportement peuvent se définir en termes d'un type unique de comportement : envoyer des messages à des acteurs (pris de [HEW 73]).
Un acteur peut être considéré comme un expert vivant en société et communiquant avec d'autres experts pour résoudre des problèmes. Chaque expert peut lui-même être une société d'experts plus élémentaires. Le modèle est donc réparti : les connaissances et le comportement sont diffusés parmi les acteurs, qui effectuent des tâches en parallèle, de manière indépendante.

3.2.1. Historique

Le modèle "acteur" pour la programmation concurrente a été proposé par Carl Hewitt [HEW 77]. Celui-ci permet de modéliser des objets partagés avec des changements locaux de son état, une reconfiguration dynamique des liens entre les objets et le comportement concurrent des objets (les acteurs). Gul Agha [AGH 86] a défini le modèle "acteur" avec un petit nombre de

primitives très puissantes qui sont la base du langage Act++ [KAF 90] et de notre langage LDBA [QUI 92d].
Dans la suite, nous décrivons le modèle "acteur", puis le langage LDBA et finalement, nous présentons un exemple en utilisant LDBA.

3.2.2. Définition du modèle

Le modèle "acteur" utilise le concept d'encapsulation, c'est-à-dire que chaque objet nommé Acteur est un agent actif et autonome qui communique librement avec les autres par envoi de messages. L'univers des acteurs évolue dynamiquement suivant les différentes activités des acteurs. Ces derniers communiquent entre eux pour déléguer des tâches et pour transmettre des résultats. En général, tous les langages acteurs ont ces mêmes caractéristiques; néanmoins, dans le modèle proposé par Gul Agha il existe une différence : les changements d'état de l'acteur sont remplacés par un changement de son comportement [AGH 86][MAS 89].

Dans la suite un acteur sera décrit par :

- Une adresse, dans laquelle il existe une boîte aux lettres, qui contient les messages destinés à l'acteur. Ces messages sont ordonnés selon l'ordre d'arrivée.
- Des données locales, appelées accointances, qui sont les autres acteurs que l'acteur connaît directement. Elles correspondent aux variables d'instance d'une classe.
- Le comportement, qui correspond aux actions qui doivent être faites par l'acteur à la réception d'un message.

La notion d'héritage n'existe pas dans ce modèle. Un acteur est une entité autonome, qui détient toute la connaissance requise pour effectuer des tâches spécialisées. Un acteur peut être vu comme une file d'attente de messages (FIFO) et une machine A_n [MAS 89] qui traite le message courant de la file. Lorsque la machine accepte un message, elle peut envoyer des messages aux autres acteurs dont elle connaît les adresses, et elle peut créer de nouveaux acteurs. Elle doit aussi créer une nouvelle machine A_{n+1} qui va traiter le message n+1 (figure 3.13). C'est-à-dire que l'acteur doit spécifier un comportement de remplacement pour le traitement du message n+1.
Les machines A_n et A_{n+1} sont indépendantes et chacune d'entre elles ne traite qu'un message. Ces machines encapsulent leur comportement, sans pouvoir agir sur les comportements des autres machines. Elles ne peuvent pas non plus recevoir d'autres messages, ni spécifier d'autres comportements de remplacement C'est-à-dire que l'acteur ne connaît que l'interface de ses accointances, qui peuvent être considérées comme une abstraction de leur représentation physique.

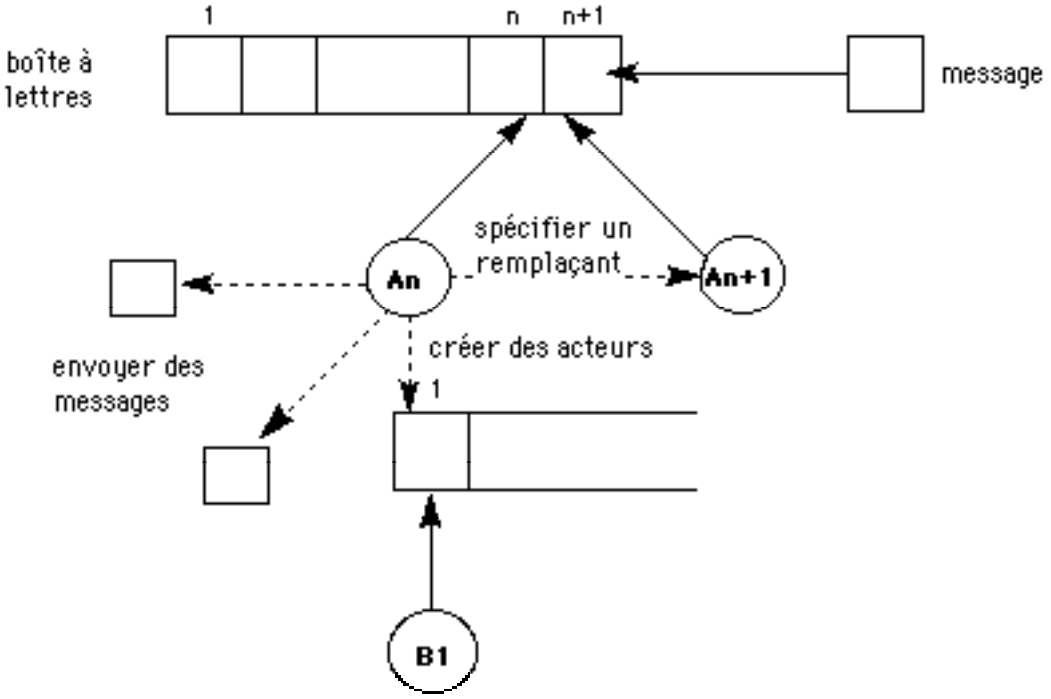


Figure 3.13 : Le comportement d'un acteur A_n pendant le traitement du n-ième message : il peut envoyer des messages aux autres acteurs, créer d'autres acteurs, et il doit spécifier son comportement de remplacement A_{n+1} pour le traitement du message n+1.

Le comportement d'un acteur est décrit par une liste de connaissances, ou relations, qui représente les données locales de l'acteur, et par une liste des messages reçus. Les acteurs qui spécifient leur comportement de remplacement avec des relations identiques sont non-sérialisés : chacun d'entre eux peut traiter sans problème plusieurs messages en parallèle. Par contre, spécifier le comportement de remplacement avec des relations différentes entraîne un changement d'état, aussi ces acteurs sont-ils sérialisés.

3.2.3. Description de LDBA : langage acteur réparti

Le système LDBA a été développé à l'Université des Andes [QUI 92d][MEN 91] dans la cadre du projet GEMINIS au sein de l'équipe HIDRA. L'objectif de ce projet est de développer un langage Acteur tirant parti du parallélisme d'un système réparti local basé sur UNIX. LDBA est une mise en œuvre du langage ACT++ [KAF 90], lequel est basé sur le modèle de Gul Agha. En pratique, la réalisation de LDBA est une extension de C++ [STR 86], avec une hiérarchie de classes qui permet l'exploitation du parallélisme inhérent au modèle "acteur" dans un réseau local.

3.2.3.1. Les primitives de LDBA

Le modèle "acteur" [AGH 86] est un modèle de programmation concurrente dans lequel le traitement est mené par les acteurs, qui communiquent entre eux par échanges de messages. Le modèle comprend quatre éléments de base : les acteurs, les boîtes aux lettres, les comportements et les relations.

Un acteur est un objet suffisant. L'interaction entre les acteurs ne se fait qu'à travers les messages. Chaque acteur est associé à une boîte aux lettres, dont l'adresse est utilisée pour identifier l'acteur. Les messages qui ont été envoyés à un acteur sont gardés dans sa boîte aux lettres et ils sont lus un par un dans un ordre FIFO. La lecture de la boîte aux lettres est une action bloquante. Si un acteur essaie de lire un message de sa boîte aux lettres et qu'elle est vide, il sera bloqué jusqu'à l'arrivée d'un message. Un acteur A peut envoyer un message à un acteur B, seulement si A connaît l'adresse de la boîte aux lettres de l'acteur B. Cette adresse est connue par l'acteur qui a créé B. Donc, l'acteur qui crée B peut l'envoyer dans un message aux

autres acteurs. Ces deux dernières propriétés permettent de reconfigurer dynamiquement l'interconnexion entre les acteurs.

Le comportement d'un acteur détermine sa façon de réagir aux demandes spécifiées dans le message. Le comportement d'un acteur est défini par le corps **behavior script**. Ce script contient les définitions des méthodes et des relations. Ces relations sont les adresses des autres acteurs auxquels l'acteur peut envoyer des messages. Le système est homogène et les seules entités qui peuvent exister sont les acteurs. Les relations du modèle de Agha sont les variables, qui ne peuvent être utilisées que pour la lecture, et non pour l'écriture. En effet, la concurrence au niveau des instructions ne peut pas être garantie s'il existe des dépendances de données entre les instructions (étant donné que chaque instruction est exécutée en parallèle : la granularité est fine[7]). Dans notre langage LDBA, les relations peuvent être modifiées grâce à l'utilisation de la granularité au niveau de méthodes (à grande granularité).
Dans la suite nous décrivons les différentes actions qui peuvent se dérouler dans le modèle : créer de nouveaux acteurs, envoyer des messages aux autres acteurs et spécifier un comportement de remplacement.

3.2.3.2. La création des acteurs

En LDBA, ils existent deux types d'objets : les passifs et les actifs. Les objets actifs sont sous-classes de la classe **Acteur** (définie dans le langage), tandis que les objets passifs constituent toutes les autres classes (définies par l'utilisateur). Les acteurs sont créés dynamiquement en accord avec les besoins grâce à l'opérateur **New** (cet opérateur est différent au new de C++). L'opérateur New reçoit comme argument le nom de la classe acteur qu'il faut créer, et celui-ci retourne l'adresse de sa boîte aux lettres. A partir de ce moment les autres acteurs peuvent lui envoyer des messages dans sa boîte aux lettres. Cette adresse est connue par l'acteur qui l'a créé (à partir de New) et celui-ci peut envoyer par messages cette adresse aux autres acteurs.

3.2.3.3. L'envoi de messages

La sémantique pour appeler les méthodes d'un objet passif est la même que celle d'un appel normal d'une fonction dans un programme, et peut être assimilée à celle des transmissions synchrones de ABCL/1 [YON 87]. Les méthodes d'un acteur sont appelées à partir d'un message asynchrone. Il existe deux types de messages : les messages de demande et les messages de réponse.
Un message de demande est utilisé pour appeler une méthode, et un message de réponse est utilisé pour rendre les résultats d'une méthode. Pour ces deux types de messages, il existe aussi deux sortes de files d'attente de messages : les **Mboxes** et les **Cboxes**. Les Mboxes sont les boîtes aux lettres d'un acteur et les Cboxes permettent la programmation avec "**futures"** [LIE 87], qui correspond à une promesse de livraison (réponse). Dans ce cas là, le calcul se fait de manière parallèle.
Un message de demande est composé du nom de la méthode qui doit être exécutée par l'acteur qui reçoit le message, et des arguments de la méthode. Pour recevoir une réponse, il faut spécifier une Cbox dans les arguments comme l'identification de **l'acteur continuation** [HEW 77], qui est l'acteur qui va recevoir la réponse du message de demande. Si nous ne spécifions aucun acteur continuation, nous avons un asynchronisme pur, et si nous spécifions les acteurs continuation, nous avons les transmissions anticipées de ABCL/1 [YON 87]. Un acteur peut recevoir un message d'une Cbox grâce à l'opérateur >> et il peut envoyer un message grâce à l'opérateur <<. Par exemple, si nous avons un acteur **additionneur** dont Mbox est **mboxSum** et avec une méthode **Addition**, qui permet d'additionner deux entiers, l'appel sera :

```
mboxSum << "additionneur::Addition" << entier_1 <<entier_2 << maCbox;
```

---

[7] Définition de la granularité : Quelque que soit le paradigme utilisé, le résultat de la décomposition du problème est un ensemble de traitements dont la cardinalité représente le degré de parallélisme que nous avons pu extraire du problème global. Le rapport de la durée moyenne d'une séquence de traitements à la durée moyenne d'une communication est la **granularité**; elle exprime le degré d'importance relative des communications dans le temps de résolution du problème.

où **maCbox** est la Cbox où sera laissé le résultat de l'addition.

Il existe principalement deux raisons pour l'existence des Cbox :

- Permettre une implantation efficace d'un acteur qui attend une réponse et qui ne peut pas continuer son exécution.

- La Cbox est un canal de communication entre les acteurs. Elle permet à l'acteur, qui a envoyé un message de demande, d'attendre la réponse directement de l'acteur qui a été créé pour satisfaire la demande.

Un acteur peut faire référence à sa Mbox à travers la variable **MaMbox**. Pour le même exemple précédent, la méthode Addition peut obtenir ses arguments (les variables à additionner) par:

```
MaMbox >> par_1 >> par_2 >> UneCbox;
```

Comme le nom de la méthode a été utilisé pour sélectionner son appel, en MaMbox seuls les arguments de la méthode sont reçus. Le nombre d'arguments d'un appel à une méthode est donné dans le premier argument. Par exemple, si la méthode Addition de l'acteur additionneur reçoit le numéro d'arguments dans le premier argument, l'appel à la méthode sera :

```
Mail  monMail;
…
monMail = (mboxSum << "additionneur::Addition");
monMail << 2 << UneCbox << entier_1 << entier_2;
UneCbox >> laAddition;
```

Dans l'exemple suivant, une réponse comme CboxResp << addition est gardée dans la Cbox [CAR 88]. S'il n'y a aucun message, l'acteur est bloqué jusqu'à l'arrivée d'un message. Néanmoins, il existe un opérateur **in()**, qui peut être utilisé pour l'acteur par savoir s'il y a des messages dans sa Cbox.

La méthode **addition** va être définie comme :

```
int addition, par;
Cbox CboxResp;

addition= 0;
MiMbox >> n >> CboxResp;
for (i = 0; i < n; i++) {
   MiMbox >> par;
   addition+= par;
}
CboxResp << addition;
```

3.2.3.4. Spécification d'un comportement de remplacement

Un comportement est écrit en utilisant les primitives des acteurs et les constructions de C++. Les instructions du comportement sont exécutées de façon séquentielle. L'opération **become** permet de changer le comportement d'un acteur dynamiquement pendant son exécution. Un même acteur peut être associé à différents comportements à différents moments. Ces comportements peuvent avoir différentes méthodes ou structures de données, mais l'adresse de la boîte aux lettres est toujours la même.

Dans le modèle de Agha, l'opération "become" a besoin du nom d'un comportement et d'une liste de relations. Dans notre modèle, l'opération "become" reçoit un pointeur sur un nouvel acteur ou la constante **Self**. Cette constante est utilisée pour spécifier un comportement de remplacement à l'acteur lui-même.

L'opération "become"a deux rôles importants dans le modèle :

- c'est la seule façon qu'a un acteur pour changer son état en permanence.

- celle-ci maintient aussi la consistance des relations de l'acteur.

Dans notre réalisation, l'utilisateur indique explicitement si un acteur est sérialisé (c'est-à-dire qu'il n'exécute qu'une méthode à la fois) par la constante **SERIALISEE** comme argument pour la construction des méthodes. Par exemple, si l'acteur **Magasin** reçoit le message retrait (faire un retrait), il décrémente sa quantité de la valeur demandée, la méthode sera :

```
Magasin::Magasin(void)
:Acteur (SERIALISEE)
{quantité = MAX_ARTICLE;}


void Magasin::retrait (void)
{
int demande;


MaMbox >> demande;
if (quantité==0)
   exit (ERREUR);
 else {
   quantité = quantité -demande;
   become (Self);
 }
;
}
```

Au contraire, si l'utilisateur ne spécifie pas que les méthodes sont non-sérialisées, l'acteur permet d'exécuter plusieurs instances des mêmes méthodes sans aucun mécanisme de synchronisation [KAF 90]. Dans le cas de l'exemple précédent, si l'acteur Magasin n'est pas sérialisé, nous pouvons avoir des inconsistances dans l'état de l'acteur lors des changements de ses relations. L'opération become(Self) est nécessaire, car il faut que le décrément de la variable Article soit permanent.

3.2.3.5. Résumé des opérations

La table 3.1 montre les différentes opérations qui ont été développées dans LDBA.

Opération	**Classe**	**Signification**
New	aucune	Créer un nouvel acteur
become	Acteur	Spécifier le comportement de remplacement
<<	MBox, Cbox, mail	Envoyer des messages
>>	Mbox, Cbox	Recevoir des messages
in	Cbox	Voir si la Cbox est vide

Table 3.1: Opérations du modèle "acteur" en LDBA.

3.2.3.6. Les opérations éloignées

L'existence des listes de demande (MBoxes) et de réponse (CBoxes) permet de faire une réalisation répartie du modèle "acteur" comme dans LDBA. Dans cette réalisation répartie, la communication entre les méthodes, qui peuvent être exécutées dans des différentes machines, se fait de façon totalement transparente pour l'utilisateur. Le déclenchement réparti des méthodes est aussi transparent pour l'utilisateur. Pour le faire, nous avons utilisé le paradigme client-serveur, et nous avons défini un langage très simple avec un style d'appel de procédure à distance RPC ("remote procedure call") [BIR 84] pour les opérations éloignées. Dans ce cas, l'émetteur est bloqué jusqu'à ce que le processus destinataire du message en ait effectué le traitement et ait renvoyé une réponse. Le placement des processus qui correspondent aux méthodes se fait de telle façon que la répartition de la charge de travail des processeurs soit équilibrée.

Un exemple : Pour montrer le fonctionnement du système, nous présentons l'exemple de la multiplication de deux matrices.

```
class Multiplicateur: public Acteur {
   public:
     Multiplicateur(void);
     Multiplicateur(void);
     void Multiplicateur(void);
     int Produit (int file[3], int matrice[3][3], int col, int tam);
};

Multiplicateur::Multiplicateur(void)
: Acteur ()
{ }

Multiplicateur::Multiplicateur(void)
{ }
```

```
void Multiplicateur::Multiplier (void)
{
Cbox  Resp;
 int   n,i,j;
 int   oB[3][3];
 int   file_Ai[3];

// Reçoit le nombre de colonnes de la matrice
 MaMbox >> n;
// Reçoit une file de la matrice A, c'est-à-dire A[i][]
 for (i = 0; i < n; i=i+1)
    MaMbox >> file_Ai[i]
 ;
// Reçoit la matrice B
 for (i = 0; i < n; i=i+1)
    for (j = 0; j < n; j=j+1)
       MaMbox >> oB[i][j]
 ;
// Reçoit la Cbox dans laquelle il faut envoyer la matrice résultat.
 MaMbox >> Resp;
 for (i = 0; i < n; i=i+1)
    Resp << Produit (file_Ai, oB, i, n)
 ;
}
```

```
int Multiplicateur::Produit (int file[3], int matrice[3][3], int col, int tam)
{
 int sum, i;
 sum = 0;
 for (i = 0; i < tam; i=i+1)
    sum = sum + file[i] * matrice[i][col]
 ;
 return sum;
}

void EcrireMatrice (int mat[3][3], int m, int n)
{
 int i,j;
// Ecrit la matrice mat
 for (i = 0; i < m; i=i+1) {
    for (j = 0; j < n; j=j+1)
       printf ("%d ", mat[i][j])
    ;
    printf ("\n");
 }
}

void LireMatrice (int mat[3][3], int m, int n)
{
 int i,j;
 for (i = 0; i < m;i=i+1)
    for (j = 0; j < n; j=j+1) {
       printf ("Position [%d][%d]: ", i, j);
       scanf ("%d", &mat[i][j]);
    }
 ;
}
```

```
int main ()
{
 Mbox   Acteurs[3];
 Cbox    RespFiles[3];
 Courrier Où[3];
 int    A[3][3],
         B[3][3],
         C[3][3];
 int   i,j,k,l;

 // Synchronization et initialisation du système
 if (start() == -1) {
    cout << “Erreur \n.”;
    exit (1);
  }
 LireMatrice (A, 3, 3);
 LireMatrice (B, 3, 3);

 // Le programme crée les acteurs Multiplicateurs
 for (i = 0; i < 3; i=i+1)
    Acteurs[i] = New (“Multiplicateur”)
 ;
 for (i = 0; i < 3; i=i+1) {
    Où[i]=(Acteurs[i]<<“Multiplicateur::Multiplier”);
    Où[i] << 3;
    for (j = 0; j < 3; j=j+1)
       Où[i] << A[i][j]
    ;
    for (k = 0; k < 3; k=k+1)
       for (l = 0; l < 3; l=l+1)
          Où[i] << B[k][l]
    ;
    Où[i] << RespFiles[i];
 }

 // Nous attendons les files résultats de la multiplication
```

```
    for (i = 0; i < 3; i=i+1)
        for (j = 0; j < 3; j=j+1)
            RespFiles[i] >> C[i][j]
    ;
    EcrireMatrice (C, 3, 3);
    end();
    return 0;
}

methods Multiplicateur in multiplicateur {
    void Multiplier ();
}
```

Dans la figure 3.14, nous pouvons voir l'illustration de l'exemple précédent.

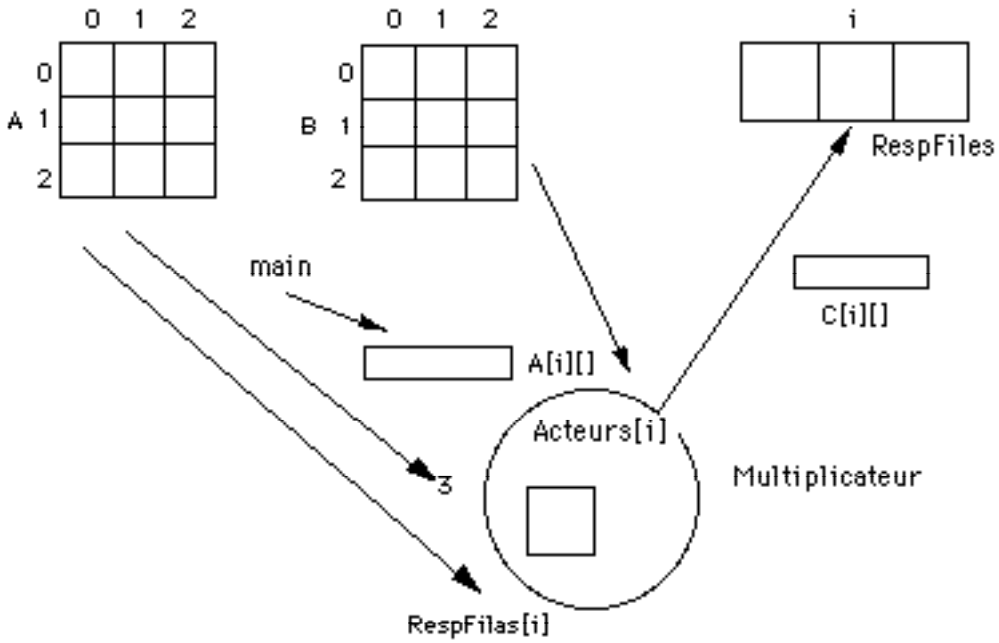


Figure 3.14 : Création et fonctionnement des acteurs multiplicateurs.

3.2.3.7 La mise en œuvre du LDBA

Dans cette section, nous présentons, dans un premier temps, la mise en œuvre du LDBA et finalement, nous présentons les problèmes techniques rencontrés.

Le système du LDBA a été construit en C++, l'interface a été construite en X-Windwos et pour la communication physique des acteurs sur plusieurs machines DEC, RISC6000 et Hewlet Packard connectées par un réseau Ethernet nous avons utilisé les "sockets". Tout le système a été développé sous Unix 4.3 BSD.

Lors de l'exécution d'un programme, le système choisit un machine pour que celle-ci devienne "le maître" et les autres machines deviennent "esclaves". Le maître gère le système pour le correct déroulement correct du programme : il fait la synchronisation des machines (fonction "start" : programme 2), décide dans quelle machine est créé un nouveau acteur et finalement

active la communication avec les différents machines (fonction activer_machine : programme 3).

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// int start (void)
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//| PROPOS: Fonction pour initialiser le système. Elle demande pour l'état du processeur MAITRE
//| ou ESCLAVE, ouvre son boîte aux lettres, etc.
//|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

int start (void) {
    PAQ paq;
    int i,err;

   // Pour ignorer le signal de fin ("exit") des processus
    signal (SIGCLD, SIG_IGN);
  cout << "machine_maître= " << machine_maître << "\n";
  cout << "machine_1= " << machine_1 << "\n";
   // Boîte aux lettres pour la révision de interblocages
   if( nobloquoes() == ERREUR) return ERREUR;
    for(i=MAITRE ; i< NUM_PROCESSEURS; i++){
      if(activer_machine(i) == ERREUR)
         return ERREUR ;
      if (recevoir_de_machine(i) == ERREUR)
         return ERREUR;
    }
    for(i=MAITRE ; i< NUM_PROCESSEURS; i++)
     if (envoyer_à_machine(i) == ERREUR) return ERREUR;

    for(i=MAITRE ; i< NUM_PROCESSEURS; i++){
       if( i == MAITRE )
        if ( (write(sck[MAITRE],paq.datos,MAXDATOS) != MAXDATOS) && (errno) ){
             cerr << "ERREUR:  MAITRE\n";
             return(ERREUR);
          }
       if( close(sck[i]) != OK ){
            pERREUR("ERREUR: sck[]");
            return ERREUR;
        }
  }
   return OK;
}
```

Programme 2 : Fonction start - Initialisation et synchronisation du système

```
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// int activer_machine (T_Proc NumeroMachine)
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//| PROPOS:   Activer le sous-système de communication d'un machine remote
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
int  activer_machine( T_Proc NumeroMachine)
{/*-------------------------------------------------------------------------*/
char                    c;
int                     fd;
struct hostent          *hp;
int                     i;
struct protoent         *ptr;
int                     port;
int                     sock_global;
struct sockaddr_in      saddr;
struct servent          *sp;
char                    machine[MAX_STR];
PAQ                     paq;

       cout << "Activer machine " << NumeroMachine << "\n";
       if(NumeroMachine == 0)
         sprintf(machine,machine_maître.theString);
       if (NumeroMachine >= NUM_PROCESSEURS){
              cerr << "ERREUR: NumeroMachine= " << NumeroMachine << "doit être plus petite que
NUM_PROCESSEURS= " << NUM_PROCESSEURS << "\n";
         return(ERREUR);
        }
       /* Obtenir information sur une machine remote */
       cout << "machine= " << machine << "\n";
      if ((hp = gethostbyname(machine)) == NULL){
           pERREUR("");
           cerr << "ERREUR: La machine n'existe pas : " << machine << "\n";
           return(ERREUR); }
      if ((sp = getservbyname("ldba", "tcp")) == NULL){
          pERREUR("ERREUR: il n'existe pas d'information au sujet de LDBA");
           return(ERREUR);  }
       saddr.sin_port = sp->s_port;
      bcopy(hp->h_addr, &saddr.sin_addr, hp->h_length);
       saddr.sin_family =  AF_INET;
       ptr = getprotobyname("tcp");
       if ((sck[NumeroMachine] = socket(AF_INET, SOCK_STREAM, ptr->p_proto)) < 0){
           cerr << "ERREUR: Il n'est pas possible de créer le socket\n";
           return(ERREUR);  }
      if (connect(sck[NumeroMachine], &saddr, sizeof(saddr)) != OK){
           cerr << "ERREUR: La connexion ne marche pas\n";
           return(ERREUR); }
       paq.REWRITE();
      paq.PAQT_Proc(NumeroMachine);
      if ( write(sck[NumeroMachine],paq.datos,MAXDATOS) != MAXDATOS ){
          cerr << "ERREUR: NumeroMachine\n";
          return(ERREUR); }
       return(OK);
/*-------------------------------------------------------------------------*/}
```

Programme 3 : Fonction activer_machine - cette fonction active le système de communication d'une machine eloignée.

Pour l'exécution d'un programme écrit en LDBA, il faut d'abord compiler le programme de l'utilisateur avec les différents modules du système LDBA. Cette compilation crée un fichier C++ et finalement, il faut compiler ce fichier avec le compilateur de C++.

Le système LDBA est composé de 5 modules : un module d'interface, un module du compilation ("parsing"), un module de fonctions de LDBA, un module des "include" et finalement le module de l'utilisateur.

- Un module d'interface : qui a été développé en X-Windows. Dans ce module il y a un ensemble de fonctions pour que les acteurs puissent ouvrir des fenêtres et établir des dialogues avec l'utilisateur. Etant donné que le programme s'exécute sur un réseau, les différents acteurs s'exécutent sur divers machines. Cette exécution répartie entraîne un problème : si un acteur veut ouvrir une fenêtre, celle-ci s'ouvrira-t-elle sur la machine où s'exécute l'acteur ou bien sur la machine maître ?. Nous avons pris la décision d'ouvrir toutes les fenêtres sur la machine maître, étant donné que pour l'utilisateur l'exécution répartie doit être absolument transparente pour lui.
- Un module de compilation : ce module fait la transformation d'un programme écrit en LDBA en code C++. Pour cette transformation, nous avons utilisé Yacc.
- Un module de fonctions du LDBA : dans ce module ce trouvent toutes les fonctions propres d'un langage acteur. Par exemple la fonction New (programme 4).

```
//|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//| Acteur::Acteur (int Type)
//|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//| PROPOS: Construction de la Classe Acteur.
//|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Acteur::Acteur (int Type)
: BoîteLettresPet()
{
   TypeActeur = Type;
   BoîteLettresPet.ChangeTypeActeur (Type);
}

//|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//| Acteur::~Acteur(void)
//|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//| PROPOS: Détruire un objet de la Classe Acteur
//|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Acteur::~Acteur(void) {

   BoîteLettresPet.Mbox::~Mbox();
}

//|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//| Mbox& New (char* NomActeur)
//|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//| PROPOS: Créer une instance d'un acteur Nomcteur et retourner son adresse (Mbox)
//| Attention : Pour la création d'un acteur, on fait une requête au processus de communication
//| local avec le nom de l'acteur. Après le processus de communication nous retourne
//| l'adresse (la machine et la Mbox) où a été créé l'acteur. Pour créer l'acteur, le système
//| crée un processus Unix (avec fork()).
//|~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Mbox& New (char* NomActeur) {

   T_Pet_Creation_L     Message;
   T_Info               Reponse;
   Mbox*                UneMbox;

   cout << "comence New\n";
   Message.PosTablAct = NomActeur;
   Message.MaqPet = (int) QuelleMachine;
   TablCanal[QuelleMachine].Abrir(TablManijas[QuelleMachine]);
   if(TablCanal[QuelleMachine].Erreur != OK)
     cerr << "ERREUR: En New Abriendo Canal con BoîteLettres\n";
   TablCanal[QuelleMachine] << Message;
   if(TablCanal[QuelleMachine].Erreur != OK)
     cerr << "ERREUR: New T_Pet_Creation_L\n";
```

```
   TablCanal[QuelleMachine] >> Reponse;
   if( TablCanal[QuelleMachine].Erreur != OK)
     cerr << “ERREUR: Reponse new Acteur\n”;
   TablCanal[QuelleMachine].Fermer();
   UneMbox = new Mbox;
   UneMbox->ChangeMaquina (Reponse.Parametre.Cola.Maquina);
   UneMbox->ChangePosTabla (Reponse.Parametre.Cola.PosTabla);
   cout << “Fini New\n”;
   return *UneMbox;
}
```

Programme 4 : Fonction pour créer un nouvel acteur.

Parmi les problèmes techniques rencontres pour le dévelopement de LDBA, nous allons eu : la synchronisation, la communication, les interblocages (“deadlocks”) de processus et l’opération “become”.

- Le premier problème s’est présenté lors de l’initialisation du système et pendant la communication entre les acteurs. Nous avons déjà parlé de l’initialisation (fonction start : programme 2). Le problème de la communication a été plus difficile à resoudre étant donné que l’envoi de messages se fait de façon dynamique. Nous avons parlé d’un processus maître, qui est chargé de gérer le système. C’est lui qui reçoit toutes les demandes pour la création des acteurs et qui decide en quelle machine il faut le créer. Pour établir cette communication, dans chaque machine il existe un processus local chargé des communications locales et aussi des différentes requêtes. Au fur et à mesure qu’une requête est faite par un acteur, c’est le processus local P qui la reçoit. Ensuite, selon le type de la requête, le processus P est chargé de répondre ou de transmettre la demande au processus maître. Le processus maître doit donc lire sur plusieurs canaux simultanément sans jamais se bloquer. Pour éviter les interblocages dans le système de communication et de gestion de LDBA (Fonction vérification Cycle : programme 5).

```
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  Macro  VerificationCycle(Machine)
// ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
//  PROPOS : Faire la vérification pour éviter un interblocage pendant
//  la communication entre deux machines.
//  Pour la vérification de possibles interblocages, nous avons utilisé une
//  matrice avec toutes les connexions, qui existent actuelement. Si on
// ajoute la nouvelle connexion et il existe un cycle, alors il est possible
// qu’un interblocage se fasse.
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#define    VerificationCycle(Machine)    cout << ”VerificationCycle(” << Machine << ”)\n”;
                    Arc.Operation = R_ADIC_ARC;
                    Arc.source = (int) MachineActuelle;
                    Arco.Arrive = Machine;
                    CanalNoBloq.Abrir(ManijaNoBloq);
                    if(CanalNoBloq.Erreur != OK)
                              cerr << ”ERREUR:  pour ouvrir le Canal\n”;
                    cout << ”R_ADIC_ARC ”<< Arc.Source<< ’,’<< Arc.Arrive << ”\n”;
                    CanalNoBloq << Arc;
                    if(CanalNoBloq.Erreur != OK)
                              cerr << ”ERREUR: R_ADIC_ARC\n”;
                    CanalNoBloq >> IlyaBloq;
                    if((CanalNoBloq.Erreur != OK))
                               cerr << ”ERREUR: R_ADIC_ARC\n”;
                    if (IlyaBloq == IL_Y_A_UN_CYCLE) {
                              cout << ”IL_Y_A_UN_CYCLE\n”;
                    Arc.Operation = R_ELIM_ARC;
                    cout << ”R_ELIM_ARC ”<< Arc.Source<< ’,’<< Arc.Arrive << ”\n”;
                    CanalNoBloq << Arc;
                    if(CanalNoBloq.Erreur != OK)
```

```
                cerr << ”ERREUR: R_ELIM_ARC\n”;
        Arc.Operation = R_FERMER_CANAL_ADY;
        cout << ”R_FERMER_CANAL_ADY ” << ”\n”;
        CanalNoBloq << Arc;
        if(CanalNoBloq.Erreur != OK)
                cerr << ”ERREUR\n”;
        CanalNoBloq.Erreur = close(CanalNoBloq.Connexion);
        if(CanalNoBloq.Erreur != OK)
                cerr << ”ERREUR:  en ElimineCycle\n”;
        DéjàEnvoyée = Il_n’est_pas_possible_d’envoyer;
         if (Lista == VIENS_DE_RECEVOIR)
                Pendientes[Machine].AdittioneInfo(Info, sizeof (T_Envio_Serveur);
        return Ce_n’est_pas_possible_d’envoyer;
        }
```

Programme 5 : Fonction pour éviter les interblocages dans le système de communication de LDBA.

- Le deuxième problème a été la construction de la fonction “become” pour les acteurs sérialisés. Il fallait créer un autre acteur avec un comportement de remplacement différent. Cette création est faite avec un processus Unix de la même façon que pour tous les autres acteurs. Néanmoins, il existe une contrainte : dans ce cas, l’acteur créé doit être créé dans la même machine que l’acteur ancien (fonction “become” : programme 6). Ceci peut entraîner une surcharge dans une machine particulière.

```
void Acteur::become (void* UnActeur) {

  /* structures de données */
   BoîteLettresPet.ChangeTypeActeur (((Acteur *) UnActeur)->TypeActeur);
   ((Acteur *) UnActeur)->BoîteLettresPet = BoîteLettresPet;
   BoîteLettresPet.Act = UnActeur;
   signal (SIGCLD, SIG_IGN);
   InfoBecome.TypeActeur = BoîteLettresPet.DemeTypeActeur();
   InfoBecome.PosTablMbox = BoîteLettresPet.DemePosTable();
   InfoBecome.Change      = NO_CHANGE;
   if (pipe(Stop_Fils) == -1) {
        printf (”***** ERREUR:  Stop_Fils.\n”);
        exit (1);  }
   if (pipe(Stop_Metodo) == -1) {
        printf (”***** ERREUR:  Stop_Metodo.\n”);
        exit (1); }
   if (fork() == 0) {
       read (Stop_Fils[0],(char *) &continuer, sizeof (continuer));
        return; }
   TablBecome[QuelleMachine].Abrir(TablManijasBecome[QuelleMachine]);
   if(TablBecome[QuelleMachine].Erreur == ERREUR)
     cerr << ”ERREUR: En ’void Acteur::become(void * UnActeur)’ \n”;
   TablBecome[QuelleMachine] << InfoBecome;
   continuer = 1;
   write (Stop_Fils[1],(char *) &continuer, sizeof (continuer));
   do {
       TablBecome[QuelleMachine] >> NvaPet;
       if (NvaPet.TypeMessage == R_ORD_EJECUCION) {
           TablBecome[QuelleMachine].Fermer();
            PosTable = NvaPet.Acciones.OrdEjec.PosTablMbox;
            Processus=TablMbox[PosTable]->EjecuteMetodo
              (NvaPet.Acciones.OrdEjec.NomMetodo,  VIENE_DE_BECOME, Stop_Metodo[0]);
           TablBecome[QuelleMachine].Abrir(TablManijasBecome[QuelleMachine]);
           if(TablBecome[QuelleMachine].Erreur != OK)
              cerr << ”ERREUR: En ’void Acteur::become(void * UnActeur)’ \n”;
            InfoBecome.Change = CHANGE;
           TablBecome[QuelleMachine] << InfoBecome;
          TablCanal[QuelleMachine].Abrir(TablManijas[QuelleMachine]);
          if(TablCanal[QuelleMachine].Erreur == ERREUR)
```

```
            cerr << ”ERREUR: En ’void Acteur::become(void* UnActeur)’ \n”;
            ReponseEjec.Processus = Processus;
           ReponseEjec.Metodo = NvaPet.Acciones.OrdEjec.Metodo;
           ReponseEjec.MaqPet = NvaPet.Acciones.OrdEjec.MaqPet;
            ReponseEjec.TypeResp = VIENS_DE_BECOME;
             ReponseEjec.PosTablMbox = PosTable;
            TablCanal[QuelleMachine] << ReponseEjec;
           TablCanal[QuelleMachine].Fermer();
            continuer = 1;
           write (Stop_Metodo[1],(char*) &continuer, sizeof (continuer));
        }
    } while (NvaPet.TypeMessage != R_FINI_BECOME);
   TablBecome[QuelleMachine].Fermer();
    exit (0);
}
```

Programme 6 : La fonction become pour la création d‘un comportement de remplacement.

3.3. Les différentes mises en œuvre de l’algorithme parallèle de classification

Nous avons développé trois implémentations différentes pour l’algorithme parallèle de classification d’objets dans TROPES : la première mise en œuvre n’utilise aucun modèle parallèle de résolution de problèmes. La deuxième utilise le modèle “acteur” et la dernière utilise le modèle de “tableau noir”.

La première mise en œuvre est une implémentation à la “force” de l’algorithme parallèle. Dans cette implémentation le contrôle n’est pas séparé des mécanismes de raisonnement. Donc, une instance d’un processus (pour chacune des procédures il y a un processus différent) doit être chargée : d’abord de vérifier l’état des points de vue et des classes, puis de réaliser sa tâche spécifique (par exemple marquer une classe). Néanmoins, il est possible d’arriver à des incohérences dans l’état global du point de vue, car, il y a une zone de données partagée pour tous les processus: le concept. Par exemple,deux processus différents peuvent changer à la fois une même classe, car chaque processus est complètement indépendant de l’autre. Le contrôle est fait localement au niveau du processus.

Cette implémentation comporte plusieurs désavantages :

- le contrôle est très “obscur”, et les problèmes tels que la terminaison de la classification de l’objet sont très difficiles à détecter.
- il peut y avoir des incohérences lors de la modification simultanée d’une même structure de données.
- l’impossibilité de déterminer l’état global.
- le principal est la difficulté pour ajouter de nouvelles opérations sur la base de connaissances, tels que la classification de classes, le filtrage, etc.

Ces raisons ont conduit à modéliser l’algorithme parallèle d’une autre façon : en utilisant les modèles parallèles de résolution de problèmes.

Pour la deuxième mise en œuvre, nous avons utilisé le modèle “acteur”. Dans cette implémentation le contrôle est séparé des mécanismes de raisonnement. Néanmoins, il existe un problème dans cette mise en œuvre : les acteurs n’utilisent pas de mémoire commune. Ce problème rend difficile la synchronisation de tous les processus qui interagissent entre eux pour classifier l’objet dans un même point de vue. Car il peut y avoir des problèmes puisqu’un acteur est libre d’avoir un comportement quelconque avec très peu de relations par rapport au comportement actuel.

Les deux principaux désavantages de cette implémentation sont :

- l’état global de la classification en un instant donné est très difficile à déterminer.
- pendant la classification d’un objet dans un concept, le système fait des inférences (par utilisation des passerelles). Ces inférences sont difficiles à modéliser avec les acteurs étant donné le principe d’encapsulation de ces derniers.

Par exemple, si pendant la classification, une classe C/PVi devient "impossible" ou "sûre", il faut propager cette marque dans le point de vue PVi, et dans les autres points de vue à travers des passerelles. Dans le cas où le système arrive à un autre point de vue PVj (à une classe plus basse que la classe actuelle D/PVj), pour lequel il y a un acteur qui est en train de faire l'obtention d'information, ce dernier ne peut pas apercevoir l'inférence. Il faut lui envoyer un message prioritaire, mais il ne le traite qu'à la fin du traitement du message actuel.

La dernière mise en œuvre utilise le modèle de "tableau noir". Dans cette mise en œuvre le contrôle est séparé des mécanismes de raisonnement. L'analogie entre le modèle de connaissances TROPES et le modèle de "tableau noir" sera présentée dans le chapitre suivant et une description plus détaillée de la mise en œuvre sera présentée dans le chapitre 6.

3.4. Conclusions

Nous présentons d'abord les conclusions pour le modèle "tableau noir", puis pour le langage acteur et finalement les conclusions de la comparaison entre ces deux modèles.

Comme nous venons de le voir, plusieurs objectifs sont atteints par l'utilisation d'un modèle de "tableau noir":

- le modèle de "tableau noir" permet de faire coopérer plusieurs domaines de connaissances. Ces sources de connaissances sont indépendantes.
- le modèle de "tableau noir" permet d'exprimer de manière explicite des heuristiques de résolution. La solution est construite en exploitant de manière aussi opportuniste que possible les données et les connaissances.
- en introduisant un raisonnement sur plusieurs niveaux d'abstraction, le modèle de "tableau noir" permet de réduire l'espace de recherche et sa combinatoire. La solution est construite de manière incrémentale.
- enfin, le fonctionnement parallèle et asynchrone des sources de connaissances rend possible une implantation sur une architecture multiprocesseur.
- dans le modèle "acteur" la résolution d'un problème se fait par délégation des tâches d'un acteur à un autre acteur, tandis que pour le modèle de "tableau noir", elle se fait par coopération entre les différentes sources de connaissances.
- dans le modèle "acteur" la communication entre les acteurs se fait par messages, tandis que pour le modèle de "tableau noir" elle se fait à travers le tableau.

Finalement, nous avons développé trois mises en œuvre du modèle TROPES : la première, un algorithme développé sans aucun modèle parallèle de résolution de problèmes; la deuxième en utilisant le modèle acteur (LDBA) et la troisième en utilisant le modèle de "tableau noir" (DSBB). A partir de ces deux dernières réalisations, nous pouvons conclure :

- pour modéliser les différentes parties du processus de classification d'un objet dans TROPES, il est plus clair de le faire avec le système de "tableau noir".
- chaque acteur a sa propre mémoire privée. Cela pose un problème pour obtenir un état global de la classification en un instant donné.
- le fait que les sources de connaissances partagent l'information dans le "tableau noir" et que l'interaction entre les sources de connaissances ne se fasse qu'à travers le "tableau noir" entraîne un contrôle plus facile à gérer.
- le modèle "acteur" est asynchrone. Cet asynchronisme pose des problèmes pour la propagation de marques dans les différents points de vue.
- dans le modèle "acteur" la résolution d'un problème se fait par délégation des tâches d'un acteur à un autre acteur, tandis que pour le modèle de "tableau noir", elle se fait par coopération entre les différentes sources de connaissances. Dans le modèle "acteur" la communication entre les acteurs se fait par messages, tandis que pour le modèle de "tableau noir" elle se fait à travers le tableau. Ces différences influent beaucoup sur la conception des algorithmes de contrôle. Par exemple, l'état global de la classification en un instant donné est beaucoup plus facile à déterminer avec le modèle de "tableau noir" qu'avec le modèle "acteur".

- l'approche "tableau noir" s'avère un outil technologique de haut niveau bien adapté à la résolution des applications complexes car elle permet une représentation de la connaissance multi-modale, la manipulation de schémas de raisonnement multi-formes, ainsi qu'une résolution coopérative des problèmes.

Pour ces raisons nous avons choisi le modèle de "tableau noir" pour modéliser la parallélisation de l'algorithme de classification dans TROPES (§ 4 et §6).

4. ANALOGIE ENTRE LE MODELE TROPES ET LE MODELE DE TABLEAU NOIR

Dans ce chapitre est présentée une analogie entre le modèle TROPES lors d'une opération (par exemple la classification d'un objet dans la base de connaissances), et le modèle de tableau noir, sur lequel différentes sources de connaissances ou instances de ces sources font des modifications. Ici, le concept de modification est une généralisation du fait d'utiliser le tableau noir. Nous rappelons que le modèle de tableau noir permet d'appréhender la résolution d'un problème au travers d'un processus incrémental et opportuniste, et que ces sources sont chargées d'apporter leurs connaissances spécialisées sur le problème afin d'arriver à une solution [BOI 90].

Comme nous l'avons dit dans le chapitre sur la spécification de TROPES, une base de connaissances est composée d'un ensemble de concepts; chaque concept est composé d'un ensemble de points de vue, et ces derniers sont composés chacun d'un ensemble de classes qui forment un arbre. L'idée est donc d'envisager chaque concept comme un tableau noir et chaque point de vue comme un panneau, auxquels sont associées les sources de connaissances qui peuvent accéder au tableau de façon indépendante par rapport aux autres sources. Dans ce cas "l'accès à un objet" veut dire que la source de connaissances peut le voir et aussi le modifier. Néanmoins, étant donné que dans TROPES il existe des passerelles entre les différents points de vue, lorsqu'une source arrive à une passerelle et la modifie, cette modification a des répercussions sur l'état des autres sources.

Dans la suite, nous allons exposer les différents composants plus en détail.

4.1. Approche générale

Dans cette section, nous allons établir une analogie entre chaque composante du modèle TROPES et les composantes du modèle de tableau noir.

TROPES peut être envisagé comme un ensemble de données qui représentent la connaissance (parmi les éléments de cet ensemble se trouvent la base de connaissances, les concepts, les points de vue, les classes, les passerelles et les instances) dont le contrôle est séparé de la connaissance, et d'un ensemble des "opérations" : ces "opérations" sont faites sur ces connaissances. Le mot "opération" peut signifier une tâche complexe telle qu'un processus de raisonnement ou la classification d'un objet, ou bien une tâche simple telle qu'une requête. Un système de tableau noir est composé d'un tableau noir, des sources de connaissances et du contrôle. L'analogie proposée entre les différentes composantes de TROPES et les composantes du modèle de tableau noir est la suivante (figure 4.1) :

- entre les opérations sur la base de connaissances et les sources des connaissances. Il existe différentes sortes d'opérations : classification de classes, filtrage, classification des instances, l'exécution de tâches. Ces opérations peuvent être représentées chacune comme une source de connaissances.
- entre les éléments d'une base de connaissances et le tableau noir.
- entre le contrôle implicite des opérations de TROPES et le contrôle explicite des systèmes de tableau noir.

Analogie	TROPES	Le modèle de tableau noir
Données	La base de connaissances : les classes, les points de vue et les objets.	Le tableau
Résolution du problème à travers :	Les "opérations" : la classification, le filtrage, etc.	Les sources de connaissances
Contrôle	Implicite dans les "opérations"	Explicite : les sources de connaissances du contrôle et le tableau du contrôle.

Figure 4.1 : L'analogie entre les différentes composantes de TROPES et du modèle de tableau noir.

4.2. La base de connaissances de TROPES et le tableau

Un modèle puissant de représentation des connaissances doit avoir les éléments nécessaires pour que l'expert puisse exprimer sa connaissance des concepts, des règles, des relations, etc. Il doit aussi pouvoir utiliser efficacement l'information qui a été donnée et raisonner avec elle pour résoudre des problèmes et pour acquérir de nouvelles connaissances [MAR 89]. Finalement, les mécanismes d'exploitation du modèle doivent supporter et expliquer le raisonnement suivi pour résoudre une tâche et partager sa connaissance avec l'utilisateur [FIK 85].

Dans le modèle de tableau noir, le tableau est une structure de données globale. Il représente le "médium" général d'interaction entre les différents résolveurs au cours de la résolution d'un problème donné (d'après [BOI 90]). Le tableau contient les éléments de la solution. Celui-ci représente l'espace solution des problèmes.

Dans le modèle TROPES, la connaissance est séparée du contrôle, qui est implicite dans les différentes "opérations" sur la base. Cette séparation facilite la modification des connaissances et l'ajout de nouvelles informations à la base [MAR 89].

Le contrôle dans le modèle de tableau noir est également séparé de façon explicite des données, et comme nous l'avons dit, l'activité des sources de connaissances est dirigée par les événements. Chaque changement intervenant dans le tableau constitue un événement. Cet événement peut satisfaire la condition de déclenchement d'une source de connaissances. Il peut exister dans un tableau des éléments de la solution du problème et des éléments qui n'appartiennent qu'au contrôle.

Il est donc tentant de faire une analogie entre le tableau noir et la base de connaissances. Pour cela, nous reprenons les idées du modèle de tableau noir BB1 [HAY 84], dans lequel existent deux tableaux noirs : un de domaine et un de contrôle.

Le tableau noir de domaine enregistre les éléments de solution pour un problème, appelés événements. Il est structuré en niveaux hiérarchiques, qui représentent différents degrés d'abstraction ou d'analyse du problème. Aux niveaux supérieurs de la hiérarchie correspondent des objets plus abstraits que dans les niveaux inférieurs de la hiérarchie.

La structure du tableau de contrôle est aussi hiérarchique. Ce tableau permet d'organiser les événements relatifs à la résolution du problème de contrôle. Ces événements sont accessibles par toutes les sources de connaissances.

Nous ferons d'abord l'analogie entre le tableau noir de domaine et la base de connaissances du modèle TROPES. Ensuite, nous reviendrons sur le thème du tableau de contrôle dans la section 4.4.

4.2.1. Association entre le tableau noir et les bases de connaissances en TROPES.

Comme nous venons de le voir, il existe certaines similarités entre une base de connaissances et le tableau noir :

- Dans la base de connaissances, comme dans le tableau noir, on a la connaissance des experts sur un domaine. Ils représentent tous les deux l'espace solution d'un problème donné. Par exemple, l'espace de recherche pour la procédure de classification d'une instance est le concept [MAR 89].
- Les deux servent à leur façon comme un médium d'interaction pour les mécanismes correspondants de solution des problèmes : les mécanismes de raisonnement et les sources de connaissances, lors de la résolution d'un problème.
- Les deux représentent l'état courant de résolution du problème.
- Dans les deux cas, le contrôle est séparé de la connaissance. Dans la base de connaissances, ce contrôle est implicite dans les opérations qui se déroulent sur la base, et dans le modèle de tableau noir le contrôle est explicite (par exemple, le contrôle peut être représenté explicitement comme une source de connaissances).
- Dans le modèle de tableau noir, le contrôle est fondé sur les événements. Ces événements représentent les changements produits sur le tableau noir comme conséquence de l'exécution d'une source de connaissances et dont le but est d'arriver de façon incrémentale à une solution du problème. Dans TROPES, les opérations sont "dirigées" par la connaissance.

Il existe aussi une similarité importante entre les deux modèles: les niveaux hiérarchiques. Dans le modèle du tableau noir, le tableau est organisé de façon hiérarchique, et les niveaux représentent différents degrés d'abstraction ou d'analyse du problème. Dans TROPES, la connaissance est aussi organisée de façon hiérarchique, e les classes sont reliés par des liens de spécialisation. Il est donc possible d'envisager une analogie entre les niveaux du tableau et les niveaux du TROPES. Dans les deux modèles, au fur et à mesure que l'on descend dans la hiérarchie, on s'approche de la solution "finale" du problème. Nous reviendrons sur ce sujet dans la section 4.2.3.

Une base de connaissances dans TROPES est composée d'au moins un concept, composé lui même de plusieurs points de vue. Au niveau d'un concept, un objet est identifié de façon unique par la valeur de sa clé. L'objet doit aussi satisfaire toutes les contraintes des attributs de ses classes d'appartenance dans les différents points de vue du concept. Ces concepts sont complètement indépendants et déterminent une partition de l'univers du discours. Nous pouvons donc faire l'analogie entre le tableau noir et la base de connaissances (figure 4.2). Les concepts de cette base de connaissances et leurs propriétés représentent l'espace solution pour un problème donné.

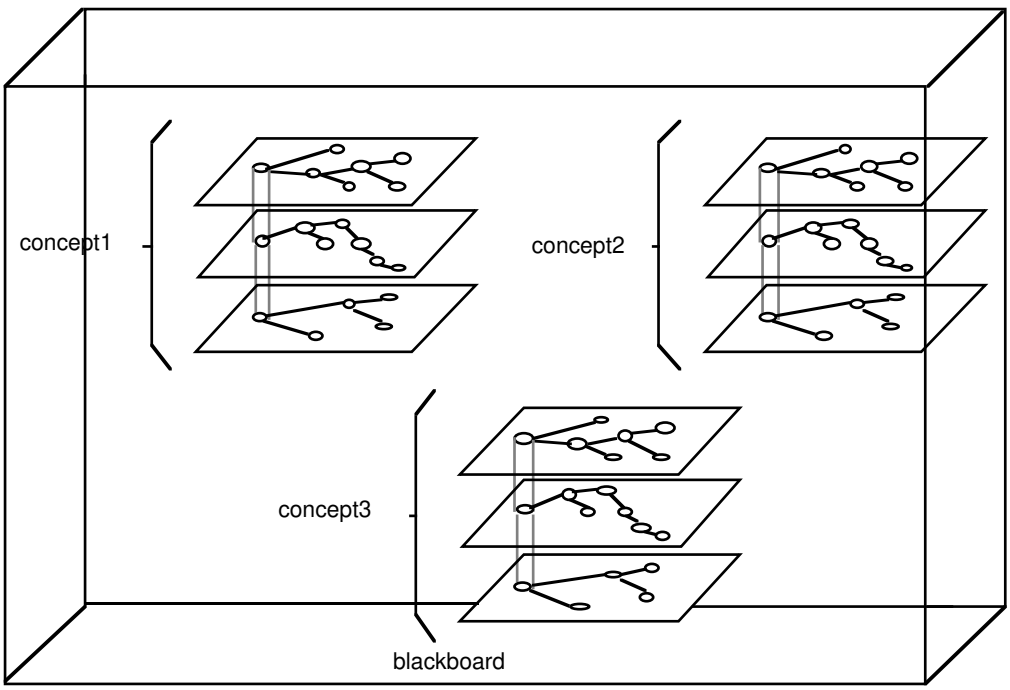


Figure 4.2 : Une base de connaissances composée de trois concepts indépendants forme le tableau.

Enfin, comme nous avons dit, un tableau peut avoir plusieurs panneaux, où chaque panneau peut être considéré lui même comme un tableau. Il nous semble possible d'associer chaque panneau à un concept de la base de connaissance. C'est-à-dire que nous allons diviser l'espace de solution en sous-espaces indépendants. Ces sous-espaces ou panneaux ne sont que les différents concepts. Les concepts, d'une base de connaissances de TROPES, sont complètement indépendants et déterminent une partition de l'univers du discours. La raison de considérer chaque concept comme un sous-espace indépendant pour les différentes opérations sur la base de connaissances, est simple: il n'existe qu'une sorte de liens entre les concepts. Ces liens sont les objets composés, qui ne sont pas pris en compte dans ce travail. Par la suite, lorsque nous parlons d'un tableau noir, nous ne faisons référence qu'à un concept (figure 4.3). Donc, l'analogie entre une base de connaissance composée de plusieurs concepts et le tableau noir n'est plus traitée.

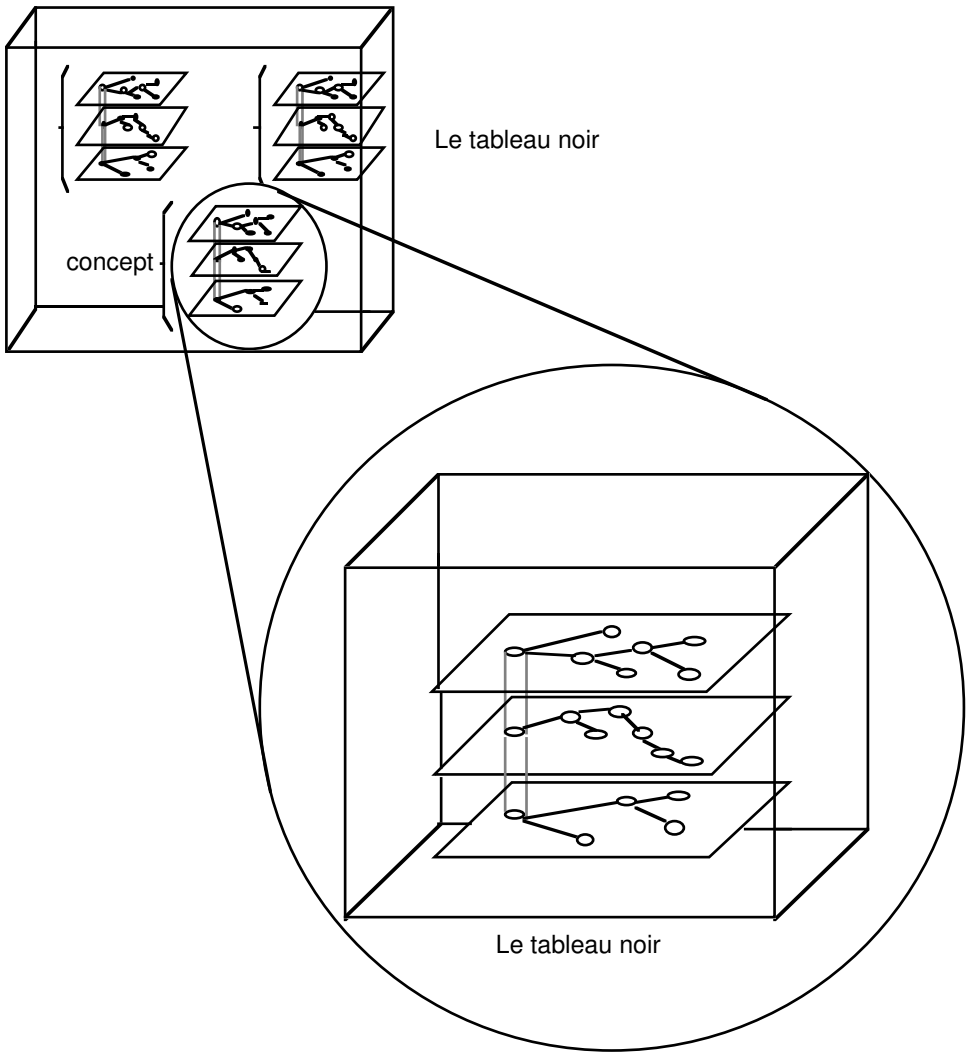


Figure 4.3 : Pour notre travail, la base de connaissances n'est composée que d'un concept. Donc, pour la suite un tableau noir représente un concept.

Dans les sections suivantes nous traitons des différentes composantes d'une base de connaissances dans TROPES.

4.2.2. Les concepts et le tableau noir

Nous venons de montrer qu'il existe une analogie entre un concept de TROPES et le tableau noir dans les modèles de tableau noir (figure 4.4). Ce tableau noir forme donc l'espace solution pour les différentes opérations réalisées sur la base de connaissances.

Un concept est formé par un ensemble de points de vue. Chacun de ces points de vue forme un arbre de spécialisation de classes. Chacun de ces arbres représente des visions différentes d'un même objet : les points de vue.

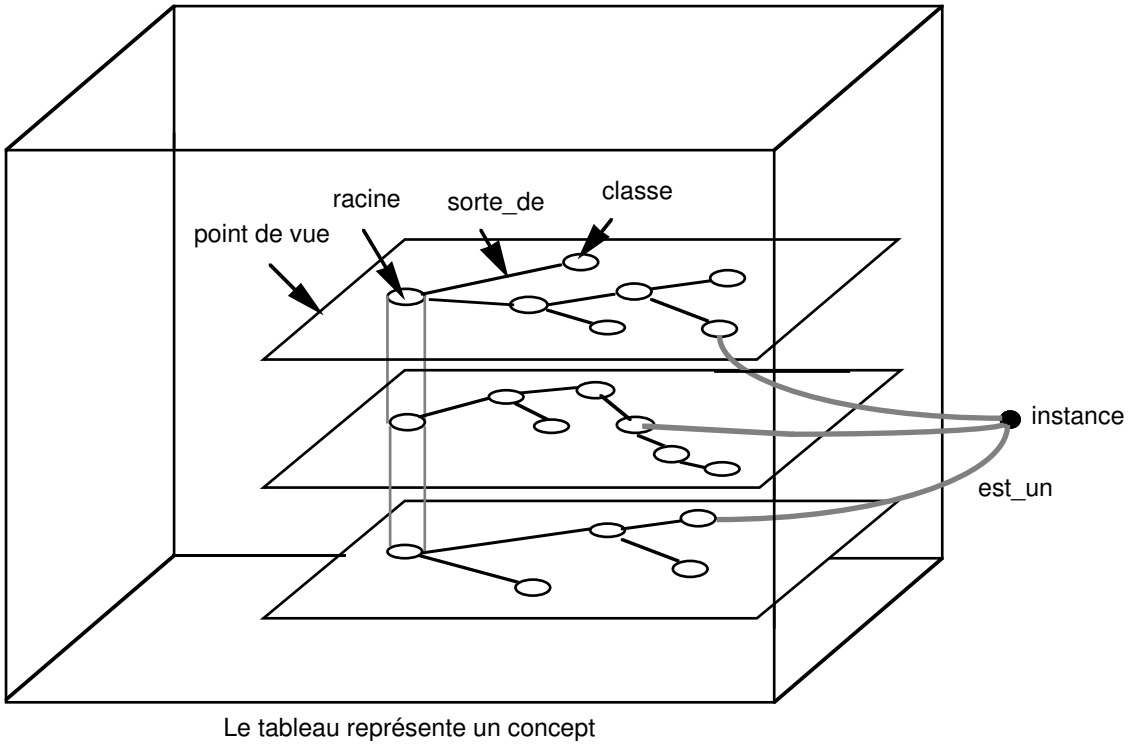


Figure 4.4 : Exemple d'un concept composé de trois points de vue indépendants. Ce concept est l'espace de recherche pour les différentes opérations sur la base de connaissances.

Un panneau du tableau noir peut être considéré lui-même comme un tableau noir. De cette manière nous pouvons décomposer le tableau en plusieurs panneaux. Il est donc possible d'envisager une analogie entre un point de vue du concept et un panneau du tableau noir, de la même façon que nous l'avons fait pour une base de connaissances et un tableau noir (figure 4.5).
Etant donné que la base de connaissances n'est composée que d'un concept, par la suite, l'analogie considérée est celle d'un concept envisagé comme le tableau noir, et les points de vue envisagés comme les panneaux du tableau noir.

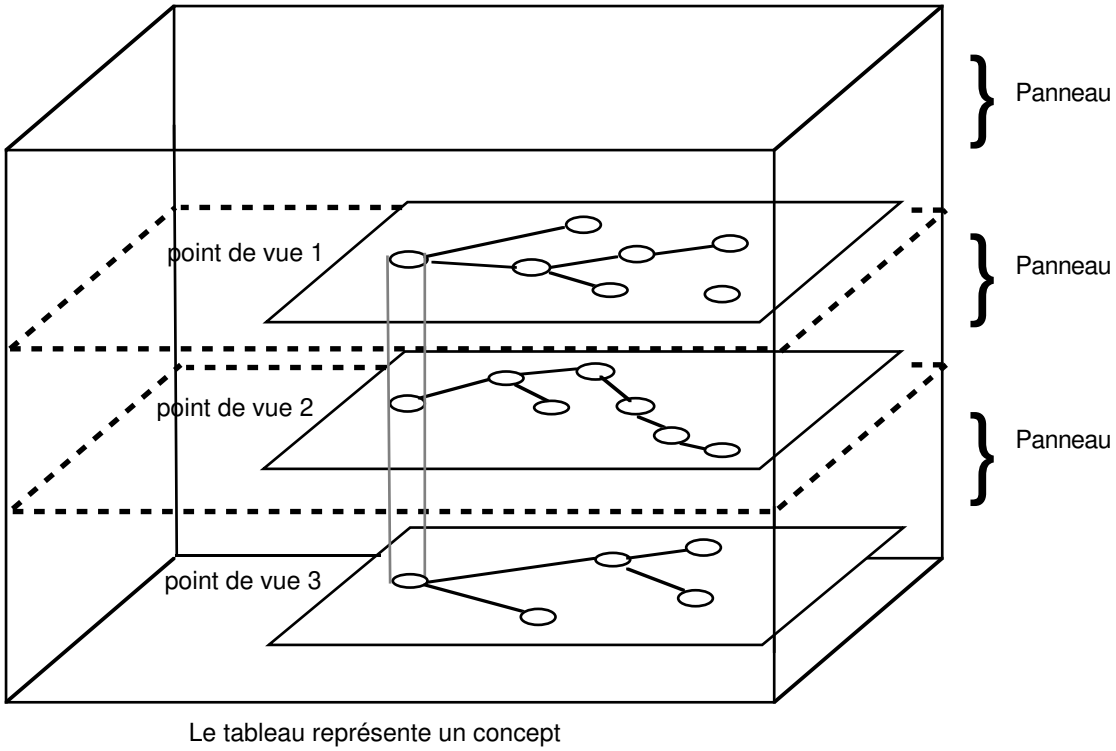


Figure 4.5 : Chaque point de vue peut être vu comme un panneau du tableau noir

4.2.3. Les points de vue, les passerelles et le tableau noir

Dans TROPES, les objets qui appartiennent à un concept peuvent être vus selon différents points de vue. Un point de vue peut être défini en général comme une position conceptuelle de laquelle un observateur regarde un objet. Cette approche part de l'hypothèse que le monde est unique, fait qui entraîne que l'union de toutes les perspectives d'un objet produit une représentation cohérente de l'objet. Ainsi, un concept est composé d'un ensemble de points de vue.

D'autre part, étant donné qu'un objet est rattaché à une classe dans chaque point de vue du concept, une opération sur ce concept doit être réalisée sur l'intégralité des points de vue du concept. Par exemple, lors d'une requête pour une instance, étant donné que l'instance est rattachée à tous les points de vue, si nous voulons savoir les contraintes des différents attributs, il faut regarder dans tous les points de vue.

Il existe deux possibilités dans le modèle TROPES lors du déroulement d'une opération sur un point de vue PVi :

- Il existe au moins une passerelle entre PVi et PVj, donc un changement sur le point de vue PVi peut entraîner un changement sur l'autre point de vue PVj.
- Il n'existe aucune passerelle entre les points de vue, donc le déroulement de l'opération dans chaque point de vue est complètement indépendant du déroulement dans l'autre point de vue.

Le déroulement d'une opération sur un point de vue peut entraîner des changements dans l'état des opérations sur l'autre point de vue. Par exemple, lors de la classification d'un objet, si celui-ci appartient à une classe E dans un point de vue PVi, et si la classe est une passerelle entre PVi et PVj, alors l'objet appartient aussi à la même classe dans le point de vue PVj. Dans ce cas, la classification de l'objet dans PVj va être dirigée vers la classe E.

Si nous voulons faire une analogie entre TROPES et les modèles de tableau noir, nous devons prendre en compte le fait que dans le modèle de tableau noir, la communication entre les sources de connaissances ne se fait qu'à travers le tableau. Donc, nous devons arriver à que les opérations sur la base de connaissances se réalisent de façon indépendante dans les différents points de vue.

Pour cela, il suffit que le contexte pour une opération P ne soit qu'un point de vue PVi. Donc, dans le cas où nous arrivons à une passerelle, et qu'un changement s'est produit, et si ce

changement entraîne des modifications dans l'état de l'autre point de vue PVj, alors les modifications ne seront pas faites directement sur ces points de vue pour l'opération P; ces modifications seront faites par une autre opération Q dont le contexte est PVj (figure 4.6). Cette opération peut être une instance de l'opération originale P.

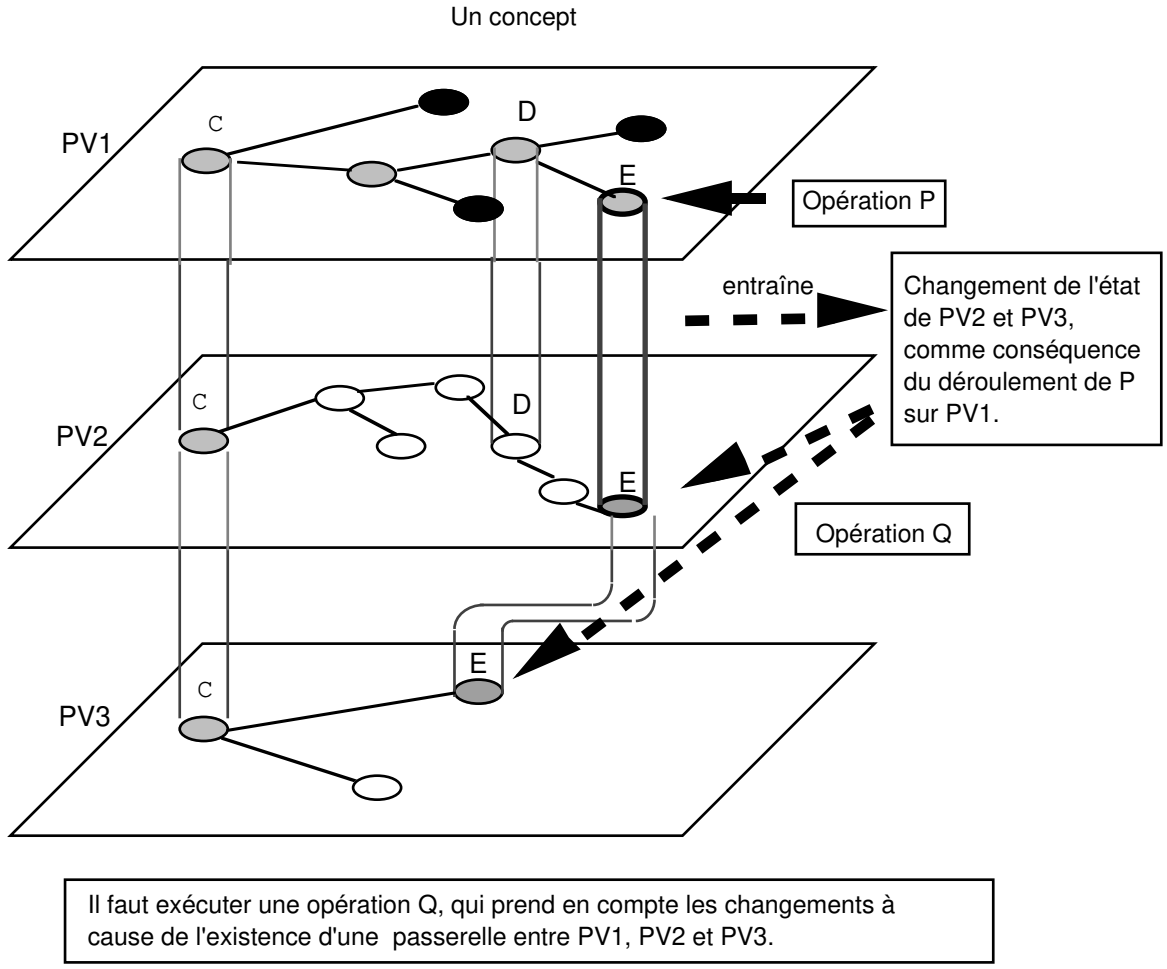


Figure 4.6 : Opérations dans TROPES à travers des passerelles.

Ces opérations sont donc faites de la même façon pour tous les points de vue du concept. En général, lors d'une opération sur un concept, la façon de raisonner ou de faire des inférences est la même puisque les points de vue ne sont que des perspectives différentes d'un même objet. Autrement dit, l'algorithme qui s'applique aux différents points de vue est le même. Ceci est la conséquence de la séparation du contrôle des mécanismes de raisonnement dans les opérations. Ce contrôle devient donc explicite, et il sera considéré dans la section du contrôle.
Enfin, nous pouvons conclure qu'il est possible d'envisager chaque point de vue comme un panneau indépendant sur lequel se réalisent les différentes opérations. Dans ce cas, un panneau peut être considéré comme un tableau noir. Donc le tableau noir initial, qui représentait un concept de la base de connaissances peut être éclaté en plusieurs tableaux noirs : les panneaux, où chacun d'eux représente un point de vue (figure 4.7).
Pour résumer, dans les systèmes de tableau noir les sources de connaissances sont indépendantes et séparées. Dans TROPES, les mécanismes de raisonnement sont séparés, et ils peuvent être considérés comme indépendants. Nous avons vu qu'un changement dans un point de vue peut produire des changements dans les autres points de vue à cause des passerelles. Dans ce cas, le mécanisme de raisonnement qui se déroule sur le deuxième point de vue est affecté. Les communications nécessaires entre les différents mécanismes de raisonnement se font à travers le tableau.

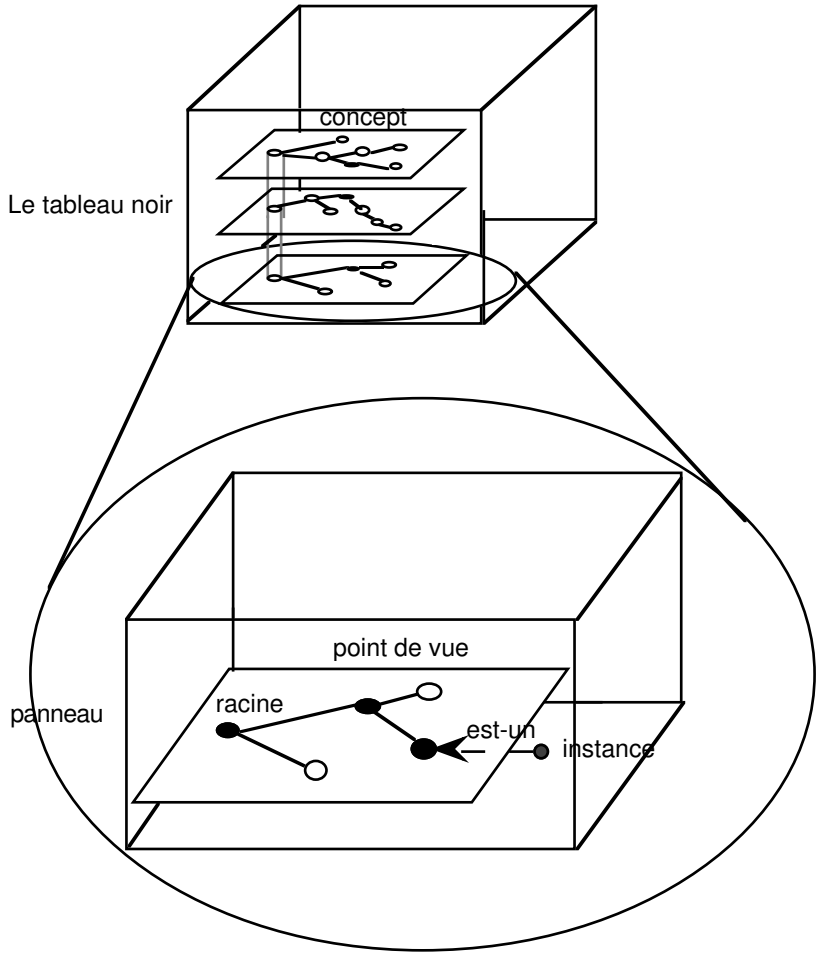


Figure 4.7 : Le tableau noir, qui représente un concept, est composé d'au moins un point de vue. Ces points de vue peuvent être envisagés comme des panneaux du tableau noir.

Un point de vue est un ensemble de classes, où ces classes sont organisées en une structure hiérarchique. La structure de cette hiérarchie est un arbre, car à chaque niveau de la structure il n'y a que des classes disjointes (voir la définition de TROPES). Or, étant donné que la structure d'un point de vue est un arbre, celui-ci a une classe racine. Cette classe racine est la classe la plus générale décrivant l'ensemble complet des objets qui appartiennent au point de vue et aussi au concept.
Les niveaux de la structure hiérarchique du point de vue représentent les différents degrés de spécialisation des classes. Au fur et à mesure que l'on descend dans l'arbre des classes, la connaissance sur les objets s'enrichit: les niveaux supérieurs contiennent des objets plus abstraits, dans le sens où nous avons moins d'information sur eux que dans les niveaux inférieurs. Nous avons vu que ces niveaux représentent des aspects similaires dans les deux modèles : TROPES et le modèle de tableau noir. Dans le modèle de tableau noir les hiérarchies représentent différents degrés d'analyse du problème, tandis que dans TROPES, les hiérarchies représentent un enrichissement de la connaissances sur l'univers du discours. Dans les deux modèles, les niveaux supérieurs représentent d'une certaine façon moins de "connaissance" sur le problème et les niveaux inférieurs plus de "connaissances". La descente entraîne donc dans ces modèles un enrichissement de la "connaissance" : dans TROPES, les sous-classes d'une classe ont plus "d'informations" que la sur-classe et si on arrive à une classe feuille, on a trouvé la "solution finale" au problème de la classification de l'objet . Pour le tableau noir la descente entraîne que la connaissance sur la solution du problème augmente et s'approche de la solution finale du problème.

4.2.4. Les objets et le tableau noir

Un objet I est un individu d'un concept, identifié de façon unique par la valeur de sa clé. Un objet a une classe d'appartenance dans chaque point de vue. Cette objet doit donc être rattaché à

l'intégralité des points de vue du concept à travers ses classes d'appartenance. Lorsqu'un objet appartient à une classe, il satisfait toutes les contraintes des attributs de cette classe (figure 4.8).

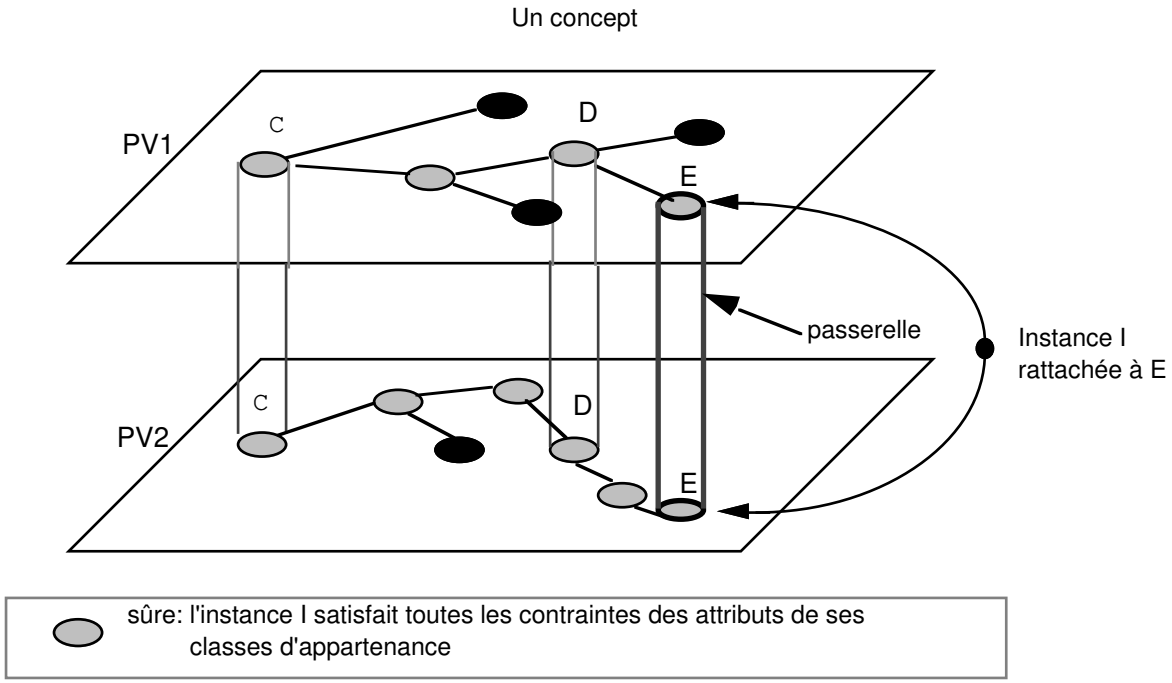


Figure 4.8 : L'objet I est rattaché à la classe E dans les points de vue PV1 et PV2.

Les objets sont donc composés d'un ensemble de valeurs pour les différents attributs de leurs classes d'appartenance. Finalement, toutes les objets font partie du tableau noir.
D'autre part, l'ensemble des objets qui appartiennent à la base de connaissances est l'union des ensembles d'objets rattachés aux différents points de vue. Cet ensemble peut être organisé de façon hiérarchique, donc lors d'une opération sur la base dans laquelle il faut faire un parcours de l'ensemble, ce parcours peut aussi être fait en parallèle. Il existe plusieurs méthodes pour le faire, et ces méthodes dépendent de la structure, et de l'opération qui est faite (Annexe 2).

4.3. Les sources de connaissances et les opérations de TROPES

Dans les bases de connaissances déclaratives, la connaissance est séparée du contrôle. Le raisonnement est entièrement dirigé par les données; il utilise des mécanismes d'inférence qui permettent la solution des problèmes. Donc, la plupart des opérations faites sur les bases de connaissances réagissent aux modifications sur les données. Or, le contrôle est implicite dans les opérations.
Dans les systèmes de tableau noir, la connaissance sur la façon de résoudre un problème est groupée autour des sources de connaissances et séparée du contrôle. Ces connaissances sont représentées de façon déclarative comme un ensemble de procédures, un ensemble de règles ou d'assertions logiques. Il existe aussi une précondition qui déclenche la source de connaissances.
Nous pouvons donc envisager une analogie entre les sources de connaissances des systèmes de tableau noir et les opérations faites sur les bases de connaissances. Dans ce travail, nous ne présentons que les opérations concernant la classification d'objets dans TROPES. Il est possible de faire la même analogie entre les sources de connaissances et les autres opérations qui existent dans TROPES. Dans la suite, nous développerons cette idée.
Nous reprenons encore les idées du modèle BB1 [HAY 84], lequel fait la distinction entre les sources de connaissances du domaine et les sources de connaissances du contrôle. Toutes ont la même structure: la condition d'une source est composée d'un déclencheur qui teste sa pertinence et d'une précondition qui teste sa faisabilité. Quand le déclencheur, un ensemble de prédicats basés sur l'événement, est vrai, la source de connaissances est déclenchée. Quand la

précondition, un ensemble de prédicats basés sur l'état du tableau noir, est vraie la source de connaissances est activable.
Maintenant, nous faisons un parallèle entre le modèle BB1 et TROPES :

- Une opération sur la base de connaissances est composée du mécanisme de raisonnement, et du contrôle qui est implicite dans l'opération. Donc, si nous séparons ce contrôle, et si celui-ci devient explicite, nous aurons deux parties pour chaque opération: le raisonnement proprement dit, et le contrôle qui est devenu explicite (figure 4.9). Par exemple, lors de la classification d'une instance, nous pouvons séparer le choix d'un point de vue des autres parties de l'algorithme: obtention d'information, appariement, propagation de marques et actualiser l'information. Le raisonnement utilise des mécanismes d'inférence qui permettent la solution des problèmes. Ces mécanismes de raisonnement sont les résolveurs des problèmes. Ils sont les agents compétents dans leur domaine.

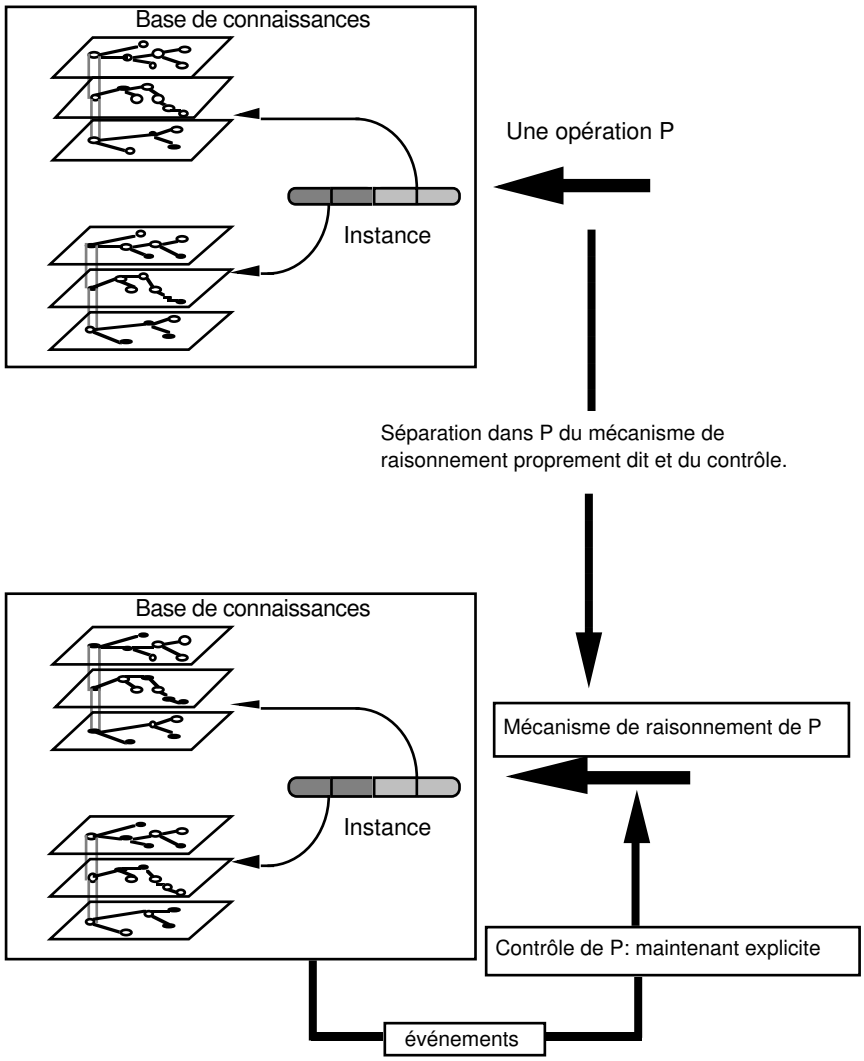


Figure 4.9 : Une opération sur la base de connaissances peut être divisée en deux parties:le raisonnement proprement dit, et le contrôle.

Nous pouvons également faire une analogie entre les sources de connaissances des modèles de tableau noir, et les opérations de TROPES, dont les mécanismes de raisonnement ont été séparés du contrôle.

- Comme nous l'avons dit, les opérations réagissent aux modifications sur les données, c'est-à-dire que le raisonnement est entièrement dirigé par les données. Ces modifications

sur les données peuvent être envisagées comme les préconditions pour le déclenchement des sources de connaissances dans les modèles de tableau noir.

Enfin, pour résumer, nous ferons l'analogie entre les différentes caractéristiques et composantes des deux modèles. Pour cela, nous reprenons la caractérisation des sources de connaissances donnée par [BOI 90].

- Dans les systèmes de tableau noir, les sources de connaissances sont indépendantes et séparées. Dans TROPES, les mécanismes de raisonnement sont séparés, et ils peuvent être considérés comme indépendants.
- Les sources de connaissances sont "conscientes" de leur rôle, des éléments qui les intéressent et de ceux qu'elles peuvent apporter dans la résolution. Cette "conscience" dans un mécanisme de raisonnement part de sa "définition" : dans les bases de connaissances déclaratives, le raisonnement est entièrement dirigé par les données; il utilise des mécanismes d'inférences qui permettent la solution des problèmes pour lesquels il n'existe pas de procédures explicites [MAR 89].
- Les sources de connaissances sont les seuls éléments intervenant sur le tableau noir. Dans TROPES, les seuls éléments qui interviennent sur la base de connaissances sont les opérations. Après la séparation du raisonnement proprement dit du contrôle, les seuls intervenant sur la base de connaissances sont les mécanismes de raisonnement.
- Les sources de connaissances peuvent soit modifier le tableau noir, soit effectuer des entrées-sorties. Dans TROPES, les mécanismes de raisonnement peuvent bien-sûr modifier la base de connaissances, par exemple à travers des inférences en ajoutant des nouvelles connaissances, etc. D'autre part, les entrées-sorties se présentent lors d'une interaction avec l'utilisateur de la base de connaissances.
- Les sources de connaissances coopèrent pour le partage d'informations. Dans une base de connaissances il est évident que les différents mécanismes de raisonnement coopèrent pour le partage d'informations : la connaissance. Il faut noter que le degré de coopération entre les sources de connaissances, qui représentent les opérations sur la base de connaissances, est certainement différent de celui entre les sources de connaissances dans les modèles de tableau noir.

  Il est aussi certain que dans les modèles de tableau noir chaque source de connaissances travaille sur un niveau différent du tableau noir. Donc, la coopération entre les différentes sources consiste plutôt à guider la solution incrémentale du problème à travers différents niveaux, afin de trouver une solution cohérente.

  Par contre, dans les points de vue, les opérations travaillent normalement sur l'intégralité des points de vue. Ici, nous pouvons dire que l'analogie peut être vue comme la façon dont coopèrent les différentes opérations. Par exemple pendant la classification, les différentes procédures (obtention d'information, appariement, propagation de marques, etc) partagent l'information pour descendre l'objet dans les classes inférieures de la hiérarchie de classes.
- Les sources de connaissances sont constituées de deux parties: la condition du déclenchement et leur corps. Le corps peut être vu comme l'algorithme du mécanisme de raisonnement, et la condition de déclenchement comme les modifications sur les classes ou points de vue lors d'un raisonnement. Par exemple, si une classe devient "impossible", la procédure de propagation de marques est déclenchée. Ce raisonnement peut déclencher un autre raisonnement, qui peut être différent ou non. Cela dépend de l'état global de la base de connaissances. Par exemple, si on arrive à une passerelle pendant la propagation de marques, il faut propager la marque dans l'autre point de vue de la passerelle. Donc, la condition de déclenchement dans le cas de TROPES est constituée de deux parties: une partie qui teste la pertinence du raisonnement et une autre partie qui teste la faisabilité du raisonnement.

Pour finir, l'analogie entre les sources de connaissances et les mécanismes de raisonnement peut être vu dans la figure 4.10.

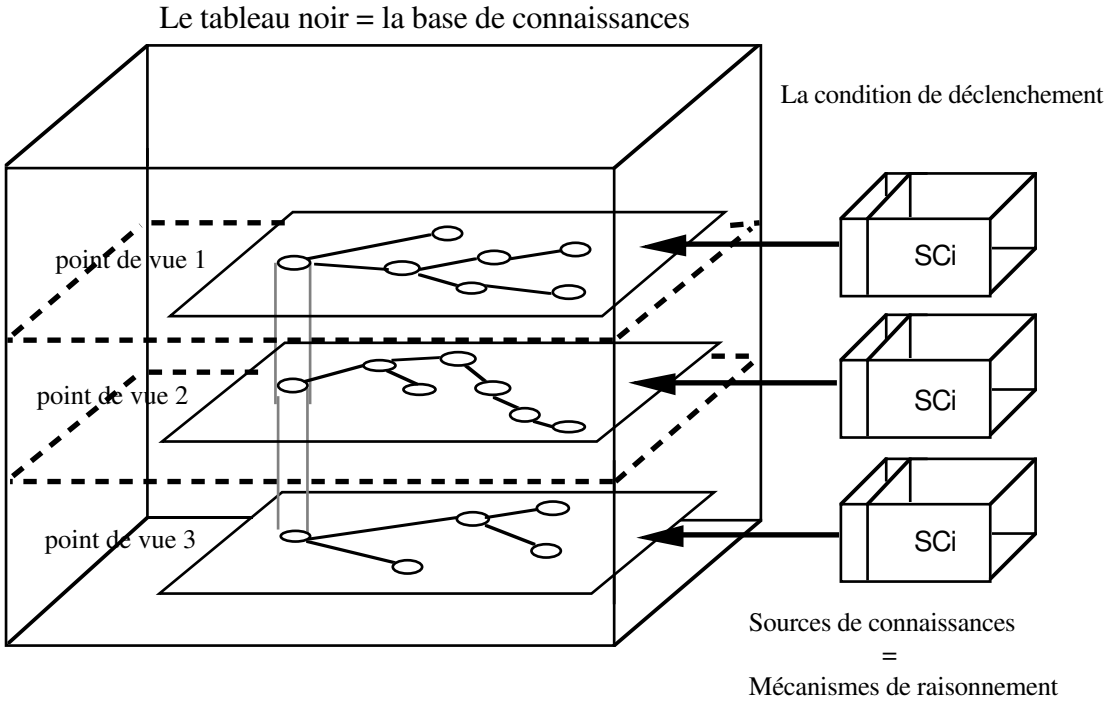


Figure 4.10 : Chaque source de connaissances travaille sur un panneau. Une source de connaissances représente une opération sur une base de connaissances et le panneau représente un point de vue. Ces sources peuvent être simplement instances de la même source ou de sources différentes.

Nous avons dit que les opérations faites sur les différents points de vue sont les mêmes lors d'une opération sur la base de connaissances. Donc, les sources de connaissances qui travaillent sur les différents panneaux sont instances de la même source. Cependant, en regardant plus en détail les sources de connaissances, elles peuvent être décomposées en sources plus petites, qui font des opérations plus spécialisées. Dans ce cas, il est possible que les sources actives dans les différents panneaux soient différentes. Nous reviendrons sur ce sujet dans la section sur l'algorithme de classification.

4.4. Le contrôle et TROPES

Le problème de savoir s'il faut avoir un contrôle ou non, et s'il existe, si ce contrôle peut être centralisé ou non, dépend exclusivement des différentes opérations, en supposant que nous allons paralléliser les différentes opérations. Par exemple, si nous voulons connaître les classes auxquelles est rattaché un objet dans les différents points de vue, il n'est pas nécessaire d'avoir un contrôle centralisé et peut-être même pas de contrôle du tout. Mais, pour des autres opérations comme la classification d'un objet dans un concept, il faut disposer d'un contrôle, si nous voulons arriver à un résultat cohérent. Dans ce dernier cas, le contrôle centralisé nécessaire est minimum par rapport au contrôle qui peut être distribué à travers les différents processeurs.

Le contrôle va être constitué de deux parties : le contrôle du système et le contrôle de dialogue avec l'utilisateur, qui a été exprimé dans le chapitre sur la classification en parallèle dans TROPES (§ 2).

Le contrôle du système est constitué d'un tableau noir de contrôle et d'un ensemble de sources de connaissances qui planifient la résolution du problème, et dont les éléments peuvent être :

- Des éléments de contrôle, par exemple pour choisir un point de vue comme actif.
- Des objets qui servent à la communication entre les différentes sources lors d'une opération sur la base de connaissance, car dans les systèmes de tableau noir, les différentes sources de connaissances ne peuvent communiquer qu'à travers le tableau; cette communication est nécessaire pour arriver à une solution cohérente, lors d'une opération sur la base de connaissances.

Le contrôle va être séparé en deux parties: contrôle de bas niveau et contrôle de haut niveau. Nous présenterons d'abord le tableau noir de contrôle, puis le contrôle de bas niveau, et finalement le contrôle de haut niveau.

- le tableau noir de contrôle est composé de six niveaux :
    - le niveau problème : A ce niveau est exprimé l'énoncé du problème et des contraintes explicites liées à sa résolution. Sont énoncées aussi toutes les différentes opérations qui peuvent être réalisées sur la base de connaissances, par exemple, la classification d'un objet dans une base de connaissances donnée.
    - le niveau stratégie : ce niveau contient les stratégies de résolution générales. Par exemple, pendant la classification, pour choisir le point de vue, il existe plusieurs stratégies.
    - le niveau focus : ce niveau contient une technique de résolution plus détaillée . Il permet d'établir le séquencement des sources de connaissances.
    - le niveau heuristique : ce niveau contient des événements qui permettent de déterminer le critère le plus pertinent pour classer les sources de connaissances activables. Dans la description de l'algorithme de classification, nous reviendrons sur ce sujet.
    - le niveau agenda : ce niveau répertorie toutes les actions faisables du système, les sources de connaissances déclenchables et les sources de connaissances activables.
    - le niveau action : ce dernier niveau contient toutes les décisions d'action qui ont été prises.
- Le contrôle de bas niveau est de certaine façon indépendant de l'opération. Il doit faire la mise-à-jour du tableau avec les modifications faites pour les sources de connaissances, etc. Pendant la classification, par exemple, au fur et à mesure que l'algorithme de classification descend dans la hiérarchie de classes, il peut marquer les différentes classes comme impossibles, possibles ou sûres. Il faut donc mettre à jour le tableau noir global: la base de connaissances. Dans cette mise à jour, au fur et à mesure que le tableau noir trouve des passerelles, celles-ci sont utilisées pour propager l'information dans les autres points de vue. Ceci correspond à l'opportunisme dans les systèmes de tableau noir. C'est-à-dire, ce qui permet d'exprimer explicitement des heuristiques de résolution. Les heuristiques de contrôle s'adaptent dynamiquement à l'état courant de la solution. Par exemple, si une classe C/PVi devient "impossible" et il existe une passerelle de C/PVi vers D/PVj, l'opportunisme consiste donc à utiliser cette passerelle pour propager la marque "impossible" dans le point de vue PVj.
- Le contrôle de haut-niveau est chargé de planifier la résolution du problème. Par exemple, lors de la classification il faut choisir un point de vue. Il prend des décisions explicites pour résoudre le problème de contrôle: il décide quelles actions effectuer et quand les effectuer, par exemple, le choix d'un point de vue.

Dans la (figure 4.11), nous pouvons visualiser l'analogie complète entre le modèle TROPES et les modèles de tableau noir.

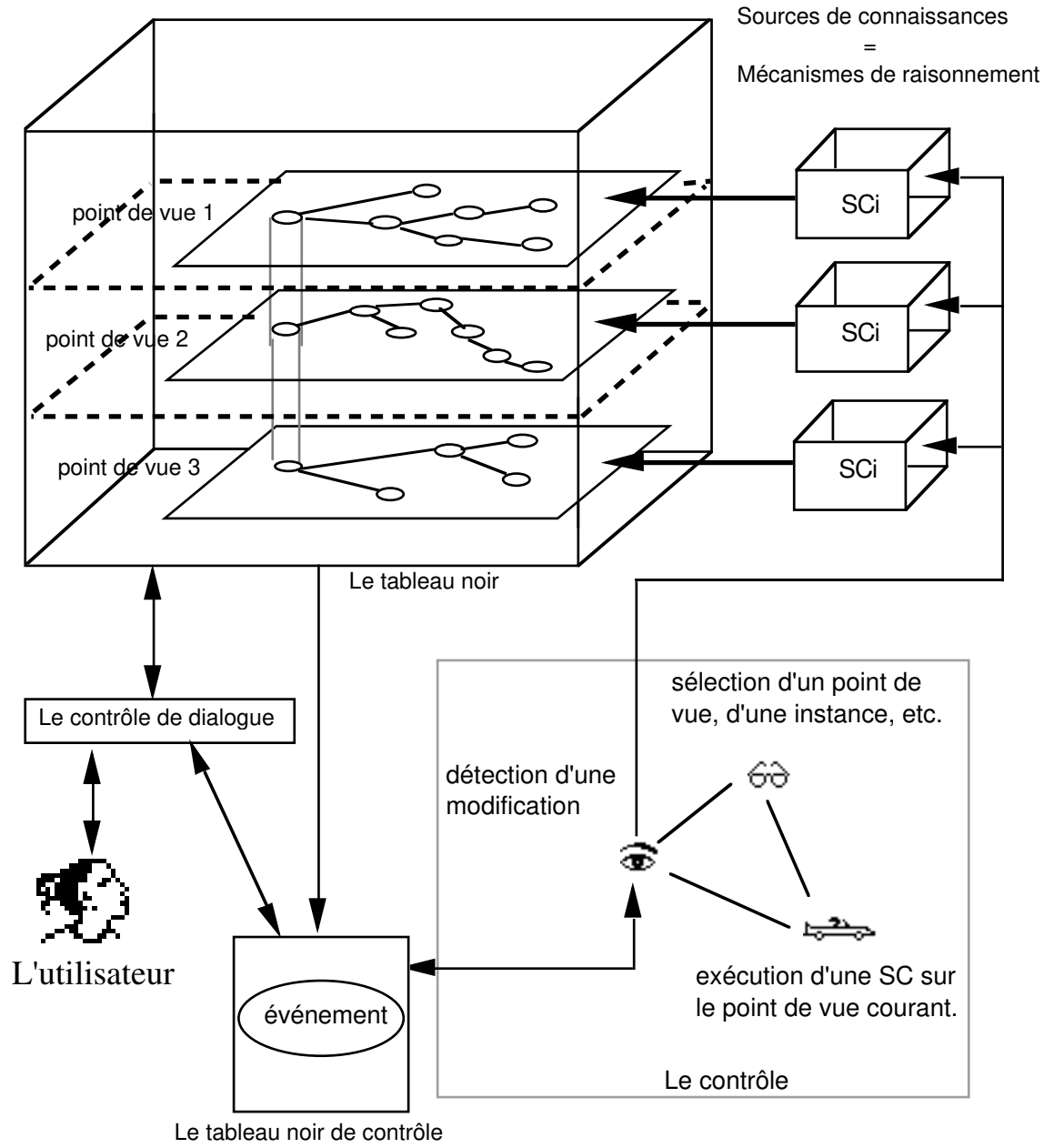


Figure 4.11 : Analogies entre TROPES et le modèle de tableau noir.

Nous venons de dire que les opérations sur la base de connaissances réagissent aux modifications sur les données. Ceci correspond exactement à l'étape de détection des systèmes de tableau noir. Après une modification sur les données, il faut choisir un point de vue, une classe et une instance d'une source de connaissances. Ceci correspond exactement à l'étape de sélection. Par exemple, lors de la propagation de marques, il faut choisir une classe C d'un point de vue PVi à partir de laquelle, la procédure de propagation va commencer à propager. Enfin, après avoir sélectionné le point de vue, la classe et l'instance de la source, il faut exécuter cette dernière sur la classe et le point de vue sélectionnés. Ceci correspond aussi à une étape du contrôle dans le tableau noir : l'exécution.

4.5. Conclusions

Dans ce chapitre a été présentée une analogie entre le modèle de tableau noir et le modèle TROPES. Après cette analogie, il est clair qu'il est possible de modéliser TROPES en utilisant le modèle du tableau noir. Cette modélisation nous permettra de mettre en œuvre la classification en parallèle de manière naturelle, car le parallélisme est une caractéristique inhérente du modèle de tableau noir.

Parmi les analogies faites dans ce chapitre, nous pouvons citer :

- Les deux servent à leur façon comme un médium d'interaction pour les mécanismes de raisonnement et les sources de connaissances, mécanismes dédiés à la résolution d'un problème.
- Les deux représentent l'état courant de résolution du problème.
- Dans les deux cas le contrôle est séparé de la connaissance. Dans la base de connaissances, ce contrôle est implicite dans les opérations qui se déroulent sur la base, et dans le modèle de tableau noir le contrôle est explicite (par exemple, le contrôle peut être représenté explicitement comme une source de connaissances sur un autre tableau).
- Dans le modèle de tableau noir, le contrôle est fondé sur les événements. Dans TROPES, les opérations sont "dirigées" par la connaissance.
- Dans les systèmes de tableau noir, les sources de connaissances sont indépendantes et séparées entre elles. Dans TROPES, les mécanismes de raisonnement sont séparés entre eux.
- Les sources de connaissances sont les seuls éléments intervenant sur le tableau noir. Dans TROPES, les seuls éléments qui interviennent sur la base de connaissances sont les opérations.
- Les sources de connaissances coopèrent pour le partage d'information. Dans une base de connaissances il est évident que les différents mécanismes de raisonnement coopèrent pour le partage de la connaissance.

5. ANALYSES DE COMPLEXITE DE L'ALGORITHME PARALLELE DE CLASSIFICATION DANS TROPES

Ce chapitre présente une analyse de la complexité du modèle séquentiel, puis nous présentons la complexité du modèle parallèle, et finalement, une comparaison entre les deux modèles.

5.1. Le modèle séquentiel

Nous avons donné l'algorithme de classification séquentiel dans la section 1.3.3. Cet algorithme est composé de cinq pas : l'obtention d'information, l'appariement, la propagation de marques, la mise à jour d'information et le choix du point de vue (algorithme 5.1). Dans la suite, nous allons calculer la complexité des diverses procédures qui composent cette algorithme en utilisant le modèle RAM[8] [AHO 74].

```
tant que (non condition pour finir la classification)
        obtention d'information
        appariement
        propagation de marques
        mise-à-jour d'information
        choix du prochain point de vue
fin tantque
```

(Algorithme 5.1)

D'abord, nous faisons l'analyse de la procédure d'obtention d'information, puis des procédure d'appariement, de propagation de marques, de mise-à-jour et finalement de la procédure du choix du prochain point de vue.

5.1.1. Complexité de la procédure séquentielle d'obtention d'information

Dans la procédure d'obtention d'information, le système complète l'information pour la classe actuelle en demandant des informations à l'utilisateur. Les valeurs pour les différents attributs doivent satisfaire les contraintes de la classe. Le calcul de ces contraintes et la validation des valeurs par rapport aux contraintes est la fonction la plus complexe de cette procédure. Pour faire le calcul de complexité nous allons supposer :

- $\mathbf{n}$ = nombre maximal de sous-classes d'une classe.
- $\mathbf{a_n}$ = nombre maximal d'attributs définis ou affinés pour une classe.
- $\mathbf{ct}$ = nombre maximal de contraintes d'un attribut.
- $\mathbf{c_i^k}$ = k-ième contrainte de l'attribut $\mathbf{a_i}$.

[8] RAM : Random Access Machines.

L'algorithme d'obtention d'information est :

pour chaque attribut $\mathbf{a_i} \in$ classeActuelle ($i \in [1..\mathbf{a_n}]$) →
    demander la valeur $\mathbf{v_i}$ pour l'attribut $\mathbf{a_i}$
    pour chaque contrainte $\mathbf{c_i^k} \in \mathbf{a_i}$ ($k \in [1..\mathbf{ct}]$) →
        évaluer($\mathbf{a_i}$,$\mathbf{c_i^k}$,$\mathbf{v_i}$)
    fin
fin

Sous ces hypothèses, nous pouvons faire le calcul de la complexité en supposant que l'utilisateur donne la valeur pour chacun des attributs de la classe actuelle. Au fur et à mesure que les valeurs sont données, elles sont validées par rapport aux contraintes.
Soit :

$\mathbf{f_i}$ = maximum(**évaluer**($\mathbf{a_i}$,$\mathbf{c_i^1}$), ..., **évaluer**($\mathbf{a_i}$,$\mathbf{c_i^k}$), ...,**évaluer**($\mathbf{a_i}$,$\mathbf{c_i^{ct}}$)), où

évaluer($\mathbf{a_i}$,$\mathbf{c_i^k}$) est la complexité de la validation de la valeur $\mathbf{v_i}$ pour l'attribut $\mathbf{a_i}$ ($i \leq \mathbf{ct}$)

par rapport à $\mathbf{c_i^k}$.

Donc, la complexité maximale de chaque validation de la valeur d'un attribut par rapport à une contrainte est $\mathbf{O(f_i)}$. La complexité (figure 5.1) totale de la procédure d'obtention d'information pour la classe actuelle est alors :
O(d'obtention) = $\mathbf{a_n} * (\mathbf{f_1} + ... + \mathbf{f_n}) \leq \mathbf{a_n} * \mathbf{ct} *$ maximum($\mathbf{f_1}$,...,$\mathbf{f_n}$) = $\mathbf{a_n} * \mathbf{ct} * \mathbf{f_0}$,
où $\mathbf{f_0}$ = maximum($\mathbf{f_1}$,...,$\mathbf{f_n}$).
O(d'obtention d'information) = O($\mathbf{a_n} * \mathbf{ct} * \mathbf{f_0}$).

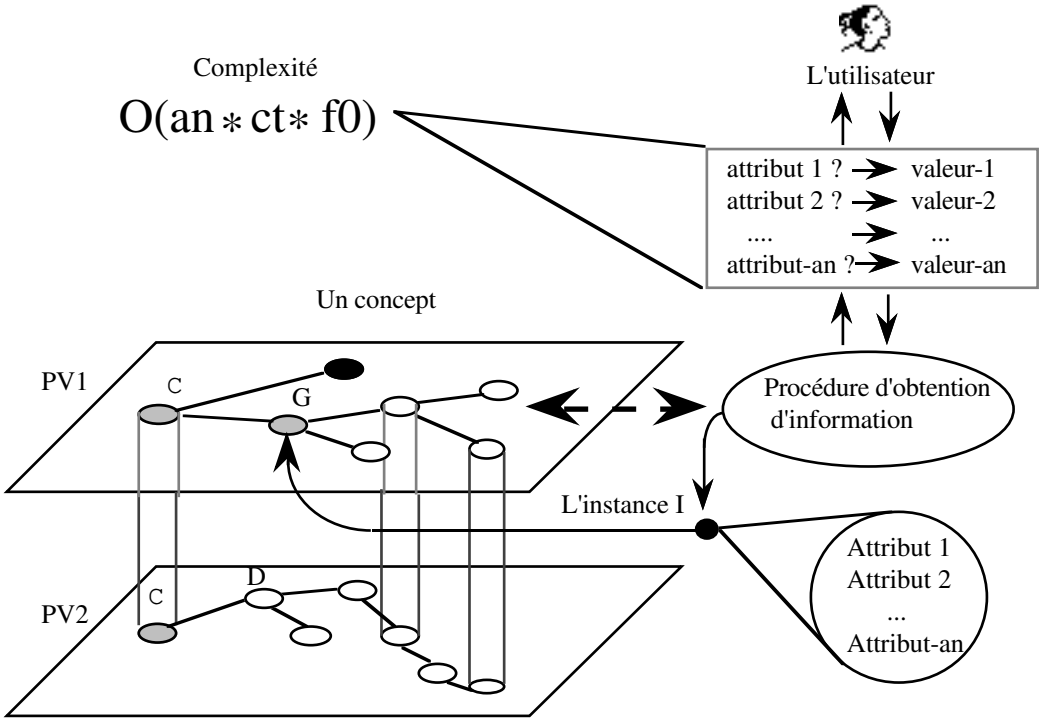


Figure 5.1 : Complexité de la procédure d'obtention d'information.

5.1.2. Complexité de la procédure séquentielle d'appariement

La procédure d'appariement fait le choix d'une classe sûre parmi les sous-classes de la classe actuelle. Pour faire ce choix, la procédure doit demander à l'utilisateur les valeurs inconnues des attributs de ces sous-classes afin de trouver la sous-classe sûre, qui deviendra la nouvelle classe. Au fur et à mesure que sont données les valeurs, elles sont validées par rapport aux contraintes des attributs. Cette fonction de validation est la même que la procédure d'obtention d'information, donc nous pouvons garder les hypothèses de la fonction précédente.
L'algorithme d'appariement est :

pour chaque sous-classe $\mathbf{C_j}$ de la classeActuelle ($j \in [1..\mathbf{n}]$) →
    pour chaque attribut $\mathbf{a_i} \in \mathbf{C_j}$ ($i \in [1..\mathbf{a_n}]$) →
        demander la valeur $\mathbf{v_i}$ pour l'attribut $\mathbf{a_i}$
        pour chaque contrainte $\mathbf{c_i^k} \in \mathbf{a_i}$ ($k \in [1..\mathbf{ct}]$) →
            évaluer($\mathbf{a_i}$,$\mathbf{c_i^k}$,$\mathbf{v_i}$)
        fin
    fin
fin

Dans la suite, nous pouvons faire le calcul de la complexité de l'appariement, en supposant que l'utilisateur donne la valeur pour chacun des attributs des sous-classes que le système lui demande. Au fur et à mesure que les valeurs sont données, elles sont validées par rapport aux contraintes. La complexité maximale de chaque validation par rapport à une contrainte est $\mathbf{O(f_0)}$, et cette validation est faite pour chaque sous-classe. Alors, la complexité (figure 5.2) totale pour la procédure d'appariement est :
$\mathbf{O(d'appariement) = O(n * an * ct * f_0).}$

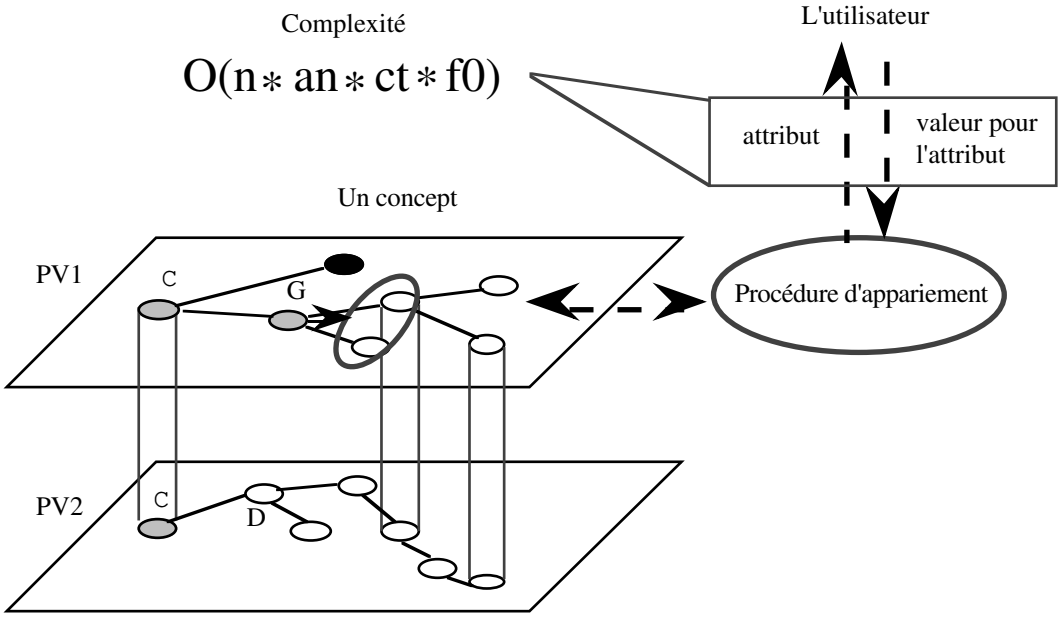


Figure 5.2 : Complexité de la procédure d'appariement sans prise en compte des contraintes des attributs.

5.1.3. Complexité de la procédure séquentielle de propagation de marques

Pour la procédure de propagation de marques, la façon dont les sous-classes d'une classe impossible sont marquées entraîne une complexité différente pour cette procédure.
Dans la suite, nous montrons deux algorithmes différents pour la propagation de marques. Le premier fait une propagation de marques dans toutes les classes appartenant au point de vue. Le deuxième algorithme, avant de faire la propagation de marques, regarde les passerelles actives

dans le point de vue. S'il n'y en a pas, l'algorithme n'effectue aucune propagation dans le point de vue.
Pour le premier algorithme, lorsque la procédure d'appariement a choisi la nouvelle classe et que cette classe est devenue sûre, les classes sœurs deviennent des classes impossibles. L'algorithme commence à effectuer le parcours de tous les sous-arbres dont la racine est une classe impossible et marquer toutes ces sous-classes comme impossibles (figure 5.3).

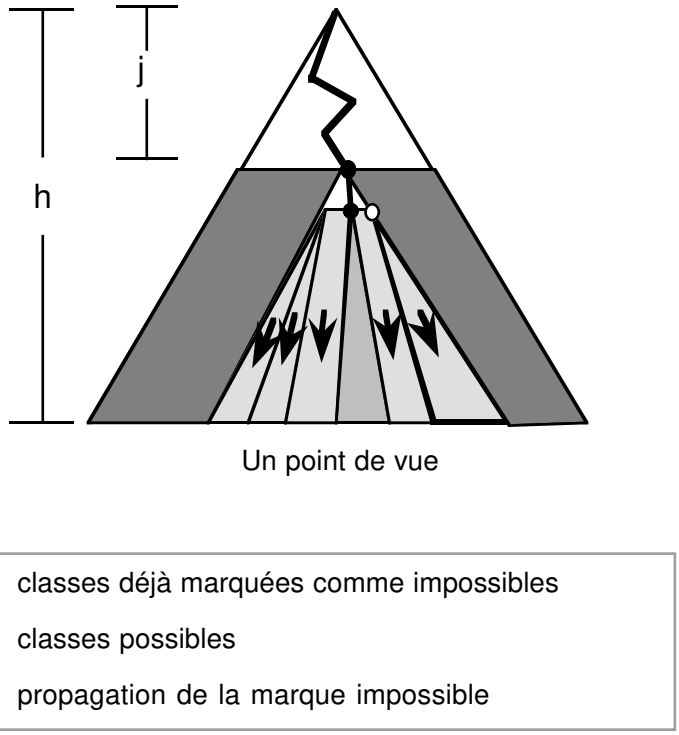


Figure 5.3 : Propagation de la marque impossible pour toutes les sous-arbres dont la racine est une classe impossible.

La complexité de ce parcours est très élevée, étant donné qu'il faut marquer chacune des classes des sous-arbres ayant comme racine les sœurs de la classe sure. Pour le calcul de la complexité soit :
j = hauteur de la classe actuelle dans le point de vue actuel.
h = hauteur de l'arbre de classes (point de vue).
n = nombre maximal de sous-classes de la classe actuelle.
pa = nombre de passerelles actives dans le point de vue actuel.
p = nombre de points de vue du concept.

Donc, pour parcourir chaque arbre, dont la hauteur est **h-j**, il faut visiter $\sum_{i=0}^{h-j} n^i$ classes, en supposant que l'arbre est complet et balancé (figure 5.4). Comme nous l'avons vu, une classe a au maximum **n** sous-classes : lorsque l'algorithme d'appariement fait le choix d'une sous-classe comme sûre celle-ci devient la nouvelle classe actuelle; les autres **n-1** classes soeurs deviennent impossibles et il faut donc parcourir tous leurs arbres en visitant $(n-1) * \sum_{i=0}^{h-j} n^i$ classes, et chacune de ces sous-classes est marquée par l'algorithme comme impossible. La complexité de marquer une classe est **O(1)**. Au total, la complexité de la procédure de propagation de marques dans le point de vue actuel est :

- Sans les passerelles :

$$T_{propag}(j) = (n-1) + (n-1) * T_{propag}(j-1) = n^{h-j+1} \leq n^h$$

$$T_{propag}(h) = 1$$

- Avec les passerelles :

  $$T_{propag}(j) \leq p * n^h$$

  Car si on arrive à une classe déjà marquée, elle n'est pas marquée une nouvelle fois.

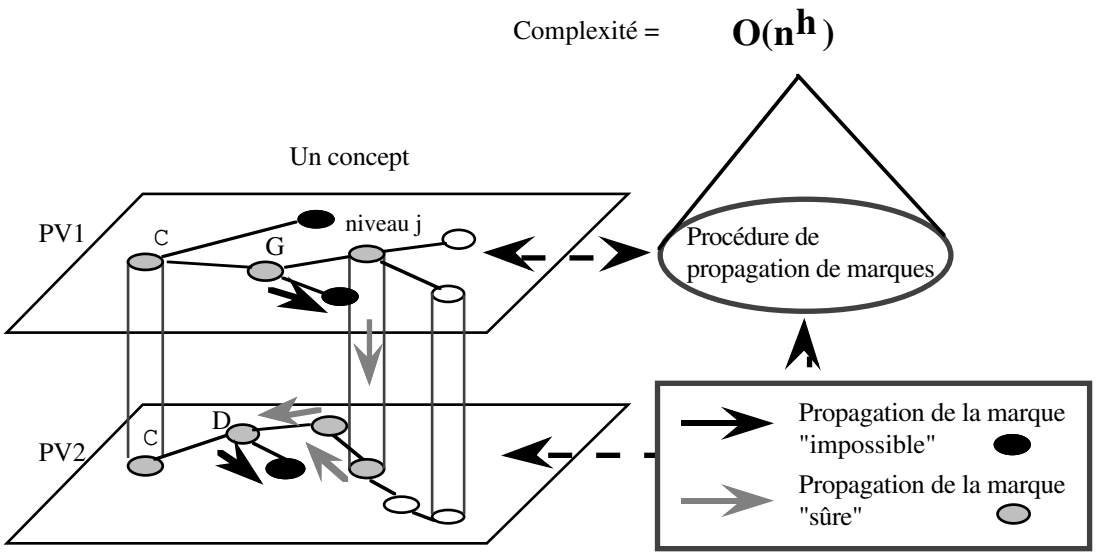


Figure 5.4 : Complexité de la procédure de propagation de marques dans l'arbre à partir du niveau j.

Une autre possibilité pour la de propagation de marques est un algorithme de fermeture transitive, lequel tient en compte le sens des classes impossibles et des passerelles impossibles. Lorsqu'une classe devient impossible, toutes ses sous-classes deviennent elles aussi impossibles. Donc, la procédure de classification de l'instance dans ce point de vue n'a pas besoin de l'information de ces classes impossibles étant donnée l'impossibilité que l'instance appartienne à celles-ci. Donc, le système ne reviendra jamais sur ces classes. Dans ce cas, nous pouvons nous demander s'il est nécessaire de marquer ces classes. La réponse est non, car l'état de ces classes n'a aucune importance pour la suite de la classification.
Dans le cas où il existe des passerelles dans le sous-arbre d'une classe impossible, la seule chose à faire est de marquer la passerelle comme impossible. Lorsqu'une passerelle devient impossible parce qu'une de ses classes dans un point de vue PV_i est devenue impossible, le système peut déduire que l'autre classe de la passerelle dans un autre point de vue PV_j est aussi impossible (si la passerelle est bi-directionnelle).
Dans ce dernier cas, pour faire la propagation de marques dans le point de vue PV_j à partir de la classe D/PV_j , il suffit de marquer la classe D comme impossible. Il n'est alors pas nécessaire de parcourir son sous-arbre, pour les mêmes raisons exposées à dessus.
La complexité de la procédure de propagation de marques, qui prend en compte le sens des classes et les passerelles impossibles est beaucoup plus réduite que la complexité du premier algorithme analysé du fait qu'il ne parcourt pas les sous-arbres des classes impossibles.
Les paramètres de ce dernier algorithme, qui prend en compte la structure des points de vue, sont :
j = niveau dans le point de vue de la nouvelle classe actuelle.
h = hauteur maximal de l'arbre de classes (point de vue).
n = nombre maximal de sous-classes de la classe actuelle.
pa = nombre de passerelles actives dans le point de vue actuel.

p = nombre de points de vue du concept.

- A chaque pas, lorsque la procédure d'appariement a trouvé une nouvelle classe sûre **C** parmi les sous-classes de la classe actuelle, la procédure de propagation marque toutes les classes sœurs de **C** comme impossibles. La complexité de ce pas est **O(n-1)** (figure 5.5).

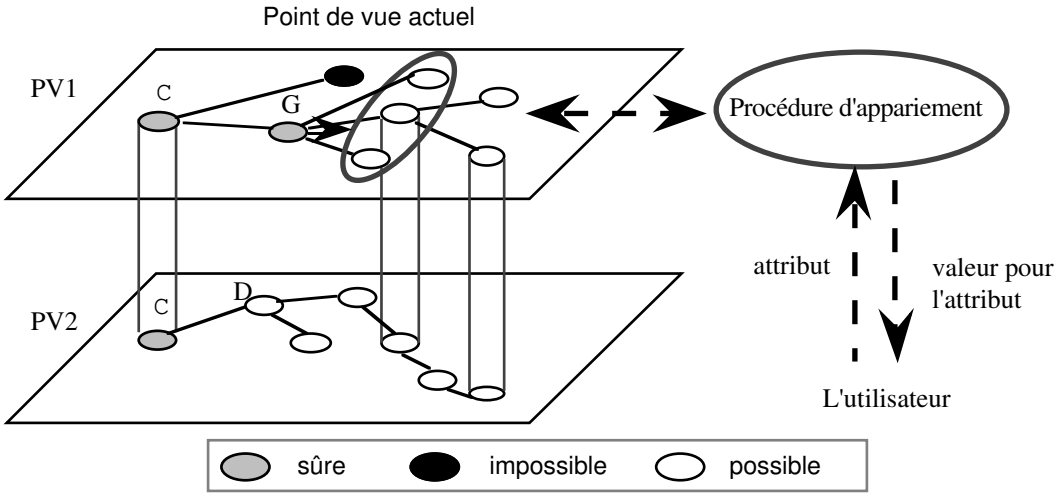


Figure 5.5 : La procédure de propagation marque comme impossibles les classes sœurs de la nouvelle classe.

- Puis, pour chacune des passerelles actives[9] l'algorithme effectue un parcours à partir de la classe de la passerelle par ses sur-classes en regardant leurs états (figure 5.6). Si l'état d'une sur-classe est impossible, la passerelle devient elle aussi impossible (figure 5.7). La complexité de ce parcours est **O(pa∗(h-j))**.

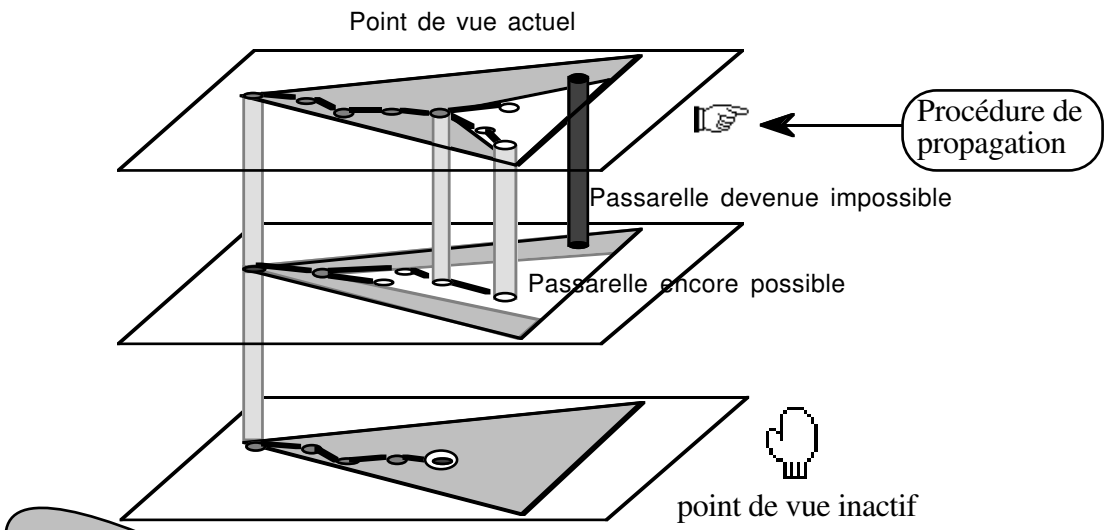
- La complexité de la propagation de marques à partir du niveau j est **O((n-1) + pa∗(h-j))**.

- La complexité totale de l'algorithme de propagation de marques pour le point de vue actuel est: $\mathbf{O(propagation) = \sum_{j=0}^{h} ((n-1) + pa*(h-j)) = h * (n-1) + (pa * h^2)}$

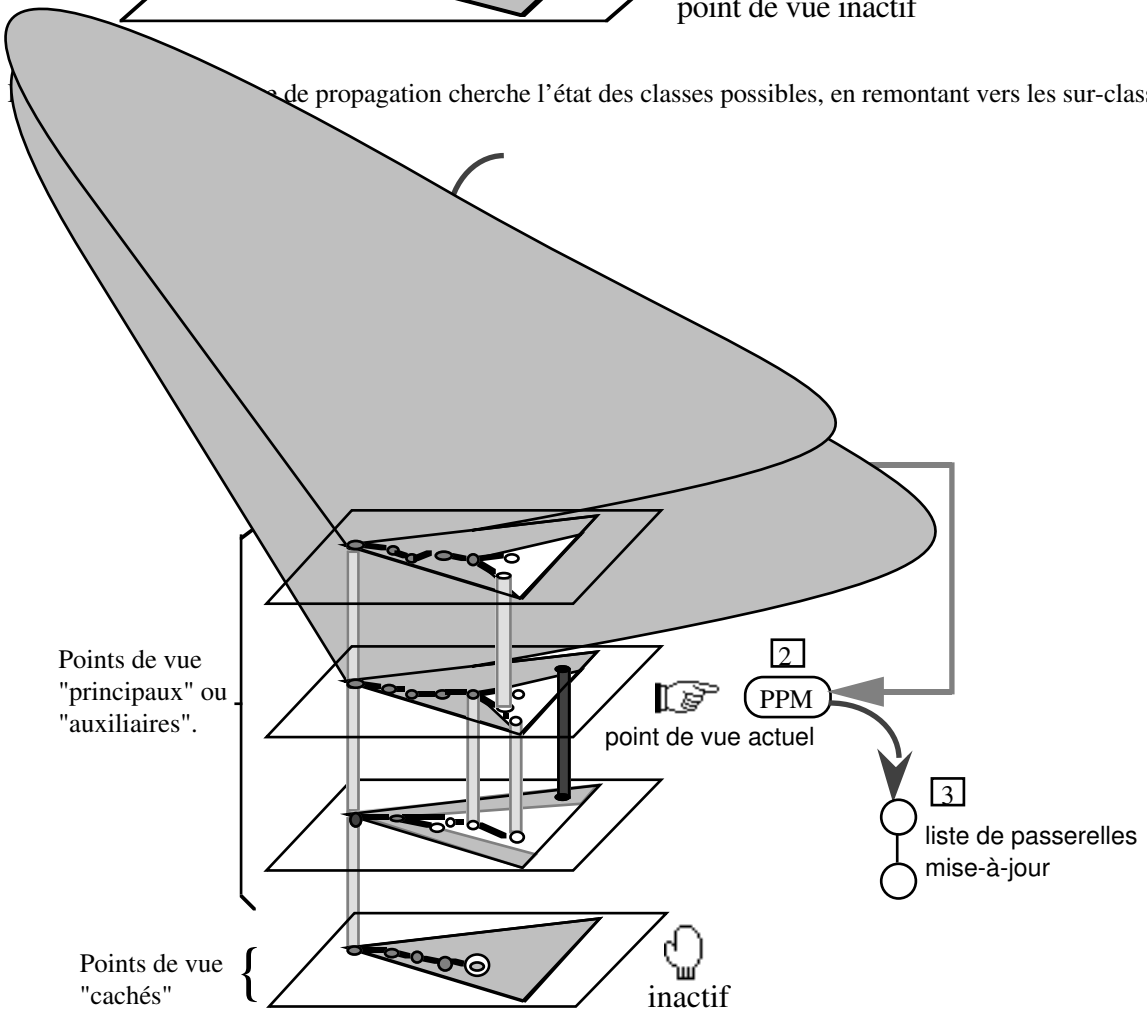
$$= \mathbf{O(pa * h^2).}$$

- Finalement, la complexité totale de l'algorithme de propagation de marques pour un concept est: $\mathbf{O(propagation\ concept) = O(p*pa*h^2).}$

[9] Une passerelle active est une passerelle où l'état est possible. L'état d'une passerelle est possible, si le système n'a jamais marqué ses classes comme sûres ou impossibles.



de propagation cherche l'état des classes possibles, en remontant vers les sur-classes.



Exécution de la procédure de propagation de marques

1	La liste de passerelles actives au début de la propagation de marques
2	Mise-à-jour de la liste de passarelles
3	La liste de passerelles encore actives dont l'état est possible

Figure 5.7 : La procédure de propagation marque (PPM) comme impossibles les passerelles où ses classes sont impossibles.

Nous venons d'analyser deux façons de faire la propagation de marques en observant la complexité pour chacun des algorithmes. Il est très clair que la deuxième façon de faire la propagation de marques a beaucoup plus d'avantages que la première, nous avons donc choisi d'implémenter le deuxième algorithme (table 5.2).

O(propagation concept - algorithme 1) = O(p∗n^h)
O(propagation concept - algorithme 2) = O(p∗pa∗h^2).

Table 5.2 : Complexité de la procédure de propagation de marques.

5.1.4. Complexité de la procédure séquentielle de mise-à-jour

Le quatrième pas de l'algorithme de classification est la procédure de mise-à-jour de l'information sur l'instance par rapport à sa nouvelle classe d'appartenance. La complexité de cette procédure est **O(1)**.

5.1.5. Complexité de la procédure séquentielle du choix du prochain point de vue

Le dernier pas de l'algorithme séquentiel de classification concerne le choix du prochain point de vue à considérer (figure 5.8). Il existe plusieurs critères de choix [MAR 89] :

critère 1 : Point de vue ayant la passerelle la plus basse vers un point de vue principal.

critère 2 : Point de vue permettant de retourner le plus vite possible au point de vue principal.

critère 3 : Au départ de la classification, l'utilisateur ordonne les points de vue auxiliaires en accord avec sa connaissance sur chacun d'eux. L'algorithme de classification prend en compte cet ordre : le prochain point de vue à choisir est le premier point de vue auxiliaire ordonné par l'utilisateur.

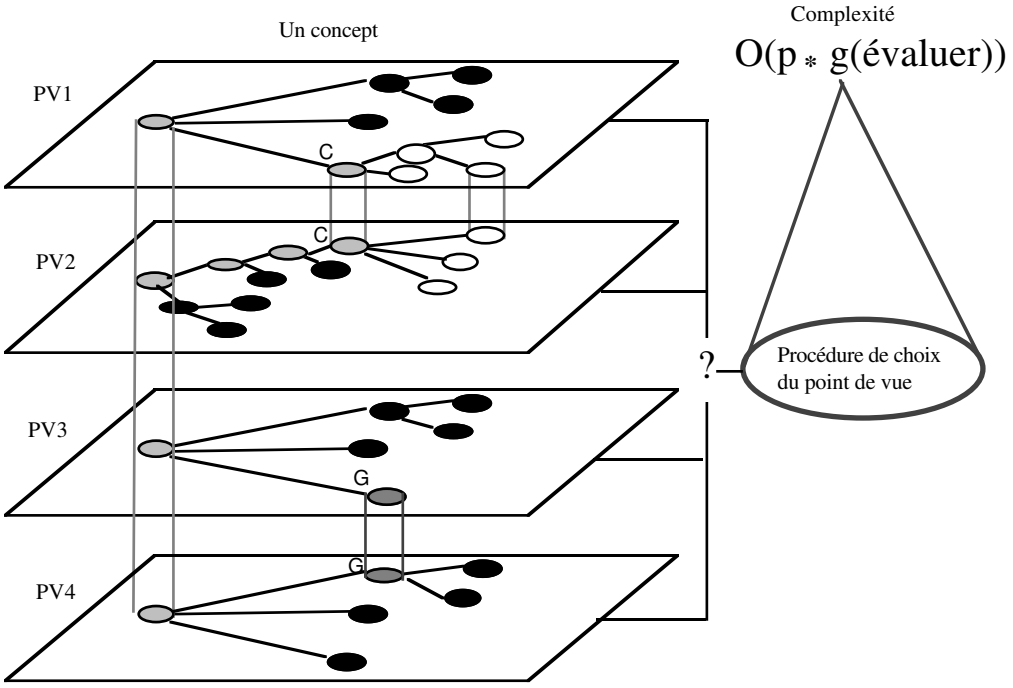


Figure 5.8 : Complexité de la procédure de choix du nouveau point de vue.

L'algorithme utilisé dans le processus de classification se base sur le dernier critère. Donc, pour choisir le prochain point de vue actuel, la procédure n'a qu'à choisir le premier point de vue dans une liste ordonnée de points de vue auxiliaires. La complexité du choix est **p∗g(évaluer)**, où **g(évaluer)** représente la fonction d'évaluation d'un point de vue et **p** est le nombre maximum de points de vue. Dans notre cas, la complexité de g(évaluer) est **O(1)**, donc la complexité de la procédure de choix du point de vue est **O(p)**.

5.1.6. Complexité totale de l'algorithme de classification séquentielle

Nous avons analysé la complexité de chacune des fonctions qui composent l'algorithme de classification d'une instance dans la base de connaissances multi-points-de-vue; nous allons calculer la complexité du cycle de classification, qui est composé de cinq pas :

tant-que not(finir)
    obtention_d'information
    appariement
    propager_marques
    mise à jour de l'information
    choix_point_de_vue
fin tant-que

soient :

- **p** = nombre du points de vue du concept
- **h** = hauteur maximale de l'arbre de classes d'un point de vue
- **n** = nombre maximum de sous-classes d'une classe
- **an** = nombre maximum d'attributs définis dans une classe
- **ct** = nombre maximal de contraintes d'un attribut.
- **f0** = complexité maximale de la validation des valeurs des attributs par rapport aux contraintes
- **pa** = nombre de passerelles dans le concept.

La complexité de chacun des pas est reportée dans le cycle de classification (dans un concept):

tant que (non condition pour finir la classification)

O(obtention d'information) = $O(an * ct * f_0)$

O(appariement) = $O(n * an * ct * f_0)$.

O(propagation de marques) = $O((n-1) + pa * (h-j))$

O(actualiser l'information) = $O(1)$

O(choix du point de vue) = $O(p)$

fin tant que

ou encore

tant que (non condition pour finir la classification)

$$O(f) \quad \text{où } f = (an*ct*f_0)+(n*an*ct*f0)+((n-1)+pa*(h-j))+1+p$$

fin tant que

Le résultat final de la complexité de l'algorithme de classification d'une instance dans tous les point de vue du concept est :

$$O(\text{classification}) = \sum_{j=0}^{h} p*((an*ct*f_0)+(n*an*ct*f_0)+((n-1)+pa*(h-j))+1+p)$$

$$O(\text{classification}) = O(p*((h*n*an*ct*f0)+(n+pa*h^2)+h))$$

$$= O(p*((h*n*an*ct*f0)+(n+pa*h^2))).$$

A partir de ces résultats nous pouvons conclure que la complexité de l'algorithme de classification dépend fortement de deux arguments : le premier est la fonction d'évaluation des valeurs des attributs par rapport à leurs contraintes, et le deuxième est la hauteur de l'arbre de classes. Si l'ordre de la fonction f_0 est constante ou linéaire, la complexité totale de la classification dépend de la hauteur des arbres des points de vue. Par contre, si la complexité de cette fonction d'évaluation est plus haute que la complexité de propagation de marques ($f_0 \geq pa*h^2$), nous pouvons conclure que la classification dépend de la fonction d'évaluation, et non pas du nombre de passerelles, ni de la hauteur de l'arbre de classes.
Si la complexité de la fonction d'évaluation des valeurs des attributs par rapport aux contraintes est considérée comme constante, la complexité est réduite à :

$$O(\text{classification}) = O(p*((h*n*an*ct)+(n+pa*h^2)+h))$$

$$= O(p*((h*n*an*ct)+(n+pa*h^2))).$$

5.2. Complexité du modèle parallèle

Dans cette section, nous allons présenter le calcul de complexité de l'algorithme parallèle de classification d'instances dans un concept multi-points de vue. En même temps, nous ferons l'analyse du nombre de processus qui sont créés pendant la classification.
Avant tout, nous présentons le modèle de calcul parallèle qui a été utilisé pour le calcul de la complexité :
Le modèle primaire de computation parallèle est le PRAM ("Parallel Random-Access Machine"). Le PRAM est un modèle idéal et peut être vu comme une analogie avec le modèle séquentiel RAM. Le PRAM peut être classifié en accord avec la façon d'accéder la mémoire partagée [KAR 90] :

- Lecture exclusive - écriture exclusive (EREW). L'accès à la mémoire est exclusive, au plus un processeur peut lire ou écrire simultanément dans la même position de mémoire.
- Lecture concurrent - écriture exclusive (CREW). L'accès à la mémoire pour l'écriture est exclusive, plusieurs processeurs peuvent lire la même position de mémoire, mais au plus un processeur peut écrire simultanément dans la même position de mémoire.
- Lecture exclusive - écriture concurrent (ERCW). L'accès à la mémoire pour la lecture est exclusive, plusieurs processeurs peuvent écrire dans la même position de mémoire, mais au plus un processeur peut lire simultanément dans la même position de mémoire.

- Lecture concurrent - écriture concurrent (CRCW). L'accès à la mémoire pour l'écriture et aussi pour la lecture peut être fait de façon simultanée dans la même position de mémoire. Pour résoudre les conflits pendant l'écriture concurrente, il existe plusieurs modèles :
    - CRCW COMMUNE : Tous les processeurs écrivant simultanément dans la même position de mémoire doivent écrire la même valeur.
    - CRCW ARBITRAIRE : Lorsque plusieurs processeurs écrivent dans la même position de mémoire, la valeur est choisie de façon aléatoire entre toutes les valeurs possibles.
    - CRCW PRORITAIRE : Les processeurs ont une priorité. Lors de l'écriture concurrente dans la même position de mémoire par plusieurs processeurs au même moment, la valeur choisie est celle du processeur le plus important (ayant la priorité la plus haute).

Néanmoins, un algorithme qui s'exécute dans un PRAM CRCW peut être simulé par un PRAM EREW avec le même nombre de processeurs avec un temps d'exécution a crû par un facteur de $O(\log P)$[10], où P est le nombre de processeurs.
Nous rappelons qu'à chaque nouveau pas de la classification, il faut, soit marquer en parallèle les différents classes comme impossibles, soit faire l'appariement d'une instance avec les sous-classes de la classe actuelle en parallèle. De plus, ces pas sont réalisées en parallèle pour les divers point de vue. Donc, ce programme est composé d'un ensemble de processus différents {p1, ..., pn} (n étant le nombre de processeurs), qui s'exécutent en parallèle sur un ensemble de données D1,…Dm quelconques (m étant le nombre du points de vue du concept). Le problème est donc fort simple dans sa formulation et semble bien s'adapter à une programmation de type "multiple instructions-multiple data" avec mémoire partagée et dont la lecture et l'écriture sont concurrentes. Pour résoudre l'écriture simultanée, le modèle PRAM CRCW COMMUNE est le mieux adapté.
L'algorithme parallèle (algorithme 5.2a et 5.2b) est composé de quatre pas : la procédure d'obtention d'information, la procédure d'appariement, la procédure de propagation de marques et la procédure de mise à jour de l'information.

h = hauteur maximale de l'arbre de classes (point de vue).
n = nombre maximal de sous-classes de la classe actuelle.
an = nombre maximum d'attributs définis dans une classe
ct = nombre maximale de contraintes d'un attribut.
f0 = complexité maximal de la validation des valeurs des attributs par rapport aux contraintes
pa = nombre de passerelles actives dans le point de vue actuel.
p = nombre de points de vue du concept.

pour chaque point de vue PV (i=1,...,p) **en parallèle**
    appeler procédure classification(PVi,l'instance `In`) **en parallèle**

Algorithme 5.2a

---

[10] Pour trouver ce facteur, il faut comprendre la façon de faire la simulation :
Soit :
N : le nombre de processeurs.
P_1, ... P_N les processeurs.
x : la valeur qui doit être connue de tous les processeurs.
Il faut faire un diffusion entre tous les processeurs de la façon suivante :
1. P_1 lit **x** et communique cette valeur à P_2.
2. P_1 et P_2 communiquent simultanément cette valeur à P_3 et P_4 respectivement.
3. P_1, P_2, P_3 et P_4 communiquent la valeur à P_5, P_6, P_7 et P_8 respectivement, etc.
Cette diffusion a besoin de log(N) pas.

où la procédure de classification de l'instance est :

pour chaque point de vue PV (i=1,...,p) **en parallèle**

/* La classe sûre du point de vue PV_i est C_j^i */

- **appeler** procédure d'obtention d'information (PV_i,C_j^i, In) **en parallèle**
- **faire en parallèle**
  - **pour chaque** sous-classe C_k^i de C_j^i (k=1,...,n)

    appeler procédure d'appariement (PV_i,C_k^i, In) **en parallèle**

    pour chaque attribut $At_{k_j}^i$ de la classe C_k^i **en parallèle**

    pour chaque contrainte $Ct_{k_{j_L}}^i$ de l'attribut $At_{k_j}^i$ **en parallèle**

    évaluer($At_{k_j}^i$, $Ct_{k_{j_L}}^i$, valeur)

    marque = "impossible"
    Pa = ensemble de passerelles du point de vue PV_i
  - **si** une sous-classe C_k^i devient "impossible" **alors**

    appeler propagation de marques(PV_i,C_k^i,marque,Pa) **en parallèle**

    sinon

    si une sous-classe C_k^i devient "sûre" **alors**

    marque = "sûre"

    appeler propagation marques(PV_i,c_k^i,marque,Pa) **en parallèle**

    pour chaque sous-classe sœur C_l^i de C_k^i (l=1,...,n; l≠k) **en parallèle**

    marque = "impossible"

    appeler propagation marques(PV_i,C_l^i,marque,Pa) **en parallèle**

    fin-si

    fin-si
- **appeler** procédure de mise à jour (PV_i,C_j^i, In)

Algorithme 5.2b : La classification en parallèle d'une instance dans TROPES

Comme nous pouvons le voir, dans le modèle parallèle la procédure de choix du point de vue n'est pas présente. En effet, dans le modèle parallèle, la classification se fait dans tous les points de vue à la fois. En conséquence, il n'existe pas de point de vue actuel. Tous les points de vue sont actuels. Si un point de vue devient inactif, l'algorithme de classification ne le prend plus en compte.

Dans la suite, nous allons exposer chacune des procédures de l'algorithme de classification parallèle, en commençant par la procédure d'obtention l'information.

5.2.1. Complexité de la procédure parallèle d'obtention l'information

D'abord, nous analysons la façon de demander les valeurs à l'utilisateur. La première façon est de demander en une seule fois à l'utilisateur l'ensemble des valeurs de tous les

attributs qui composent la classe actuelle de chaque point de vue. Dans l'algorithme séquentiel, au fur et à mesure que l'utilisateur donne une valeur pour un attribut, elle est validée par rapport aux contraintes. Cette validation est faite dans le pire des cas pour chacun des points de vue, lorsqu'ils deviennent actuels. La complexité de la procédure (séquentielle) d'obtention l'information est $\mathbf{O(p*h*an*ct*f_0)}$. Si nous faisons la validation de chaque valeur, au fur et à mesure qu'elle est donnée pour l'utilisateur, de façon parallèle dans tous les points de vue, et pour chaque point de vue nous faisons la validation en parallèle des valeurs de tous les attributs des classes actuelles par rapport aux contraintes, la complexité peut être réduite à $\mathbf{O_{//}(h*f_0)}$. Le nombre total de processeurs logiques nécessaires pour faire cette tâche est $\mathbf{H(p*an*ct)}$[11].

5.2.2. Complexité de la procédure parallèle d'appariement

Pour la procédure d'appariement, nous pouvons faire les mêmes remarques que pour la procédure d'obtention l'information. Dans le cas de l'appariement, nous pouvons réaliser également en parallèle la validation des valeurs des attributs par rapport à leurs contraintes dans toutes les sous-classes de la classe actuelle d'un point de vue (figure 5.9). Dans ce cas, la complexité de la procédure est réduite de $\mathbf{O(p*(h*n*an*ct*f0))}$ à $\mathbf{O_{//}(p*h*f_0)}$**.** De même, les valeurs des attributs peuvent être communiquées aux différents points de vue en parallèle pour valider les valeurs dans toutes les sous-classes des classe actuelles (chaque point de vue actif a sa propre classe actuelle : s'il y a **p** points de vue alors il y a **p** classe actuelles). Dans ce cas, la complexité sera réduite à $\mathbf{O_{//}(h*f0)}$**.** Etant donné qu'il y a, au plus, **p** points de vue actifs, et que pour chacun il n'y a qu'une classe actuelle, ayant **n** sous-classes, nous avons donc **n*p** sous-classes possibles. Chacune de ces sous-classes a au maximum **an** attributs, et chacun de ces attributs a au maximum **ct** contraintes. Donc, si toutes ces validations sont faites en parallèle, nous avons donc $\mathbf{H(p*n*an*ct)}$ processeurs logiques (de la procédure d'appariement), qui s'exécutent en parallèle.

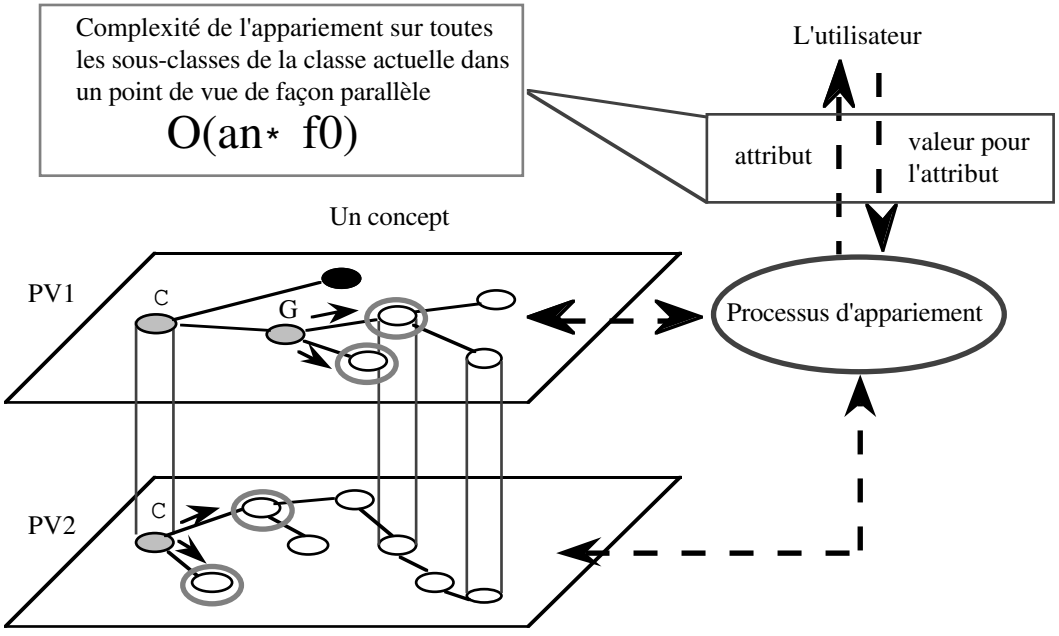


Figure 5.9 : Procédure d'appariement : Validation des valeurs des attributs par rapport à leurs contraintes dans toutes les sous-classes de la classe actuelle en parallèle dans chaque point de vue.

5.2.3. Complexité de la procédure parallèle de propagation de marques

Pour l'algorithme séquentiel de la propagation de marques, nous avons montré deux façons de faire cette propagation. La première parcourt l'arbre entier et la deuxième prend en compte les passerelles et les classes impossibles.

---

[11] **H(f(n))** veut dire que le nombre de processeurs logiques est une fonction dépendante de **f(n).** Le comportement de la fonction H est similaire à celui de la fonction O(f(n)).

Pour l'algorithme parallèle nous montrerons aussi deux façons de faire la propagation de marques dans un point de vue.
L'idée du premier algorithme est très simple. Il s'agit de parcourir en parallèle chaque sous-arbre d'une classe impossible pour marquer les classes comme impossibles. Au fur et à mesure que l'algorithme descend d'un niveau, chaque sous-arbre est parcouru lui aussi en parallèle (figure 5.10).

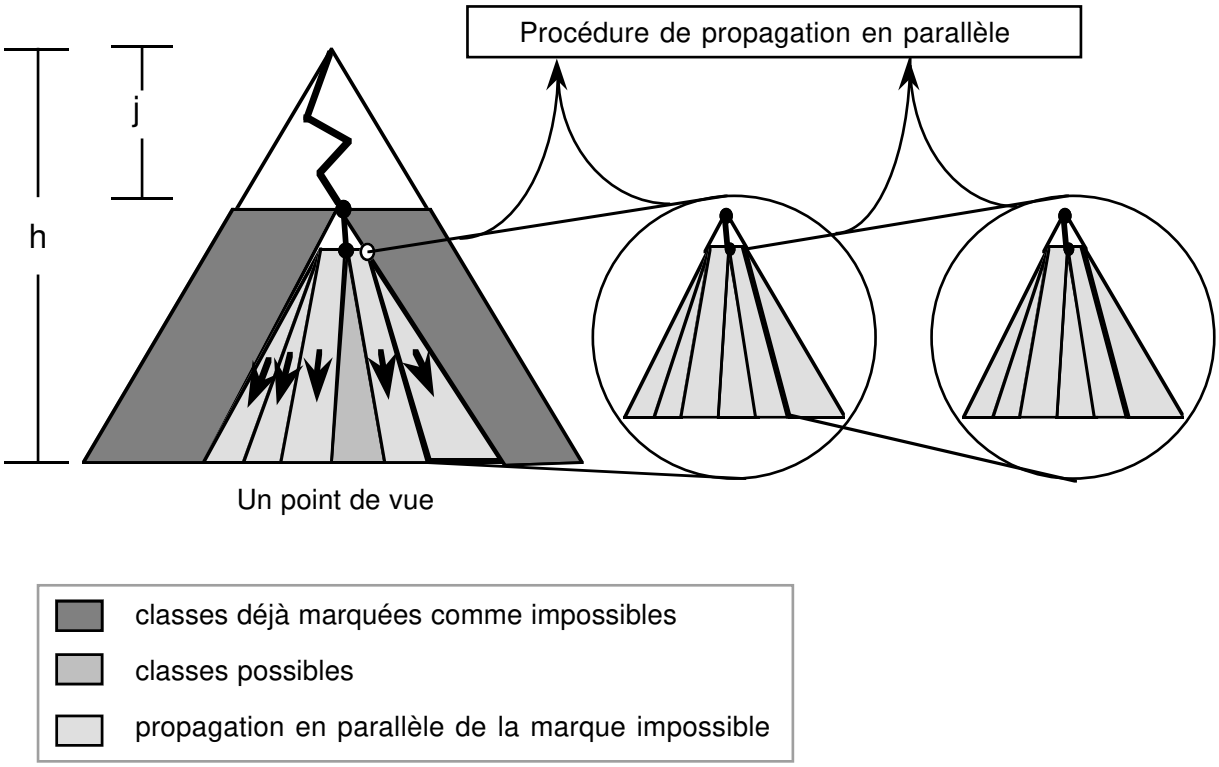


Figure 5.10 : Procédure de propagation de marques sans tenir compte de la signification d'une classe impossible et ni d'une passerelle impossible.

La complexité de cet algorithme est :

$$\mathbf{O_{//}(propagation\ \ niveau\ \ j)} = \sum_{i=1}^{h-j} 1 \ = \mathbf{O_{//}(h\text{-}j)}$$

Et la complexité de la propagation dans tous les points de vue du concept est :

$$\mathbf{O_{//}(propagation\ \ concept)} = \sum_{j=1}^{h} \left(\sum_{i=1}^{h-j} 1 \right) \ = \mathbf{O_{//}(h^2)}$$

Le nombre de processeurs logiques nécessaires pendant la propagation de marques pour un point de vue PV_i, est proportionnel au nombre de classes de PV_i. En effet, le système a besoin d'un processeur logique pour chaque classe dont le but est de marquer cette classe comme impossible. Donc, le nombre de processeurs logiques est $\mathbf{H(n^h)}$, pour un point de vue. Si initialement il y a **p** points de vue actifs, le nombre total de processeurs logiques est donc $\mathbf{H(p*n^h)}$.

L'idée du deuxième algorithme reprend celle du deuxième algorithme de propagation de marques du modèle séquentiel. Soient :
j = niveau dans le point de vue de la nouvelle classe actuelle.
h = hauteur de l'arbre de classes (point de vue).
n = nombre maximal de sous-classes de la classe actuelle.
pa = nombre de passerelles actives dans le point de vue actuel.

- A chaque pas, lorsque la procédure d'appariement a trouvé une nouvelle classe C sûre parmi les sous-classes de la classe actuelle, la procédure de propagation en parallèle marque toutes les classes sœurs de C comme impossibles. La complexité de ce pas est $\mathbf{O_{//}(1)}$.
- Puis, pour chacune des passerelles actives elle fait un parcours à partir de la classe de la passerelle vers ses sur-classes en regardant leurs états. Ce parcours est fait de façon parallèle pour toutes les passerelles actives. Si l'état d'une sur-classe est impossible, la passerelle devient elle aussi impossible. La complexité de ce parcours est $\mathbf{O_{//}(h-j)}$. Ce dernier pas est fait en parallèle dans tous les points de vue actifs.
- La complexité de la propagation de marques à partir du niveau j pour les deux premiers pas est $\mathbf{O_{//}(1)+O_{//}(h-j)=O_{//}(h-j)}$.
- La complexité totale de l'algorithme de propagation de marques pour le concept est : $\mathbf{O_{//}(propagation\ du\ concept) = O_{//}(h).}$

La complexité du deuxième algorithme de propagation de marques est plus petite que celle-ci du premier algorithme. Néanmoins, il existe un avantage du premier algorithme par rapport au deuxième : le nombre du processeurs logiques qu'il faut avoir pour faire la propagation de marques est beaucoup plus réduit que pour le second. Néanmoins, pour le deuxième algorithme le nombre de processeurs dépend du nombre de passerelles, donc si celui-ci est petit, le nombre de processeurs nécessaires pour la propagation est réduit.
Le nombre de processeurs logiques nécessaires pendant la propagation de marques en utilisant ce dernier algorithme, dans un point de vue PVi, est proportionnel au nombre de passerelles de PVi. Le système a besoin d'un processeur logique pour chaque passerelle active afin de déterminer l'état d'une sur-classe comme sûre ou impossible et pouvoir marquer la passerelle. Donc, le nombre total de processeurs logiques pour le concept est $\mathbf{H(pa*p*n^h)}$.
Le nombre de processeurs logiques pour le premier algorithme de propagation de marques est $\mathbf{H(p*n^h)}$ et pour le deuxième algorithme de $\mathbf{H(pa*p*n^h)}$. Cette différence est importante car la charge des divers processeurs dépend du nombre de processeurs logiques placés et de la communication de ces processeurs logiques [TAL 91].

5.2.4. Complexité de la procédure parallèle de mise-à-jour

La complexité de la procédure de mise-à-jour de l'information dans le modèle parallèle est la même que celle de la procédure d'actualisation l'information dans le modèle séquentiel : elle est égale à $\mathbf{O_{//}(1)}$ et le nombre de processeurs logiques est **H(1).**

5.2.5. Complexité totale de l'algorithme de classification parallèle

Finalement la complexité totale de l'algorithme parallèle de classification d'instances dans un concept multi-points-de-vue et nombre de processeurs logiques sont :

$O_{//}$(classification en parallèle)	$= O_{//}(h*f0+h*f0+1+h)$
	$= O_{//}(h*f0+h)$.
H(classification en parallèle)	$= H(p*an*ct+p*n*an*ct+pa*p*n^h+1)$
H(p,n,an,ct,h)	$= H(p*(n*an*ct+pa*n^h))$.

A partir de ces résultats, nous pouvons faire les mêmes conclusions que pour le modèle séquentiel. La complexité de l'algorithme parallèle de classification dépend fortement de la fonction d'évaluation des valeurs des attributs par rapport à leurs contraintes. Si l'ordre de cette fonction est constante ou linéaire, la complexité totale de la classification dépend de la hauteur des arbres de classes. Par contre, si la complexité de cette fonction d'évaluation est plus élevée que la complexité de propagation de marques (**f0≥h**), nous pouvons conclure que la classification dépend de la fonction d'évaluation et non pas du nombre de passerelles ni de la hauteur de l'arbre de classes.
Si la complexité de la fonction d'évaluation des valeurs des attributs par rapport aux contraintes est considérée comme constante, la complexité est réduite à :

$O_{//}$(classification en parallèle)	$= O_{//}(h+h+1+h)$
	$= O_{//}(h+h)$
	$= O_{//}(h)$
H(classification en parallèle)	$= H(p*an*ct+p*n*an*ct+pa*p*n^h+1)$
H(p,n,an,ct,h)	$= H(p*(n*an*ct+pa*n^h))$.

Si le travail est défini comme le produit de la complexité parallèle par le nombre de processeurs nécessaires pour la tâche : $W_{//}(n) = O_{//}(n)*H(n)$ [KAR 90].
Alors, pour la classification en parallèle d'une instance dans TROPES, le travail est :

$$W_{//}(p,h,n,an,ct) = O_{//}(h)*H(p*(n*an*ct+pa*n^h))$$

Selon les résultats précédents, la complexité de l'algorithme parallèle est beaucoup plus réduite que celle de l'algorithme séquentiel. Cependant, le nombre de processeurs dont le système a besoin est élevé. Si le nombre de processeurs nécessaires est plus petit que le nombre de processeurs **Np** de l'ordinateur dans lequel s'exécute la classification (principe de Brent [KAR 90]), le temps dans lequel s'effectue l'algorithme en parallèle est :

$T_{//}(p,h,n,an,ct)$	$= (O_{//}(h)*H(p*(n*an*ct+pa*n^h))/Np)$
	$= (h*p*(n*an*ct+pa*n^h)/Np)$

Le modèle utilisé, pour la mise en œuvre, est un PRAM EREW, la complexité de l'algorithme est accrue d'un facteur de log(nombre de processeurs). La mise en œuvre a été faite sur un système réparti local[12], qui s'adapte bien à un modèle EREW, pour lequel chacune des machines du système réparti peut être envisagée comme une RAM [SAN 90]. Le coût de cette simulation (du meilleur modèle adapté au problème CRCW COMMUNE sur un modèle EREW) est log(**Np**). Le temps total d'exécution en parallèle pour l'algorithme de classification sur un système réparti (EREW) est :

$$T_{//}(p,h,n,an,ct) = (h*p*(n*an*ct+pa*n^h)/Np)*log(Np)$$

Il existe aussi le concept d'algorithme PRAM optimal et efficace pour un problème P, soit :

---

[12] La distinction entre un système réparti local (sur un réseau LAN) et un système réparti non-local (sur un réseau WAN) est importante, étant donné que pour le premier les coûts de communication peuvent être ignorés, si le réseau local est un réseau à diffusion. Par contre, pour le deuxième cas, le nombre de données qu'il faut échanger pour faire parvenir un message d'une machine à une autre est très grand. Il peut arriver que le temps mis pour cet échange soit plus long que le calcul même.

- $T_{séq}(n)$ est le temps de l'algorithme séquentiel le plus rapide, qui résoudre le problème P.

- $polylog(n) = \bigcup_{k>0} O(\log^k n)$

- Un algorithme est **optimal** si :

$O_{//}(n) = polylog(n)$
$W_{//}(n) = H(n)*O_{//}(n) = O(T_{séq}(n))$

- Un algorithme est **efficace** si :

$O_{//}(n) = polylog(n)$
$W_{//}(n) = T_{séq}(n)*polylog(n)$

Nous avons développé un algorithme parallèle pour la classification d'instances dans TROPES. Ce algorithme est un algorithme efficace.

5.3. Analyse des deux modèles

Dans les sections précédentes, nous avons présenté les analyses de complexité pour le modèle séquentiel et pour le modèle parallèle (CRCW COMMUNE) de la classification d'instances dans un base de connaissances multi-points de vue. Ces complexités sont rappelées dans la table 5.3.

O(classification séquentielle)	**$= O(p*(h*n*an)+(n+pa*h^{2)}+p*h))$**
	$= O(p*(h*n*an)+(n+pa*h^2))$.
$O_{//}$(classification en parallèle)	**$= O_{//}(h)$**
$O_{//}$(classification en parallèle)	**$\leq$ O(classification séquentielle)**
H(classification séquentielle)	**$= H(1)$**
H(classification en parallèle)	**$= H(p*(n*an*ct+pa*n^h))$.**
H(classification séquentielle)	**$\leq$ H(classification en parallèle)**
Temps séquentiel = $T_{séq}$	**$\leq$ Travail en parallèle = $W_{//}$**
$T_{séq} = O(p*(h*n*an)+(n+pa*h^2))$	**$\leq W_{//}(h*p*(n*an*ct+pa*n^h))$**

Table 5.3 : Complexités des algorithmes séquentiel et parallèle (dans un modèle CRCW COMMUNE) pour la classification d'une instance

Ces résultats montrent que, l'algorithme parallèle réduit la complexité en tous les facteurs par rapport à l'algorithme séquentiel, et qu'il ne dépend que de la hauteur maximale des points de vue.
Dans l'algorithme que nous avons développé, nous n'avons pas fait l'évaluation en parallèle des attributs ni l'évaluation en parallèle des valeurs par rapport aux contraintes des attributs. L'algorithme qui a été développé est (algorithme 5.3a et 5.3b) le suivant :
soient :

h = hauteur maximal de l'arbre de classes (point de vue).
n = nombre maximal de sous-classes de la classe actuelle.
an = nombre maximum d'attributs définis dans une classe
ct = nombre maximal de contraintes d'un attribut.
f0 = complexité maximal de la validation des valeurs des attributs par rapport aux contraintes
pa = nombre de passerelles actives dans le point de vue actuel.
p = nombre de points de vue du concept.

pour chaque point de vue PV (i=1,...,p) **en parallèle**
    appeler procédure classification(PVi,l'instance `In`) **en parallèle**

Algorithme 5.3a

où la procédure de classification de l'instance est :

pour chaque point de vue PV (i=1,...,p) **en parallèle**

/* La classe sûre du point de vue PV_i est c_j^i */

- **appeler** procédure d'obtention d'information (PV_i,c_j^i, In)
- **faire en parallèle**
  - **pour chaque** sous-classe c_k^i de c_j^i (k=1,...,n)

    appeler procédure d'appariement (PV_i,c_k^i, In) **en parallèle**
  - **si** une sous-classe c_k^i devient "impossible" **alors**

    marque = "impossible"
    Pa = ensemble de passerelles du point de vue PV_i
    appeler propagation de marques(PV_i,c_k^i,marque,Pa) **en parallèle**

    sinon si une sous-classe c_k^i devient "sûre" **alors**

    pour chaque sous-classe sœur c_l^i de c_k^i (l=1,...,n; l≠k)

    marque = "sûre"
    appeler propagation marques(PV_i,c_k^i,marque,Pa) **en parallèle**

    marque = "impossible"
    appeler propagation marques(PV_i,c_l^i,marque,Pa) **en parallèle**

    fin-si
- **appeler** procédure de mise à jour (PV_i,c_j^i, In)

Algorithme 5.3b : La classification en parallèle d'une instance dans TROPES

Et la complexité de cet algorithme est (table 5.4) :

O(classification séquentielle)	**$= O(p*(h*n*an)+(n+pa*h^{2)}+p*h))$**
	$= O(p*(h*n*an)+(n+pa*h^2))$.
$O_{//}$(classification en parallèle)	**$= O_{//}(h*an*ct+h)$**
$O_{//}$(classification en parallèle)	**$\leq$ O(classification séquentielle)**
H(classification séquentielle)	**= H(1)**
H(classification en parallèle)	**$= H(p*(n+pa*n^h))$.**
H(classification séquentielle)	**$\leq$ H(classification en parallèle)**
Temps séquentiel = $T_{séq}$	**$\leq$ Travail en parallèle = $W_{//}$**
$T_{séq} = O(p*(h*n*an)+(n+pa*h^2))$	**$\leq W_{//}(h*an*ct+h*p*(n+pa*n^h))$**

Table 5.4 : Complexités des algorithmes séquentiel et parallèle (dans un modèle CRCW COMMUNE) pour la classification d'une instance

5.4. Le modèle de Circuits.

L'algorithme parallèle de classification peut aussi être modélisé en utilisant le modèle de **circuits** [KAR 92].
Un circuit est un graphe dirigé sans cycle (DAG). Les nœuds ou portes peuvent avoir comme type : **entrée**, **constante**, **et**, **ou**, **non** ou **sortie**. Le "fan-in" des nœuds **entrée** et **constante** est nul. Le "fan-in" de **et** et **ou** est 2, et le "fan-in" de **non** et **sortie** est 1.
La taille d'un circuit, notée H(n), est le nombre de portes; la profondeur du circuit, notée T(n), est le chemin le plus long d'une porte d'entrée à une porte de sortie.
Il existe deux sortes de circuits :

- les circuits bornés, pour lesquels le "fan-in" est borné (une porte peut avoir maximum 2 arètes), et le "fan-out" n'est pas borné.
- les circuits non bornés, pour lesquels le ni "fan-in" ni le "fan-out" ne sont bornés (une porte peut avoir n'importe quel numéro d'arète).

La classification d'instances dans TROPES peut être facilement modélisée en utilisant les circuits (figure 5.11). Les portes dans un circuit représentent les différentes procédures de l'algorithme de classification. Les arètes représentent la façon d'interagir entre ces procédures. Il faut maintenant montrer qu'il n'existe pas de cycle dans le circuit. Dans la définition 6 (§ 1.4), nous avons établi que si une base de connaissances dans TROPES n'est pas redondante, alors elle n'a pas de cycle. Nous avons de plus montré (§ 1.4) que les cycles peuvent conduire à l'inconsistance de la base. Nous supposons donc par la suite que la base de connaissances est consistante et non redondante.

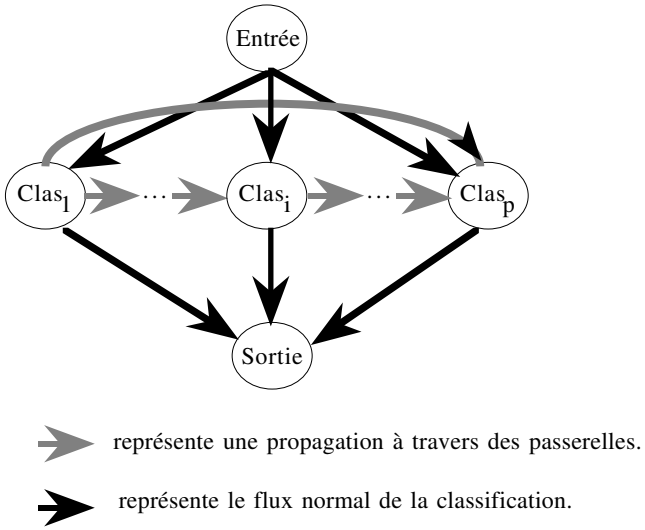


Figure 5.11 : La porte d'**entrée** représente le début de la classification. Les portes nommées **Clas$_i$** représentent la classification de l'instance dans le point de vue i. La porte de **sortie** représente le fait que l'instance a été classée dans tous les points de vue. Le circuit complet représente la classification d'une instance dans un concept.
Une porte du type **Clas$_i$** représente la procédure de classification d'instances dans le point de vue PV_i. Un telle porte est à la fois un circuit, pour lequel chaque porte représente un pas de la classification de l'instance dans PV_i. Le premier pas (pas^i_1) représente la procédure de classification dont la classe courante est la racine du PV_i. Le pas pas^i_j représente la procédure de classification dont la classe courante est la classe sûre de profondeur j du PV_i.
Etant donné que la hauteur maximale d'un point de vue est h, alors la taille de ce circuit est H(h) = h et la profondeur est T(h) = h (figure 5.12).

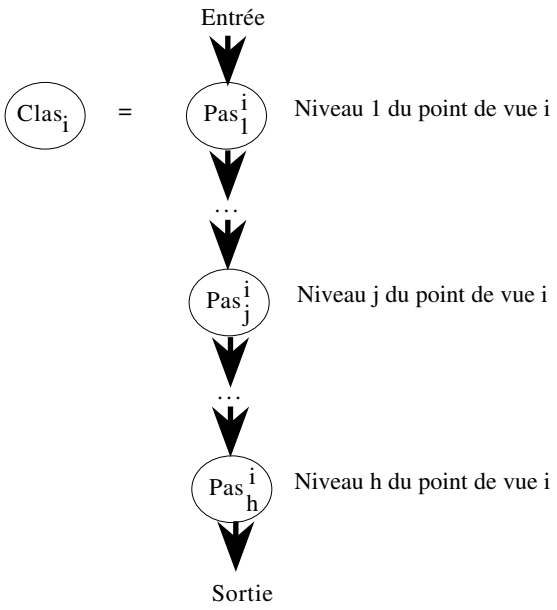


Figure 5.12 : La classification d'une instance pas à pas dans le point de vue PV_i.

Dans le concept, la classification complète de l'instance est visualisée sur la figure 5.13. Pour ce circuit, si le nombre maximum de points de vue est p et la hauteur maximale d'un point de vue est h, alors la taille est $H(p,h) = O(p*h+2) = O(p*h)$, et la profondeur $T(p,h) = O(h+2) = O(h)$.

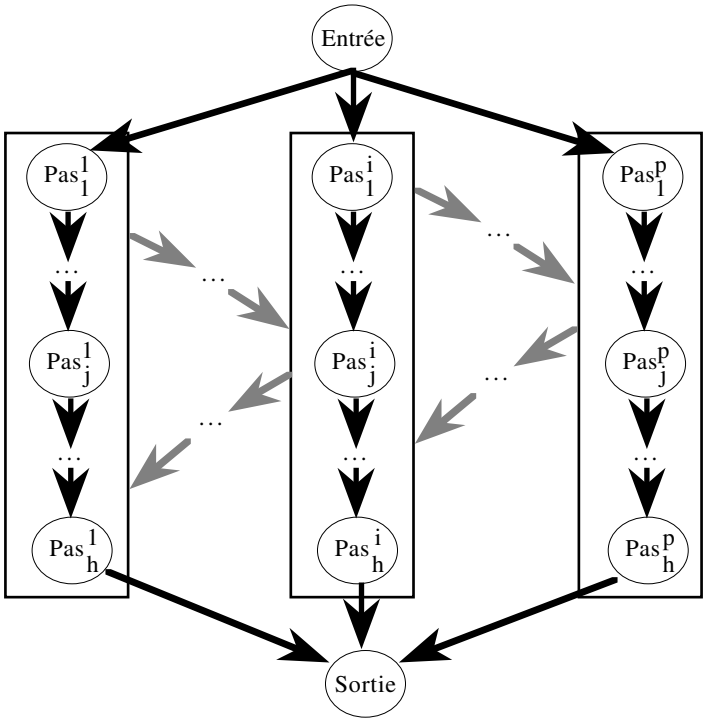


Figure 5.13 : Classification d'une instance dans un concept de TROPES.

Un pas pas^i_j de la classification dans un point de vue est composé de plusieurs procédures : l'obtention d'information, l'appariement, la propagation de marques et la mise à jour. Dans la section 5.2., nous avons calculé le nombre de processeurs logiques qui sont créés lors de la classification pour chacune de ces procédures. L'ensembles de processeurs logiques pour chaque pas est aussi représenté par un circuit (figure 5.14). Soient :

p = nombre maximum de points de vue
n = nombre maximum de sous-classes d'une classe
h = hauteur maximale d'un point de vue.
an = nombre maximum d'attribut d'une classe
ct = nombre maximum de contraintes d'un attribut.

La taille de ce circuit (en partant du niveau j) est $H(p,n,h,an,ct) = O(n*an*ct+n^{h-j}) = O(n^{h-j})$ et la profondeur du circuit est $T(p,n,h,an,ct) = O(4)= O(1)$.

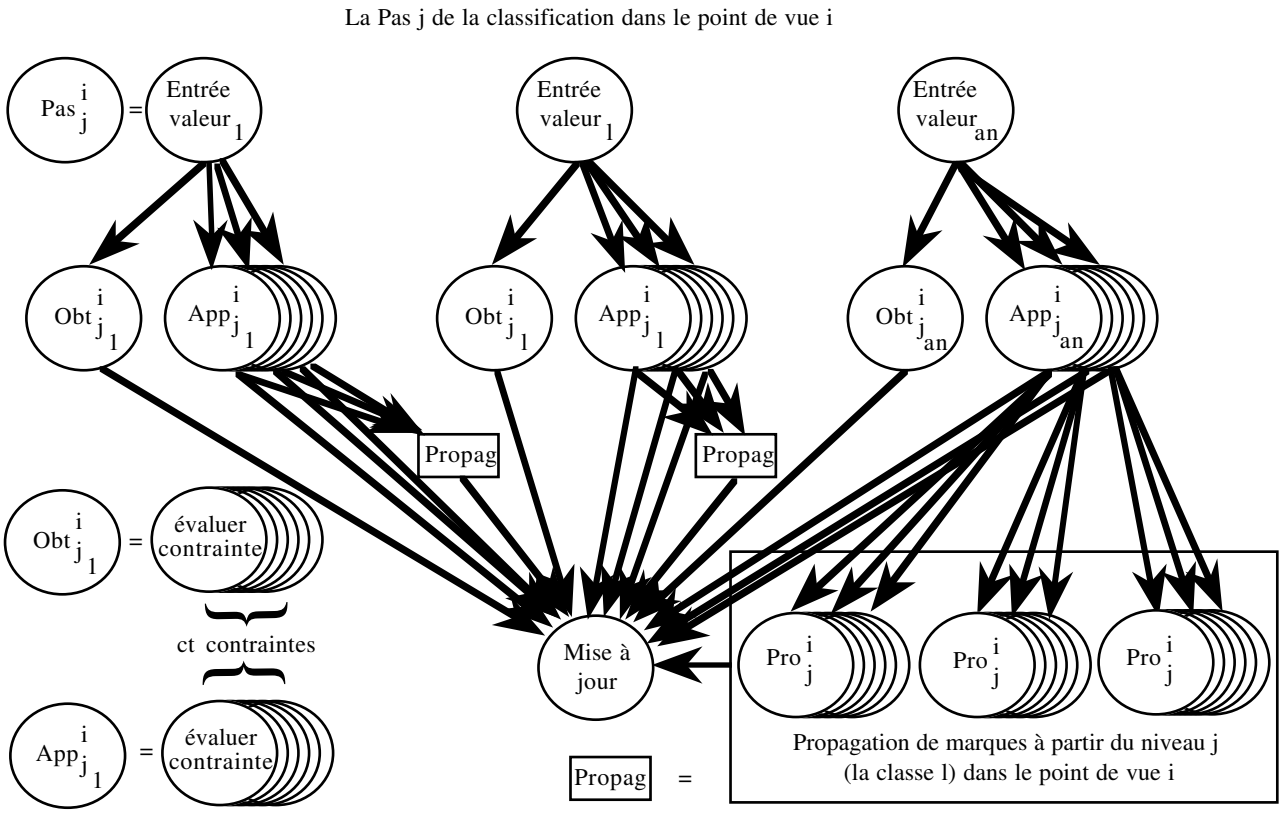


Figure 5.14 : Le pas pas^{i}_{j} représenté comme un circuit. Obj = procédure d'obtention d'information. App = procédure d'appariement. Pro = procédure de propagation de marques. Mise à jour = procédure de la mise à jour de l'instance.

Finalement, le processus complet de la classification d'une instance est représenté comme un circuit (figure 5.15). La taille de ce circuit est $H(p,n,h,an,ct) = O(p*(n*an*ct+n^{h})) = O(p*n^{h})$ et la profondeur du circuit est $T(p,n,h,an,ct) = O(4*h+2) = O(h)$.

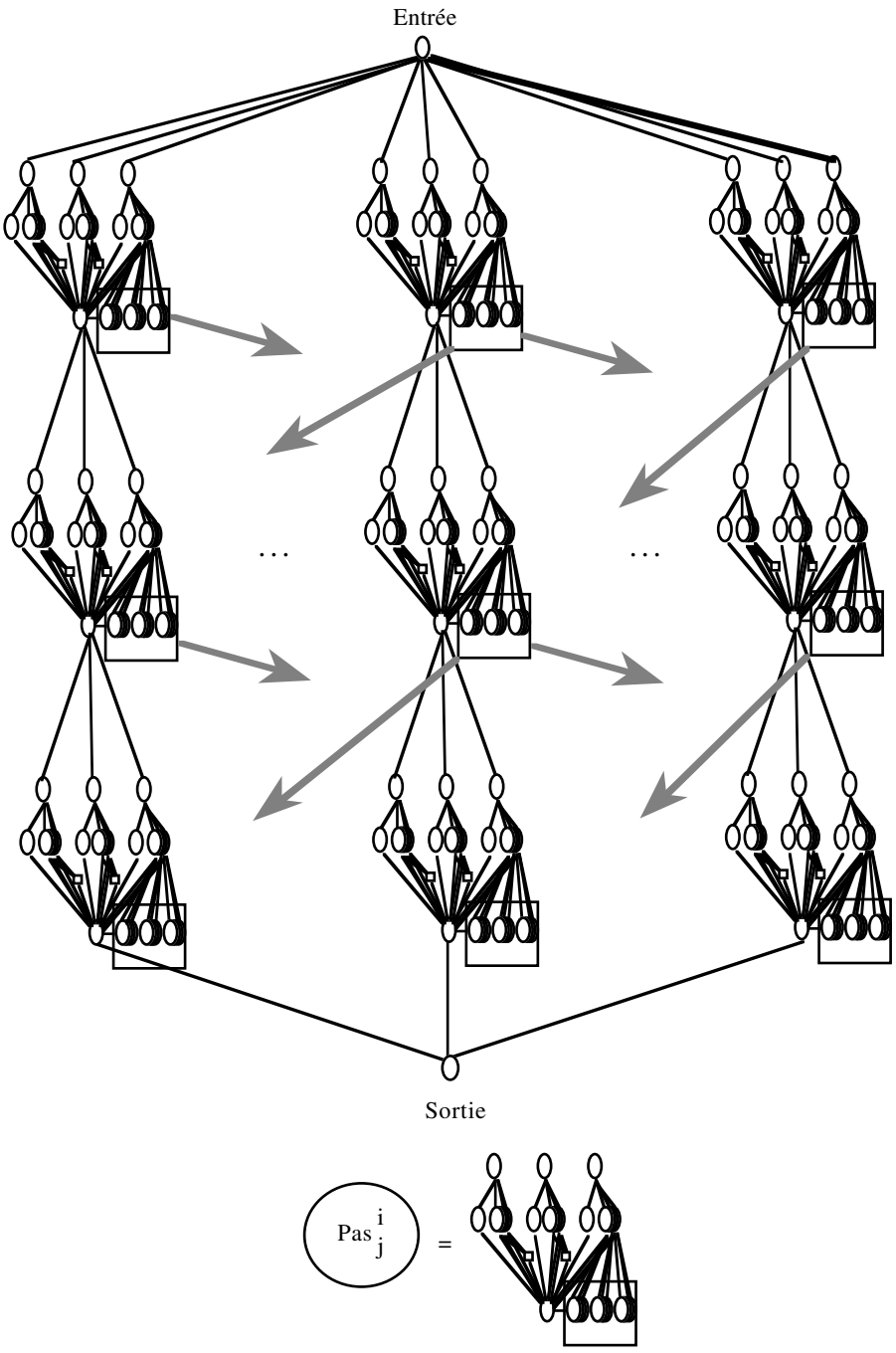


Figure 5.15 : Le circuit représentant la classification d'instances dans TROPES (figures 13 et 14 intégrés). Le pas pas$^{i}_{j}$ représente la procédure de classification dont la classe courante est la classe sûre de profondeur j du PV_i.

Ce circuit appartient aux classes AC^0 et EC^1. AC^K est la classe de tous les problèmes qui peuvent être résolus, avec un circuit non borné, en UCKT(H(n),T(n)) = UCKT(poly(n),O($\log^k$n)), avec poly(n) = $\bigcup_{k\geq 1}$ O(n^k). EC^k est la classe de tous les problèmes qui peuvent être résolus, avec un circuit borné, en CKT(H(n),T(n)) = UCKT(poly(n),O($\log^k$n)), avec poly(n) = $\bigcup_{k\geq 1}$ O(n^k).

5.5 Conclusions

Dans le chapitre 2, nous avons montré que la classification des instances dans TROPES est un problème hautement parallélisable et un algorithme parallèle a été présenté pour cette

classification. Dans ce chapitre une analyse de la complexité du processus de classification d'une instance dans TROPES a été réalisée. La complexité a été présentée pour le modèle séquentiel de classification en utilisant le modèle RAM [AHO 74] et pour le modèle parallèle en utilisant le modèle CRCW COMMUNE PRAM [KAR 90]. Après l'analyse de la complexité des deux modèles, nous pouvons conclure que l'algorithme parallèle réduit de manière significative la complexité du processus de classification par rapport à celle-ci du processus séquentiel. Pour le dernier cas, la complexité est une fonction du nombre de points de vue du concept, du nombre de sous-classes d'une classe, du nombre d'attributs définis dans une classe, du nombre de contraintes définies pour un attribut et de la hauteur de l'arbre de classes. Pour l'algorithme parallèle, cette complexité est une fonction qui ne dépend que de la hauteur du point de vue. Néanmoins, pour la mise en œuvre de l'algorithme parallèle, le modèle EREW PRAM (ce modèle est le plus commun dans les systèmes répartis [SAN 90]), avec un coût de **log(nombre de processeurs)** pour cette simulation, a été utilisé.

6. MISE EN ŒUVRE DE L'ALGORITHME PARALLELE DE CLASSIFICATION

Dans cette chapitre nous présentons la mise en œuvre de l'algorithme précédent que nous avons développé. Pour cette mise en œuvre, nous devons résoudre quelques problèmes concernant l'architecture à utiliser, les méthodes de synchronisation des processeurs (ex. pendant la phase de propagation de marques), le stockage de données, etc. Une présentation générale de ces problèmes va être fournie, ainsi qu'une solution qui s'adapte à notre problème. Finalement, nous allons donner le modèle pour le placement des processus et la communication entre eux pour l'algorithme, utilisé pour le mise en œuvre de la classification en parallèle dans le modèle TROPES.

6.1. Idée générale

Devant le nombre croissant de machines parallèles disponibles, il semble que ce type d'architectures représente l'avenir. Tout au moins, pouvons nous penser que ce type de machine risque de se généraliser. Il est donc important de tenter d'en utiliser les ressources spécifiques. Même si ces machines ne sont pas immédiatement abordables sous leur forme la plus attrayante, il est intéressant d'examiner ce qu'elles peuvent apporter pour notre problème spécifique. Une telle approche va permettre, du même coup, de mettre en évidence nos besoins tant matériels que logiciels.

L'étude qui suit va donc tout d'abord considérer un parallélisme «idéal» signalant les options les mieux adaptées à la classification dans un modèle de connaissances multi-points-de-vue, avant de présenter un algorithme qui semble dès aujourd'hui utilisable.

Au début de ce chapitre, nous avons déjà montré la possibilité de paralléliser l'algorithme de classification. Pour cela, nous avons décidé quelles sont les procédures susceptibles d'être parallélisées.

Dans la suite, nous allons étudier les problèmes du placement de l'algorithme sur une architecture parallèle et de la communication entre les processus afin de trouver la solution qui s'adapte au modèle parallèle.

Pour faire l'étude du problème de placement du processus et de communication entre eux, nous définirons d'abord le nombre de processus qu'il faut créer et la façon dont les processus interagissent entre eux. Nous présentons ici l'analyse du nombre de processus créés pendant la classification d'un objet dans un concept.

6.2. Analyse du nombre de processus et de la communication.

Pour exécuter une application sur une machine parallèle, l'étape de traduction du programme source écrit dans un langage de haut niveau en un code binaire utilisée dans les machines séquentielles n'est pas suffisante. Avant toute exécution, il est nécessaire de savoir placer les processus composant les programmes sur les divers processeurs. Le but de ce placement est de permettre au programme de s'exécuter le plus rapidement possible. Pour cela, plusieurs critères peuvent prévaloir comme par exemple : la longueur des chemins de communication, l'équilibrage de la charge de travail des processeurs et des liens de communication, etc. Dans la suite, nous allons présenter ces critères pour le placement de processus sur une architecture parallèle et puis nous montrerons comment s'appliquent ces critères à notre problème.

D'abord, nous allons définir le problème du placement (pris de [TAL 91]) :

Soit un système parallèle formé d'un ensemble T de n processeurs $\{T_1, ..., T_n\}$ qui communiquent entre eux par l'intermédiaire d'un réseau d'interconnexion statique ou reconfigurable. Soit un programme composé d'un ensemble P de k processus $\{p_1, ..., p_k\}$ communicant entre eux. Le problème d'allocation est décrit formellement par une fonction **F** de l'ensemble des processus vers l'ensemble des processeurs $F:P \rightarrow T$. Il existe n^k placements possibles. Nous montrerons un placement optimal pour notre problème sur une architecture donnée. En général, le processus d'optimisation consiste en la recherche d'une minimisation du temps d'exécution du programme [HIl 86][KAR 87].

Pour faire le placement du processus correctement, nous devons prendre en compte le type d'architecture, le contrôle, le coût de la communication, le stockage des données et bien sûr les algorithmes possibles pour faire le placement.

6.2.1. L'architecture

Le développement de l'algorithme parallèle pour notre problème entraîne le choix de l'architecture parallèle. Elle peut être fixe ou reconfigurable. Dans le cas d'une architecture fixe, divers types d'architecture parallèle ont été avancés (hypercube [LAX 84], pyramides) et dans le cas des architectures reconfigurables, divers types ont été avancés (le transputer [INM 88], le supernode [WAI 90]). Il est intéressant d'étudier l'adaptation de ces architectures au problème posé. Par la suite, nous allons aborder le principe général de ces architectures [HOC 84]. Il existe également des architectures spécialisées pour l'intelligence artificielle [HWA 87][HWA 86].
Les architectures parallèles peuvent reposer sur deux principes: la concurrence et le traitement à la chaîne, appelé aussi "pipeline". La concurrence intervient lorsque des opérations sont effectuées simultanément par des processeurs différents. Le pipeline correspond au cas où des suites des données subissent, en circulant d'un processeur à un autre, une même succession de traitements élémentaires. Dans une exécution concurrente, les processeurs sont indépendants, alors que dans le pipeline, ils coopèrent pour produire des résultats corrects.
La concurrence dans une architecture donnée peut être mise en œuvre de deux manières. Si les processeurs exécutent leur programme de manière totalement indépendante, on obtient une architecture de type MIMD (Annexe 3). Pour le cas où les processeurs ont de la mémoire partagée, il existe quatre sous-classes pour les ordinateurs MIMD (§ 5.2), par rapport à leur façon d'accéder à la mémoire [AKL 89].
L'architecture MIMD représente a priori la forme la plus souple d'architecture parallèle, car les processeurs peuvent exécuter des programmes distincts ou non, sur les données qui leur sont propres. Nous disons que ces processeurs sont asynchrones. Le principal problème posé par une telle structure est celui de l'échange d'informations entre les processeurs, origine de pertes de temps parfois considérables.
Une autre solution consiste à faire exécuter la même instruction d'un programme unique par tous les processeurs de manière synchrone, mais sur des données différentes. Nous obtenons alors l'architecture SIMD. Il existe quatre sous-classes pour les ordinateurs SIMD, par rapport à la façon d'accéder la mémoire : EREW, CREW, ERCW et CRCW, et pour ce dernier, il existe aussi les trois modèles : COMMUNE, ARBITRAIRE et PRORITAIRE.
Par rapport à l'architecture MIMD, nous y gagnons en simplicité. Une seule unité de contrôle et une seule mémoire de programme sont nécessaires pour l'ensemble des processeurs. En outre, ceux-ci ne perdent jamais de temps à s'attendre mutuellement puisqu'ils ont un mode de fonctionnement synchrone [QUI 85].
Or, comme nous l'avons déjà dit, à chaque nouveau pas de la classification, parfois il faut marquer en parallèle les différents classes comme impossibles, ou parfois il faut faire l'appariement d'un objet avec les sous-classes de la classe actuelle en parallèle, et en plus ces pas sont faits en parallèle pour les divers points de vue. Donc, ce programme est composé d'un ensemble de processus différents {p1, ..., pn} (n étant le nombre de processeurs), qui s'exécutent en parallèle sur un ensemble de données D1,…Dm quelconques (m étant le nombre du points de vue du concept). Le problème est donc fort simple dans sa formulation et semble bien s'adapter à une programmation de type "multiple instructions-multiple data" avec mémoire partagée et dont la lecture et écriture sont concurrentes. Pour résoudre l'écriture simultané le modèle commun est le mieux adapté (MIMD CRCW COMMUNE).
D'autre parte, il existe des architectures parallèles SIMD, qui ont été largement inspirées des problèmes de représentation des connaissances, et plus spécifiquement des réseaux sémantiques, comme par exemple la Connection Machine [TUC 88] et la machine SNAP (A marker-propagation architecture for knowledge processing) [MOL 92]. Un réseau sémantique est un système à héritage simple ou multiple, fondé généralement sur la théorie des prototypes. Il est constitué d'un ensemble des nœuds typés, qui représentent les concepts de l'univers de référence, et d'un ensemble d'arcs orientés étiquetés, ou liens, qui figurent les relations entre les nœuds. Un ensemble d'opérations associé au réseau permet de l'exploiter : création et suppression de nœuds et d'arcs, parcours du graphe des relations, mécanismes d'héritage et de filtrage, etc. [KAY 86].

6.2.2. Les configurations

Dans ce paragraphe, nous allons présenter quelques configurations d'architectures parallèles. La plupart d'entre elles sont statiques, car leur structure est figée lors de la construction (les processeurs sont reliés entre eux de manière non reconfigurable au cours de l'exécution d'un programme; exemples: hypercube, machine d'arbres, pipeline). Il existe une autre forme de

réseau, qualifiée de dynamique ou reconfigurable. Les processeurs ne sont pas alors reliés directement, mais par l'intermédiaire de commutateurs que nous pouvons positionner par programme (où le réseau d'interconnexion entre les processeurs peut varier en fonction de la structure du programme parallèle exécuté). Ces derniers permettent en général une plus grande souplesse dans le mode d'interconnexion[13] que les réseaux statiques. Les réseaux reconfigurables sont aussi bien adaptés aux architectures SIMD, soit pour connecter les processeurs à la mémoire, soit pour connecter les processeurs entre eux. Finalement, il existe des systèmes dits à large granularité, où un petit nombre de processeurs communiquent à travers un réseau local (ethernet, token ring, ...).

6.2.2.1. L'hypercube

Un hypercube de dimension N consiste en 2^N processeurs (ou nœuds) identiques qui sont liés de telle façon que chaque nœud a N voisins. Nous disons que deux nœuds sont voisins si et seulement si leur numérotation binaire diffère d'un et un seul bit (code de Gray). Cette codification permet également de réaliser un anneau sur l'hypercube.

Ainsi, un hypercube de dimension 3 peut être représenté comme un cube de dimension 3 (figure 6.1). La structure d'interconnexion correspondante est obtenue en plaçant un processeur à chaque sommet du cube.

G(3) = { 000 , 001 , 011 , 010 , 110 , 111 , 101 , 100 }

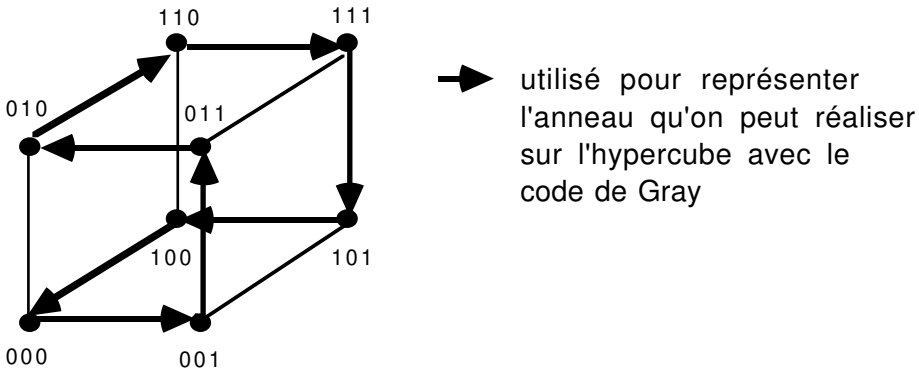


Figure 6.1 : Hypercube de dimension 3

---

[13] Les réseaux d'interconnexion ne sont pas explicitement utilisés dans les caractérisation des architectures parallèles en vue d'une classification. Le réseau d'interconnexion définit cependant la structure spécifique d'un ordinateur; c'est lui qui caractérise les facilités offertes pour la communication entre les processeurs, et sa participation dans l'activité globale de la machine est si fondamentale qu'il a une influence cruciale sur ses performances.
Un ensemble de processeurs peut être interconnecté soit par l'intermédiaire d'un bus, soit par des liaisons point à point. Il n'existe pas la notion de distance entre processeurs, cependant la bande passante disponible pour chaque processeur diminue fortement au fur et à mesure que le nombre de processeurs augmente.
Dans le cas d'une interconnexion point à point, les processeurs qui sont directement connectés communiquent efficacement car ils peuvent profiter de toute la bande passante offerte par la liaison physique; cependant, l'efficacité diminue lorsque les processeurs communicants ne sont pas directement connecté. En général la bande passante dépend de la distance qui sépare les processeurs, mesurée en nombre de liens à traverser. La notion de distance caractérise le comportement de ce genre de réseau.
Du point de vue de la distance entre processeurs, le réseau le plus performant est celui dans lequel tous les processeurs sont directement reliés d'une façon permanente par un lien de communication; c'est le réseau complètement connecté.
Pour les architectures massivement parallèles, la minimisation de la distance par une maximisation de la connectivité permanente se heurte à l'impossibilité de réalisation pratique. Deux approches se dégagent : l'une consiste à connecter les processeurs à travers un réseau statique, sous une structure régulière qui garantit certaines propriétés telles que : faible diamètre, simplicité du routage des messages et facilité d'expansion. L'autre se base sur des réseaux dits reconfigurables, dans lesquels la distance entre les processeurs communicants peut être modulée par modification dynamique de la connexion du réseau.

Un avantage des hypercubes est la transmission de données entre deux nœuds quelconques, puisque la longueur du chemin le plus court est au plus N pour un hypercube de dimension N (2^N nœuds). Cette topologie est présente dans plusieurs architectures MIMD sans mémoire commune, par exemple, le Cosmic cube [SEI 85], les iPSC, le FPS/T et le NUBE/10 [HAE 86].

6.2.2.2. La grille

La topologie d'interconnexion la plus simple est la grille unidimensionnelle, où les processeurs sont disposés sur une ligne et chacun est connecté à ses voisins directs. Un anneau est formé lorsque tous les processeurs ont deux connexions.

Il se construit des grilles multi-dimensionnelles sur le même principe. La grille la plus utilisée est celle à deux dimensions, car elle présente un bon compromis entre la complexité de réalisation et la connectivité; les processeurs se trouvant à l'intérieur de la grille ont quatre voisins et ceux se trouvant dans la périphérie en ont trois. Le nombre maximum de liens par processeur est de quatre pour n'importe quelle taille du réseau, par conséquent, leur coût d'expansion est réduit.

6.2.2.3. Le tore

Le tore reprend le principe de la grille bi-dimensionnelle, mais en plus chaque colonne de la grille est en forme d'anneau. Le tore est utilisé pour les algorithmes de clôture transitive. La raison en est que la plus courte longueur des chemins de communication entre deux nœuds est au plus N, si l'architecture est composée de N^2 nœuds (figure 6.2).

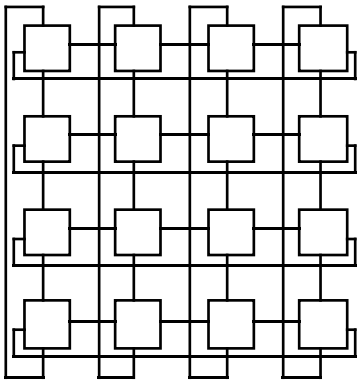


Figure 6.2 : La configuration de TORE.

6.2.2.4. Le supernode

Le supernode est une architecture multi-processeurs modulaire et hiérarchique sans mémoire commune; un module de base peut contenir jusqu'à 32 processeurs T800. Les processeurs d'un module sont interconnectés à travers un commutateur programmable. Un certain nombre de liens de communication peuvent, éventuellement, être utilisés pour des connexions vers l'extérieur du module [WAI 90] [LAN 91].

6.2.2.5. Le bus

Cette topologie est peut-être la plus simple. Tous les processeurs sont réliés entre eux par un bus commun (figure 6.3). La communication entre deux processeurs quelconques se fait à travers un réseau à diffusion.

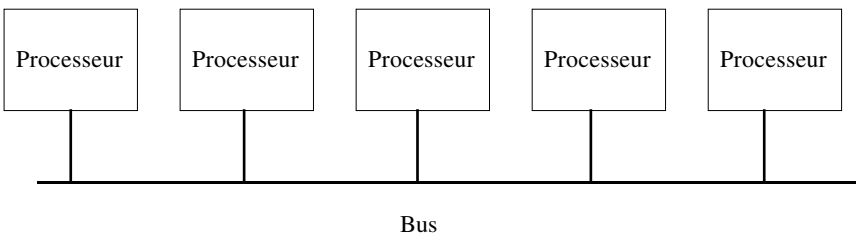


Figure 6.3 : La configuration de bus.

Pour la classification en parallèle dans TROPES en utilisant le modèle de tableau noir, la configuration qui s'adapte le mieux est la configuration de bus. En effet, les coûts de communication peuvent être ignorés, puisque pour des réseaux à diffusion la distance entre processeurs n'intervient pas dans le choix du processeur pour l'exécution d'un processus. Le seul critère pris en compte est la charge des divers processeurs. Donc, pour ce cas, le placement aléatoire peut donner de bons résultats pratiques au vu de sa simplicité de mise en œuvre [ATH 87]. Un exemple de ce type de communication est : le coût de communication de la propagation d'une marque à travers une passerelle entre deux points de vue. Ceci correspond à une communication entre deux processeurs logiques, à travers le tableau noir, qui font la propagation de marques chacun sur un point de vue différent.

6.2.3. La terminaison

Dans cette section nous présentons un protocole pour résoudre le problème de la détection de la terminaison dans une architecture parallèle dans laquelle tous les processeurs sont asynchrones [ERI 88]. Pour la solution nous supposons que tous les processeurs sont programmés de façon identique, et que la taille de l'architecture (nombre de processeurs) est connue par tous les nœuds.

Nous pouvons modéliser l'interconnexion des processeurs par un graphe orienté, où les sommets sont des processeurs et les arcs sont les liens entre eux. La communication est asynchrone quand le processeur qui envoie le message n'attend pas la réception du message par l'autre processeur pour poursuivre son travail.

Par la suite, nous allons supposer que des liens entre les processeurs sont symétriques, c'est-à-dire, s'il existe un lien entre le processeur P et le processeur Q, il en existe aussi un entre Q et P. Ce type d'interconnexion peut être modélisé au moyen d'un graphe non-orienté.

Les processeurs peuvent être dans un seul des états suivants: actif, passif ou terminé. Actif veut dire que le processeur est en train d'exécuter une tâche, passif veut dire que le processeur est libre et que nous ne lui avons pas assigné une tâche, terminé veut dire que le processeur a fini la tâche qui lui a été assignée. Il y a aussi deux types de messages : contrôle et activation. Les messages de contrôle doivent suivre les lignes de communication, et les messages d'activation peuvent être envoyés entre deux processeurs quelconques.

Dans notre cas, il existe trois raisons pour terminer la classification d'un objet dans TROPES :

- Le but est atteint : l'objet est classé le plus bas possible dans chacun des points de vue. Ceci est reconnu par le fait que tous les points de vue sont inactifs, leur plus petite classe sûre est terminale.
- L'algorithme ne trouve plus de chemins à suivre.
- L'utilisateur veut arrêter la classification.

A partir de ces raisons, le problème de terminaison est simple. De plus, le contrôle de la classification est d'une certaine façon centralisé par le contrôle du tableau noir. Si une de ces raisons devient vraie, la classification est arrêtée. Pour la première cause, le contrôle du tableau noir lorsqu'il s'aperçoit qu'il n'existe aucun point de vue actif ne déclenche plus les sources de connaissances. Pour la deuxième cause, le contrôle, lorsqu'il s'aperçoit qu'il n'y a plus d'événements dans le tableau, arrête donc la classification. Pour la dernière cause, un événement spécial se produit, et lorsqu'il est perçu par le contrôle, celui-ci arrête la classification.

6.2.4. La communication

Le développement de l'algorithme parallèle pour notre problème nous conduit à choisir la façon de faire communiquer les processeurs. Ceci peut se faire de trois façons différentes :

- Centralisée, où un processeur superviseur reçoit tous les messages et se charge de les faire parvenir à leur destinataire.
- A travers une mémoire commune.
- Distribuée, où chaque processeur fait parvenir son message à son destinataire [ATH 88]. Nous faisons ici abstraction du routage des messages.

Etant donnée la structure de notre problème, et en partant de l'analogie faite entre le modèle parallèle pour le problème de classification et le modèle tableau noir (§ chapitre 4), les deux dernières façons de communiquer sembleraient s'imposer. La deuxième cause de transmission est valable étant donné que la tableau noir est une structure de données partagée par les

différentes sources de connaissances. La dernière raison est aussi valable car le tableau noir est composé de plusieurs panneaux (les points de vue) et chacune des différentes sources de connaissances doit faire parvenir ses messages au respectif panneau. Nous reviendrons sur ce problème plus tard, lorsque nous analyserons le graphe des processus et leurs liens.
Il est aussi très important de réduire la surcharge dans les communications [BAI 87], donc le placement de processus par rapport aux données sera fait de façon telle que nous puissions minimiser les coûts de communication.

6.2.5. Le coût de transmission

Dans ce paragraphe, nous allons citer quelques exemples du coût de transmission dans différentes architectures parallèles [AND 88].

- Pour une architecture à processeurs identiques et identiquement reliés, la minimisation de la fonction C_0 permet la minimisation du temps total de communication :

  $$C_0 = \sum_{i=1,j=1,j\neq i}^{m} C_{ij}\ Y_{ij} \quad \textbf{(1)},$$

  avec :

  C_{ij} : coût de communication entre les processeurs i et j

  Y_{ij} : 1 si i et j sont sur des processeurs différents

  0 si i et j sont sur le même processeur

  m : nombre de processeurs.

- Pour une architecture à processeurs identiques pour laquelle les relations inter-processeurs sont différentes, nous pouvons minimiser la fonction C_1 pour minimiser le temps total de communication:

  $$C_1 = \sum_{l=1,q=1,q\neq l}^{n} \sum_{i=1,\ j=1,\ j\neq i}^{m} C_{ij}\ D_{lq}\ X_{il}\ X_{jq} \quad \textbf{(2)}$$

  avec i,j qui appartiennent à l'ensemble des processeurs,

  n : nombre de processeurs,

  m : nombre de processus dans chaque processeur,

  D_{lq} : coût de communication unitaire sur le lien entre les processeurs l et q,

  C_{ij} : quantité de communication entre les processeurs i et j,

  X_{il} : 1 si le processus i est sur le processeur l

  0 sinon,

  X_{jq} : 1 si le processus j est sur le processeur q

  0 sinon.

Nous avons déjà dit au début de cette section que le but du placement est de permettre au programme de s'exécuter le plus rapidement possible. Cela dépend du temps d'exécution des processus (nous avons déjà faite l'analyse de complexité du modèle parallèle) et des temps de communication entre processus. Dans notre cas, nous devons minimiser la fonction $\mathbf{C_0}$.

Nous verrons, dans la mise en œuvre du système, que chaque panneau est placé sur un processeur différent, et que tous les processus liés à ce panneau sont eux aussi placés dans le même processeur. Donc, le coût de communication est réduit à la communication entre les panneaux en raison de la procédure de propagation de marques à travers des passerelles.
Dans notre cas, il existe différents besoins de communication : par exemple, entre les processus de propagation de marques, entre les processus d'obtention d'information et le contrôle de

dialogue, etc. Nous reviendrons sur ce problème lorsque nous analyserons le graphe des processus et leurs liens.

6.2.6. Le placement des tâches

Le développement de l'algorithme parallèle pour notre problème nous conduit également à choisir la façon de faire l'allocation des tâches sur les processeurs. L'allocation peut être [TAL 91][AND 88] :

- Une allocation statique, où le placement est déterminé une fois pour toutes avant l'exécution. Celle-ci se déroule suivant le plan prévu. La plupart des stratégies d'allocation statique sont basées sur l'une des approches suivantes :
  - la théorie des graphes,
  - la programmation mathématique,
  - l'utilisation d'heuristiques.
- Une allocation dynamique centralisée, où un processeur est chargé de procéder au placement des tâches au fur et à mesure de leur exécution.
- Une allocation dynamique distribuée, où chaque processeur, à la fin du programme P exécuté sur les données Di décide sur quelles données Dj va s'exécuter.

Pour le problème de classification, il est clair que la solution qui s'adapte le mieux est une allocation dynamique centralisée. Dans ce cas, le processeur qui est chargé du placement de tâches au fur et à mesure de leur exécution est le contrôle du tableau noir. Néanmoins, au départ, nous pouvons utiliser une allocation statique pour les procédures de contrôle et les panneaux. Puis, nous utiliserons l'allocation dynamique centralisée.

Donc, une tâche fondamentale pour l'implémentation d'un algorithme parallèle est le placement des processus et des chemins de communication dans l'algorithme sur les processeurs et les chemins de la machine. En général, il existe deux problèmes : le premier, quand le nombre de processus qui peuvent être créés par l'algorithme parallèle est plus grand que le nombre de processeurs de la machine ("cardinality variation"); le second problème apparaît quand la structure de communication de l'algorithme parallèle est différente de celle de la machine ("topological variation").

L'adaptation la plus simple entre une architecture et un algorithme est évidemment l'identification de l'une avec l'autre.

Trouver une solution générale n'est pas un travail facile, puisque l'implémentation d'un algorithme composé de N processus qui communiquent entre eux, sur une architecture qui a M processeurs, est un problème NP-complet et il existe alors M^N cas différents.

Pour résoudre ce problème, il existe plusieurs modèles. Ici, nous allons décrire une solution proposée par Berman [BER 87]. La raison de présenter ce modèle est qu'il utilise des graphes pour représenter l'algorithme ainsi que pour représenter l'architecture, et puisque notre problème (la parallélisation de la propagation) est représenté par un graphe, le modèle peut être facilement utilisé pour résoudre le problème de la communication.

Le premier pas dans le modèle de Berman consiste à faire l'abstraction de l'architecture parallèle et de l'algorithme.

L'algorithme peut être modélisé comme un graphe de communication statique, dans lequel les sommets représentent les processus et les liens représentent le flux des données entre les processus. Comme en réalité un algorithme parallèle est composé d'une ensemble d'algorithmes, un pour chaque instance du problème, nous utilisons une famille de graphes de communication {Gn} pour représenter l'algorithme. Nous pouvons voir dans la (figure 6.4) un exemple d'un graphe de processus.

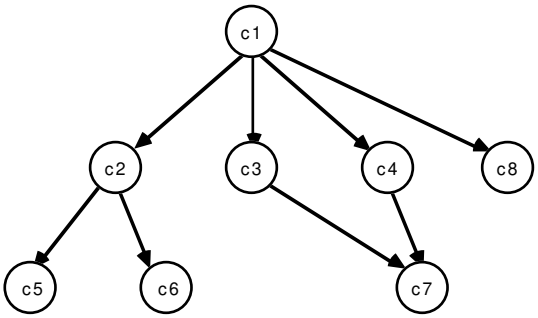


Figure 6.4 : Graphe de processus.

Pour modéliser une architecture sans mémoire partagée, nous pouvons aussi en général utiliser un graphe. Comme nous l'avons déjà dit, la communication entre les processeurs étant un des principaux rôles, nous devons être sûrs que les modèles des graphes ont la puissance pour représenter la structure de communication de l'architecture (interconnexion). Par exemple, pour représenter une maille (MESH), nous pouvons utiliser un graphe non-orienté (figure 6.5), où chaque sommet représente un processeur et chaque arc un lien entre deux processeurs.

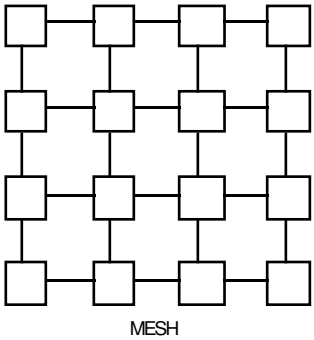


Figure 6.5 : Graphe de processeurs : MESH

Un algorithme pour implémenter une instance Gn de l'algorithme parallèle sur une architecture A modélisé par le graphe Ga est le suivant:

1) Le premier pas permet d'éliminer la variation de la cardinalité. On va supposer qu'il n'existe pas de variation topologique. Pour éliminer la variation de la cardinalité, il suffit de réduire le graphe Gn en un graphe plus petit Gk (d'un nombre moindre de sommets) de la même famille.
2) Placement de Gk sur l'architecture A (sans variation de la cardinalité). Par exemple, associer au plus un processus du graphe Gk avec un processeur.
3) Pour faire l'implémentation de l'algorithme Gn, il suffit de prendre l'image du graphe modifié Gk en Ga par multiplexage.
   Exemple: Nous allons appliquer l'algorithme au graphe de processus de la figure 6.4 et un hypercube de dimension 2 comme dans la figure 6.6.

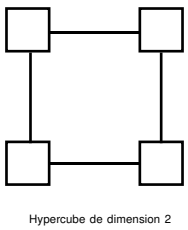


Figure 6.6 : Hypercube de dimension 2.

Si nous appliquons le premier pas de l'algorithme ci-dessus, nous devons réduire le graphe à un autre graphe de la même famille (graphes de dépendance) dont le nombre de sommets est au plus le nombre de processeurs de l'architecture. Dans notre cas, l'architecture a 4 processeurs, alors nous allons réduire le graphe à un autre graphe avec 4 sommets par regroupement (figure 6.7). Nous pouvons voir le graphe réduit dans la figure 6.8.

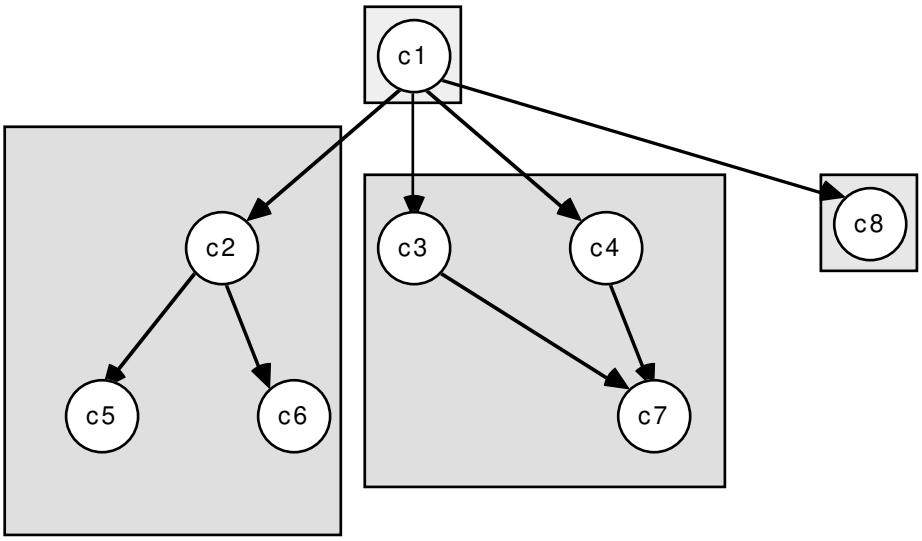


Figure 6.7 : Regroupement de processus

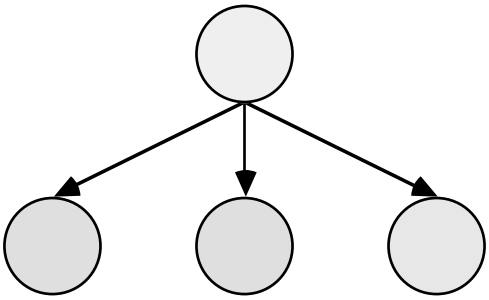
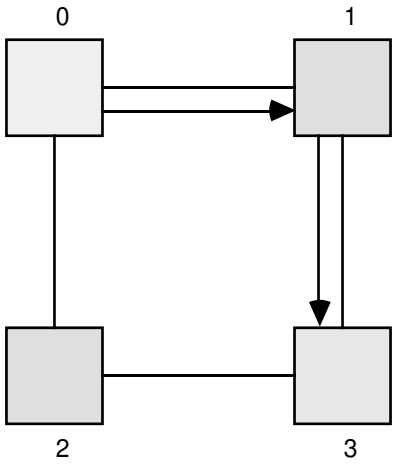


Figure 6.8 : Graphe de processus réduit

Une fois le graphe de processus réduit (figure 6.8), nous pouvons le placer dans l'architecture, à laquelle nous devons ajouter de nouveaux routages, comme dans notre cas (chemin entre le nœud 1 et 3 dans l'hypercube) (figure 6.9).



Hypercube de dimension 2

(la flèche représente le chemin entre 0 et 3)

Figure 6.9 : Hypercube de dimension 2.

6.2.7. Le stockage de données

Le développement de l'algorithme parallèle pour notre problème entraîne le choix de la façon de stocker les données dans la mémoire. Ce stockage peut être:

- Centralisé : toutes les données se trouvent dans une mémoire accessible à tous les processeurs.
- Distribué : chaque processeur possède les données qui le concernent.

La première solution pose un problème d'étranglement des communications pour l'accès à la mémoire. Il faudra étudier l'adaptation de ces types de mémorisation à notre problème.
La meilleure solution de stockage des données pour notre problème est une combinaison des deux méthodes : avoir une mémoire partagée et une mémoire distribuée. Dans le cas d'une mémoire partagée, lorsqu'un processeur a besoin de certaines données, il peut y accéder facilement sans le demander à aucun autre processeur. De plus, dans le cas de la mise à jour des structures de données, si nous utilisons des procédés d'exclusion mutuelle pour résoudre les problèmes de collision, une fois que le processeur a fini cette mise à jour, tous les autres processeurs peuvent accéder aux données sans risque d'avoir des données incorrectes [MAN 85]. C'est le cas pour les processus d'appariement de propagation de marques dans un même point de vue. Nous avons déjà analysé ce partage d'information. Pour ces processus, le tableau noir est unique et centralisé (§ 4) et il représente un panneau du tableau noir.
Dans le cas d'un stockage distribué de données, il est nécessaire pour les processus de propagation de marques, lorsqu'une passerelle devient impossible ou sûre, d'effectuer la propagation dans un autre point de vue. Il faut se souvenir que chaque point de vue constitue un panneau du tableau noir et que, pour chacun d'eux, il existe une source de connaissances (dans notre cas le processus de propagation de marques) qui est placée sur un processeur différent. Pour chaque panneau il existe plusieurs processus qui font la classification dans celui-ci (le point de vue). Tous ces processus partagent la mémoire du panneau, il faut donc disposer de mécanismes de synchronisation pour ne pas perdre la cohérence du point de vue [SHA 88].

6.3. La mise en œuvre de l'algorithme parallèle de classification en utilisant l'analogie TROPES - Tableau noir

Dans cette section, nous allons montrer la mise en œuvre de, l'algorithme parallèle de classification, sur un système réparti où les processeurs peuvent être égaux ou différents.
L'idée est de prendre le super-graphe de processus pour lequel nous allons faire le placement sur une architecture parallèle ou système réparti et d'appliquer l'algorithme de Berman [BER 87]. Pour cela, nous allons représenter l'architecture cible par un graphe connexe, non orienté Gm = (Vm,Lm), où Vm l'ensemble des sommets désigne les processeurs, et Lm l'ensemble des arètes représente les liens physiques. Ce graphe est non orienté car les liaisons entre processeurs sont supposées être bidirectionelles.

Le programme de la classification sera lui aussi représenté par un graphe connexe orienté Gp=(Vp,Lp) où Vp l'ensemble des sommets représente les processus (les sources de connaissances), et Lp l'ensemble des arètes représente les liens de communication entre les processus. Dans notre cas, le graphe est connexe, puisque pour le modèle tableau noir, toutes les sources de connaissances (les processus de l'algorithme de classification) font un travail similaire pour arriver au même but : la classification de l'objet le plus bas possible dans l'arbre de classes de chaque point de vue. En général, les arètes sont pondérées par les coûts de communication entre les processus, et les sommets par les temps d'exécution des processus.
Dans la suite, nous allons montrer la représentation de la classification d'un objet dans TROPES par un graphe (figure 6.10).

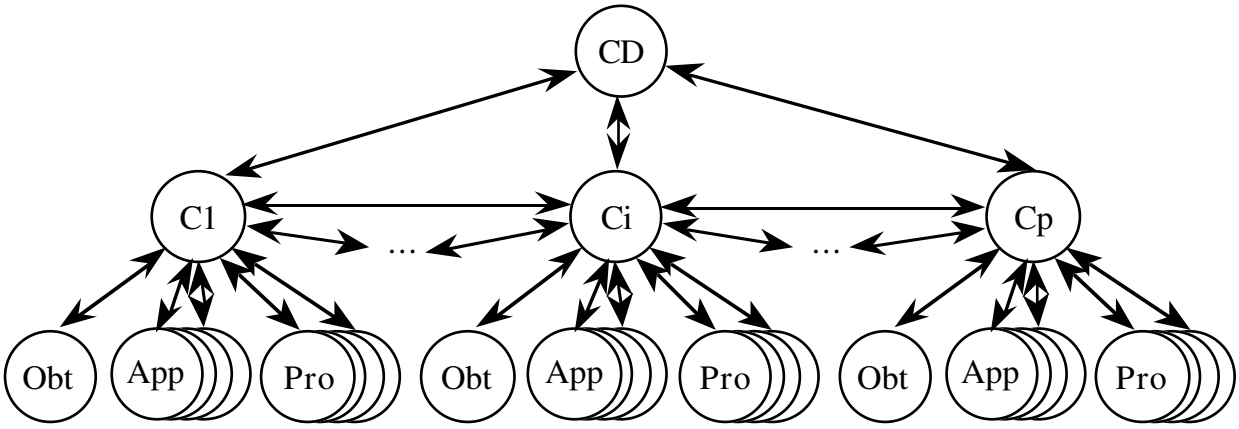


Figure 6.10 : Représentation de la classification d'un objet par un graphe orienté. Les arètes représentent les liens de communication entre les différents processus.
CD : Le contrôle de dialogue
Ci : Une instance du contrôle du tableau noir dont le panneau est le point de vue PVi.
Obt : Une instance de la procédure d'obtention d'information.
App : Une instance de la procédure d'appariement.
Pro : Une instance de la procédure de propagation de marques.

Pour cela, nous allons partir du modèle séquentiel (figure 6.11), dans lequel les différentes procédures de l'algorithme de classification sont représentées par les sources de connaissances, chaque point de vue est un panneau du tableau noir et le contrôle du tableau noir interagit avec le contrôle de dialogue pour lui demander des valeurs des attributs à l'utilisateur. Ces demandes peuvent êtres faites par décomposition des messages [TEN 87]. Au fur et à mesure que survient un changement sur un point de vue, le contrôle du tableau noir déclenche des sources de connaissances afin de mettre à jour le tableau noir.

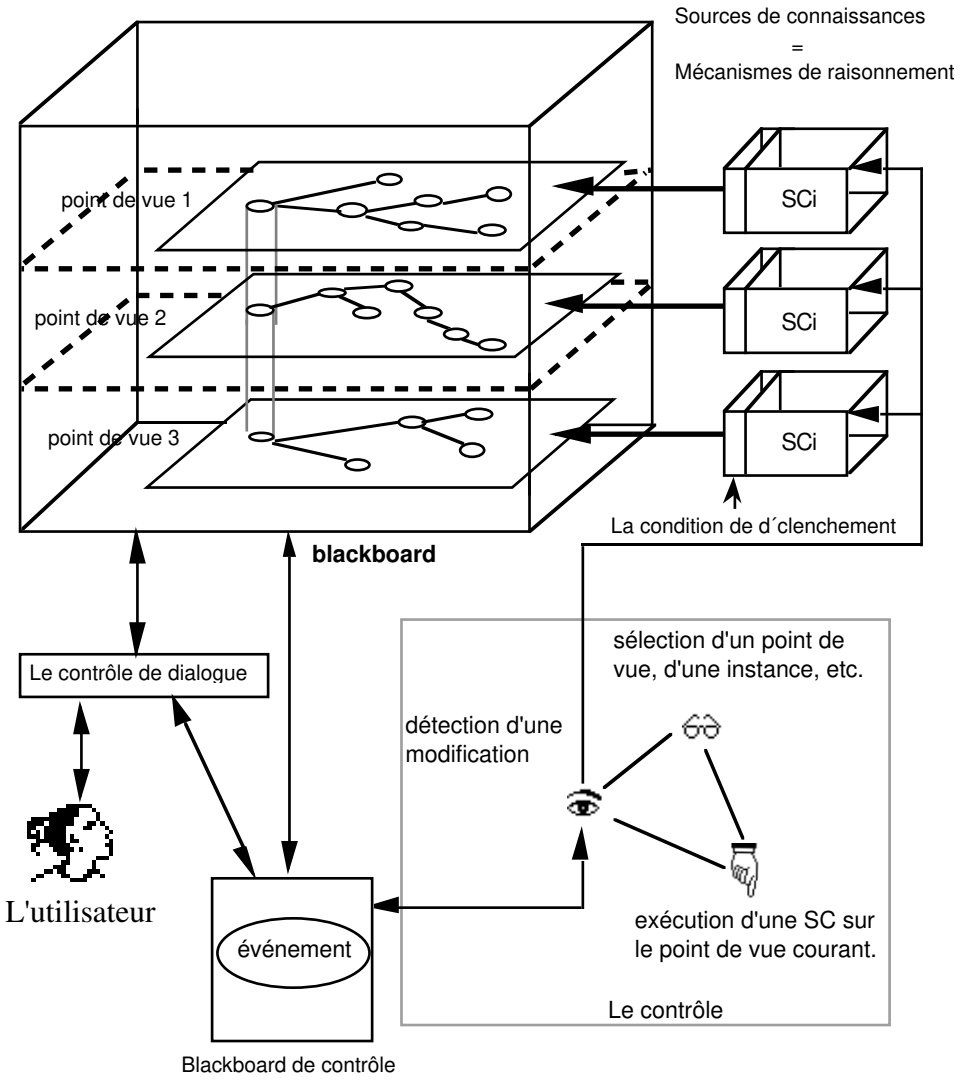


Figure 6.11 : Représentation du processus de classification comme un tableau noir.

A partir de cette représentation du processus de classification, nous pouvons modéliser le même problème en parallèle. Nous avons déjà montré l'analyse d'évaluation de contraintes en parallèle dans laquelle nous pouvons descendre par n'importe quel point de vue avec un résultat de la classification identique. Pour l'instant nous montrerons comment modéliser la classification en parallèle.
Le premier choix à faire concerne à la représentation de l'information du concept : les points de vue et les classes. Dans la section 4.2. nous avons défini une analogie entre les éléments du tableau noir et les éléments du modèle TROPES, dans laquelle le concept et les objets rattachés à ce dernier sont considérés comme l'espace solution pour les différentes opérations réalisées sur la base de connaissances. Il est possible de considérer trois formes de granularité pour la représentation des données :

- La plus grosse dans laquelle il n'existe qu'un panneau dans le tableau noir. Cet unique panneau est le concept perçu comme une structure de données globale. Il représente le médium général d'interaction entre les différents résolveurs au cours de la résolution de la classification d'un objet. Pour ce cas, il n'existe donc qu'un contrôle, qui planifie la résolution du problème. Il est chargé de déclencher les différentes sources de connaissances afin de descendre l'objet le plus bas possible dans les points de vue du concept. Lors de cette descente, tout changement d'une source de connaissances sur le tableau est détecté par le contrôle chargé de mettre à jour le tableau noir.

Cette solution entraîne le placement de tout le tableau noir, de même que le processus de contrôle sur un processeur P. Le désavantage de cette solution est qu'elle crée un goulot d'étranglement dû à la communication de tous les processeurs vers P. Cette communication correspond aux demandes des processus placés sur les autres processeurs différents de P. Les types de demande qui font le processus concernent :

- l'état d'une classe.
- le changement de l'état d'une classe possible en impossible ou sûre.
- la valeur pour un attribut.
- les sous-classes d'une classe pour l'appariement.
- les passerelles.

Le numéro de la demande est proportionnel au numéro de passerelles et de classes. Nous définirons la fonction **Com(f(n))** comme la fonction qui exprime le numéro de communications entre deux processeurs. Dans ce cas, le numéro total de communication est proportionnel à

Nombre de communications = $\mathbf{Com(p*n^h+pa+p*h*an*n)}$.

Finalement, il faut analyser le nombre de processeurs nécessaires pour le placement. Un processeur est nécessaire pour le contrôle et le tableau noir, et plusieurs processeurs pour les autres processus (les sources de connaissances qui sont déclenchées en parallèle). Le groupement des processus n'a aucune importance étant donné qu'ils ne communiquent qu'avec le contrôle pour modifier le tableau noir. Donc, le coût de communication entre les différentes sources de connaissances est nul.

- La plus fine dans laquelle chacune des classes qui appartient au concept est un panneau dans le tableau noir. Dans ce cas, le contrôle devient très difficile parce qu'il doit localiser une classe dans n'importe quel processeur. La mise à jour du concept devient pratiquement impossible et le coût du routage des messages entre les processus, par exemple pour la propagation de marques à travers d'une passerelle, est très élevé.

  L'implémentation effective de cette approche sur machine parallèle en affectant un processeur par nœud est assez élégante pour la propagation, c'est une implémentation massivement parallèle. Néanmoins, un certain nombre de problèmes se posent: le nombre de nœuds n'est pas fixe et peut dépasser les capacités de la machine, le nombre de liens n'est pas fixe ni limité et peut entraîner les mêmes problèmes, mais surtout, le contrôle des passerelles ne pourra pas être fait par l'algorithme.
- La dernière solution a été choisie pour nous comme l'optimale. La granularité de cette solution, au niveau de données, est un moyen terme entre les deux premières. L'idée de base est d'associer un point de vue à un panneau du tableau noir (figure 6.12). Pour chaque panneau il existe un contrôle, qui est chargé de mettre à jour le point de vue, et déclencher les sources de connaissances respectives lors d'une modification du panneau.

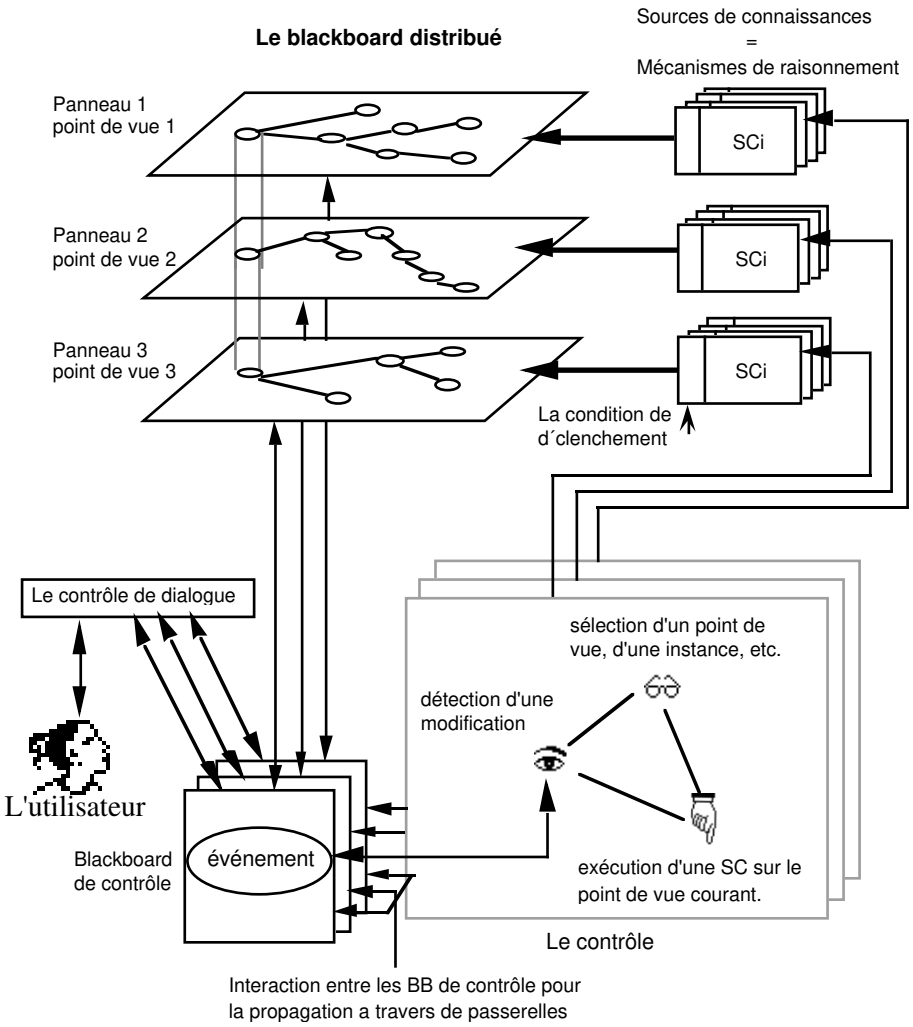


Figure 6.12 : Association d'un panneau à un point de vue.

Ces contrôles interagissent entre eux lors de la propagation de marques à travers une passerelle, et interagissent avec le contrôle de dialogue au fur et à mesure qu'ils ont besoin des valeurs pour les attributs. Le contrôle est divisé en deux niveaux :

- Le contrôle de bas niveau est d'une certaine façon indépendant de l'opération. Il doit faire la mise à jour du tableau noir avec les modifications faites pour les sources de connaissances. Par exemple, pendant la classification, alors que l'algorithme de classification descend l'objet dans la hiérarchie de classes, il peut marquer les différentes classes comme impossibles, possibles ou sûres. Donc, dans ce cas, il faut mettre à jour le tableau noir global : la base de connaissances.

- Le contrôle de haut niveau est chargé de planifier la résolution du problème, pour notre cas, le problème était classifier l'objet le plus bas dans les différents points de vue. Dans la suite, nous décrirons plus en détail le rôle de ce contrôle. Lors d'une modification d'un point de vue ou d'une classe, le contrôle doit sélectionner un point de vue ou une instance d'une source de connaissances. Ceci correspond exactement à l'étape de sélection. Enfin, après avoir sélectionné le point de vue et l'instance d'une source pour réaliser un pas d'une opération (par exemple, la propagation de marques), il faut exécuter la source de connaissances qui correspond à l'opération. Ces sources de connaissances peuvent être repertoriées comme (figure 6.13):
  - l'obtention d'information : qui se déclenche lors du choix d'une nouvelle classe, et dont le but est de compléter l'information manquante sur la classe

sûre d'un point de vue PVi. Dans l'instant, cette classe est la plus basse à laquelle est rattaché l'objet dans PVi.

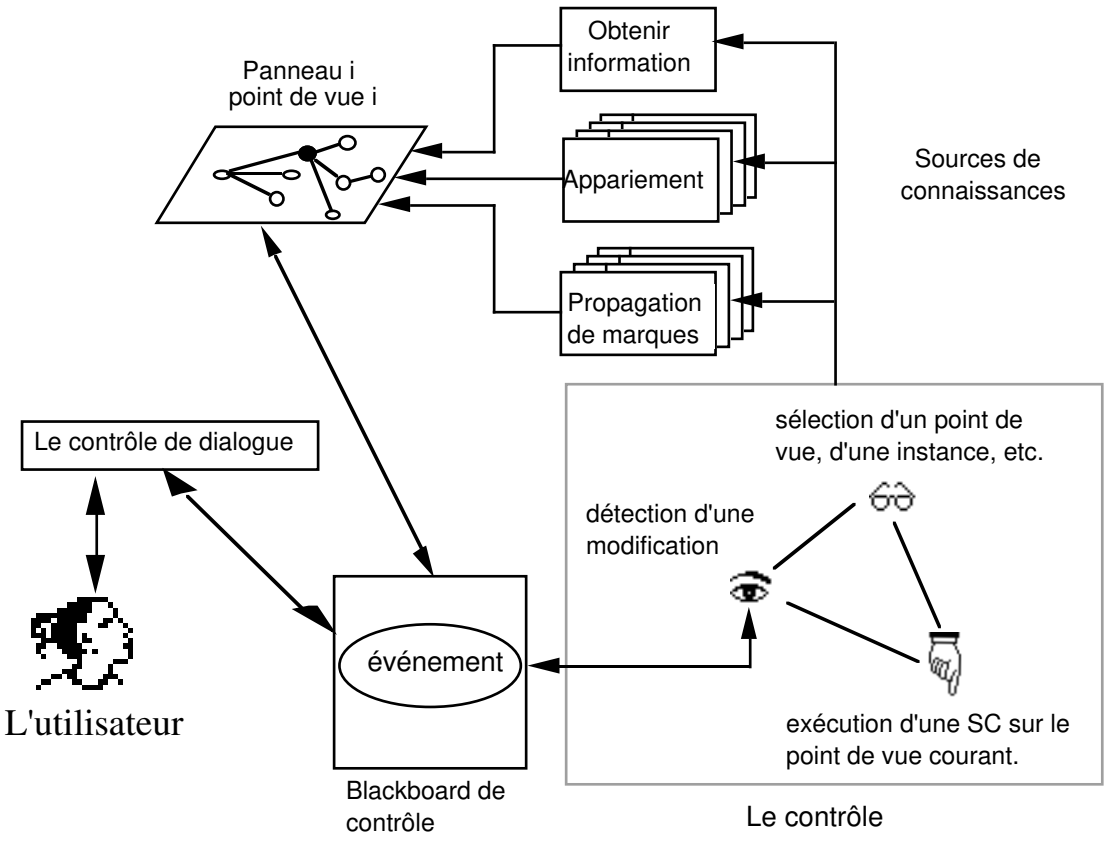


Figure 6.13 : Les sources de connaissances qui interagissent avec un panneau (point de vue).

- l'appariement : qui se déclenche pour choisir une nouvelle classe sûre parmi les sous-classes de la classe actuelle. Autrement dit, ces sources de connaissances essaient de trouver une sous-classe dont l'objet satisfait toutes les contraintes.
  Dans le cas de la source de connaissances d'appariement, le contrôle peut déclencher plusieurs instances de la source en parallèle, une pour chacune des sous-classes. Le but de chacune de ces sources est de valider les valeurs des attributs par rapport aux contraintes des sous-classes. Après cette validation, la source essaie de marquer la sous-classe comme impossible ou sûre. Au fur et à mesure que les sources de connaissances d'appariement marquent les classes, se produisent des événements qui sont détectés pour le contrôle.
  L'exécution en parallèle de plusieurs instances de cette source de connaissances n'entraîne une interaction entre eux qu'au niveau de la demande de valeurs pour les attributs manquants afin de faire le choix de la "meilleure" sous-classe possible. Cette interaction est faite au travers du contrôle de dialogue.

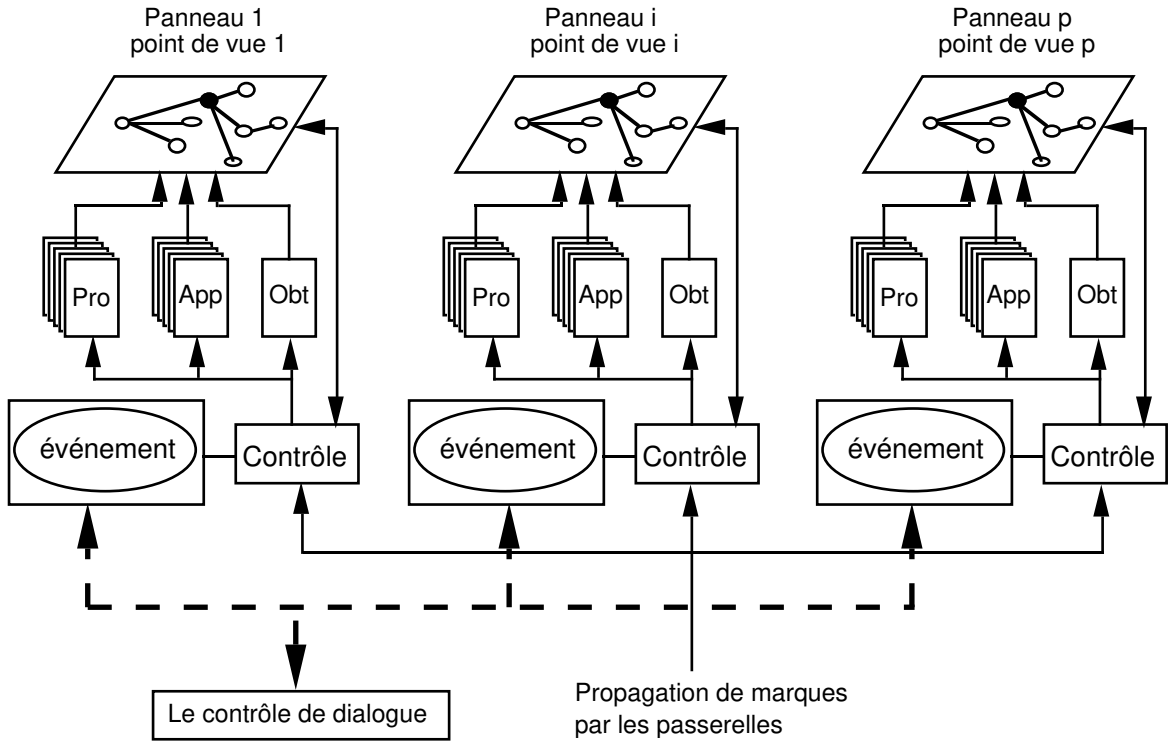


Figure 6.14 : Interaction entre les contrôles associés aux différents panneaux pour la propagation de marques à travers des passerelles.
Pro : C'est un processus de la procédure de propagation de marques. Chacune de ces instances marque un sous-arbre différent.
App : C'est un processus de la procédure d'appariement. Chacune de ces instances fait l'appariement de l'objet avec une des sous-classes de la classe actuelle.
Obt : C'est un processus de la procédure d'obtention d'information.

- la propagation de marques : lors des changements produits pour les instances de la procédure d'appariement sur les états des classes et des passerelles, plusieurs processus de la procédure de propagation de marques sont déclenchées par le contrôle.
  Le but de ces processus est donc la mise à jour des états des classes dans tous les points de vue. Autrement dit, la procédure de propagation de marques rétablit la cohérence en marquant "impossible" les sous-classes des classes impossibles, et en propageant des marques à travers les passerelles quand il le faut (figure 6.14). En 2.2.2.3, nous avons défini le premier type de propagation comme "propagation interne" et le second comme "propagation externe".
  Pour la propagation à travers des passerelles (définie entre C/PVi et D/PVj i≠j), lorsqu'une source change une des classes de la passerelle, il se produit un événement dans un des panneaux (le point de vue PVi ou PVj). Le contrôle associé à ce panneau détecte l'événement et communique au contrôle de l'autre panneau. Celui-ci se charge de déclencher un processus de la procédure de propagation de marques pour mettre à jour son point de vue.
  Ces sources coopèrent donc entre elles pour le partage d'information. Cette coopération a pour but de mener la solution incrémentale du problème à travers différents niveaux, afin de trouver une solution cohérente.

Les performances d'un programme exécuté sur une architecture parallèle dépendent principalement du placement des processus composant le programme sur les processeurs composant l'architecture. Le placement automatique permet entre autres une programmation transparente de l'architecture.
Avant l'exécution de la classification d'un objet dans un base de connaissances multi-points-vue, il est nécessaire de savoir placer sur les divers processeurs, les processus correspondant

aux sources de connaissances (ces sources sont instances de la procédure d'obtention information, de la procédure d'appariement ou de la procédure de propagation de marques), qui composent le programme de classification. Nous avons associé un point de vue à un panneau du tableau noir, ainsi qu'un contrôle pour chaqu'un de ces panneaux. Donc, les différentes instances de sources de connaissances déclenchées pour classifier l'objet dans la base sont associées aux différents panneaux. C'est-à-dire, pour chaque panneau, il existe des instances des sources chargées de classifier l'objet dans ce point de vue. Ces processus interagissent entre eux à travers le panneau du tableau noir. Pour cette interaction, il existe plusieurs alternatives :

- MIMD avec mémoire commune : Dans ce cas, les sources de connaissances peuvent modifier le point de vue sans avoir besoin d'échanger des messages entre les différentes sources ni avec le contrôle. Ces modifications peuvent être vues par toutes les autres sources, et bien sûr par le contrôle. Lors d'une modification, le contrôle se charge de déclencher les autres sources. Dans ce cas, normalement, les coûts de communication sont négligeables.

  Pour les MIMD avec mémoire commune (réseaux fortement couplés), la fonction d'allocation est moins complexe, étant donné que les problèmes de maintien de l'état courant du système et de la migration de processus ne se posent pas.

- MIMD sans mémoire commune : Dans ce cas, les processeurs sont connectés par des liens de communication, suivant certains types de topologie. Ces processus interagissent entre eux par échanges de messages. Il existe plusieurs critères pour évaluer un placement de processus sur une de ces architectures : la longueur des chemins de communication, l'équilibrage de la charge de travail des processeurs et des liens de communication [TAL 91]. Pour le problème du placement des instances des sources de connaissances dans ces architectures, il faut prendre en compte la totalité de ces critères.

  Si nous analysons le nombre de communications total du système pour le concept et pour chaque point de vue, on obtient :

  nombre total de communications pour le concept = $\mathbf{Com(p*n^h+pa+2*p*h*an*n)}$,

  nombre de communications pour chaque point de vue = $\mathbf{Com(n^h)}$,

  nombre de communications par demande de valeurs à l'utilisateur = **Com(h∗an∗n)**,

  nombre de communications par validation de valeurs = **Com(h∗an∗n)**,

  nombre de communications entre les contrôles des différents points de vue = **Com(pa)**,

  Nous pouvons conclure que la plupart des communications sont faites par les sources de connaissances dont l'espace solution est un même point de vue. C'est-à-dire, les communications faites par les sources de connaissances et le contrôle, qui interagissent entre eux pour classifier un objet dans un même point de vue, sont locaux. La communication entre les différents contrôles (il en existe un pour chaque point de vue) est minimale. Par exemple, si le nombre de passerelles est zéro, alors le nombre de communications est aussi zéro.

  Avant d'analyser le placement de processus pour ce cas, nous allons présenter quelques stratégies d'allocation avec heuristiques. Il existe deux approches générales: les algorithmes gloutons et les algorithmes itératifs.

  - Les algorithmes gloutons doivent être initialisés par une solution partielle qu'ils cherchent à étendre à chaque étape jusqu'à déterminer une solution réalisable.
  - Les algorithmes itératifs sont initialisés avec une solution complète qu'ils cherchent à améliorer à chaque itération. La complexité des algorithmes itératifs dépend du placement initial choisi, et elle est en général plus grande que celle des algorithmes gloutons, mais donne de meilleurs résultats.

  Il existe trois familles classiques d'heuristiques : les groupements de processus, la limitation de l'utilisation des routages et le recuit simulé.

Pour le placement des sources de connaissances, il est important de prendre en compte la fonction de communication Com. A partir des valeurs de cette fonction, nous pouvons déduire que :

- Pour l'allocation des processus de contrôle, le choix du processeur où est placé un contrôle par rapport aux autres processus n'est pas important, étant donné que le coût de communication est très bas = **Com(pa)**.
- L'allocation du contrôle de dialogue est très importante par rapport à l'allocation des processus de contrôle lors des requêtes des valeurs faites par les sources de connaissances (de la procédure d'obtention d'information et de la procédure d'appariement), étant donné le grand nombre de communications entre le contrôle d'un panneau et le contrôle de dialogue, qui est **Com(h∗an∗n)**.
- L'allocation des sources de connaissances qui interagissent avec le contrôle d'un panneau est importante, parce que le nombre de communications réalisées entre eux est très grand. D'un côté, pour chaque demande de valeur d'un attribut, la procédure d'obtention d'information et la procédure d'appariement demandent au contrôle du panneau les contraintes de l'attribut pour effectuer la validation. Cette validation entraîne un nombre de communications : **Com(h∗an∗n)**.
  D'un autre côté, la procédure de propagation de marques communique très souvent avec le contrôle du panneau pour changer l'état des classes du point de vue. Le nombre de communications est **Com(n^h)**.

D'après ces résultats, nous percevons qu'il est très important de placer tous les processus qui interagissent entre eux pour classifier un objet dans un point de vue très proches l'un de l'autre, afin que la longueur des chemins de communication entre eux soit très petite. Cette même conclusion est faite par rapport à l'allocation de contrôles et du contrôle de dialogue.

- MIMD sans mémoire commune de large granularité, qui communiquent à travers un réseau local : dans ce cas, nous pouvons faire la même analyse faite pour le cas précédent. Néanmoins, les coûts de communication sont ici négligeables, étant donné que pour les réseaux à diffusion, la distance entre processeurs n'intervient pas dans le choix du processeur pour l'exécution d'un processus. Le seul critère pris en compte est la charge des divers processeurs.

A présent, nous allons proposer le meilleur placement de processus pour l'algorithme parallèle de classification d'un objet dans une base de connaissances multi-points-de-vue. A partir de l'analyse précédente, nous pouvons faire un placement de processus qui prend en compte le nombre de communications de façon à minimiser la fonction **Com**. Il y a des communications qui ne peuvent pas être réduites, comme par exemple les messages des demandes des valeurs des attributs. Mais, tous les contrôles demandant au contrôle de dialogue CD des valeurs, le placement du contrôle de dialogue doit être indépendant des processus de contrôle PC, et dans le meilleur de cas, la longueur des chemins de communication entre le CD et les PC doit être la même.

Si nous groupons les processus qui interagissent entre eux pour classifier un objet dans un point de vue en un même processeur (figure 6.15), nous pouvons minimiser le coût de communication pour mettre à jour les différents panneaux. La raison en est simple : le coût de communication de deux processus qui ont d'une mémoire commune est zéro. Le nombre de groupes construits est égal au nombre de processeurs.

Pour les processus placés dans le même processeur, nous avons utilisé les tâches et les processus légers sur MACH [GOS 91] (§ Annexe 1). Une tâche est l'environnement dans lequel les processus légers peuvent s'exécuter; elle fournit un espace d'adressage virtuel et l'accès protégé aux ressources du système. Une tâche comporte des objets de communication et comptabilise les ressources qu'elle utilise. Un processus léger représente l'exécution d'un flot d'instructions. Chaque processus léger s'exécute dans le contexte d'une seule tâche jusqu'à terminaison. Un tâche peut contenir plusieurs processus légers. Les processus légers d'une même tâche peuvent s'exécuter en parallélisme vrai sur un site multi-processeurs. L'ordonnancement des processus légers s'effectue par tranches avec différentes variantes [BLA 90]. L'ensemble de tâches peut être assimilé à une PTâche du système PARX [GON 91].

Dans notre problème une tâche représente la classification de l'objet dans un point de vue. Sa structure de données est donc le point de vue. Les processus légers qui composent la tâche

représentent la procédure d'obtention d'information, la procédure d'appariement, et la procédure de propagation de marques. L'espace d'adressage virtuel pour tous ces processus légers est le point de vue.
Pour la communication entre tâches, nous utilisons des ports et les primitives de communication "send" et "receive". Ces ports permettent la désignation de tâches indépendamment de leur localisation physique. Ils permettent aux processus légers d'échanger des messages quelle que soit leur localisation et sans qu'ils aient à se préoccuper de celle de leur correspondant.

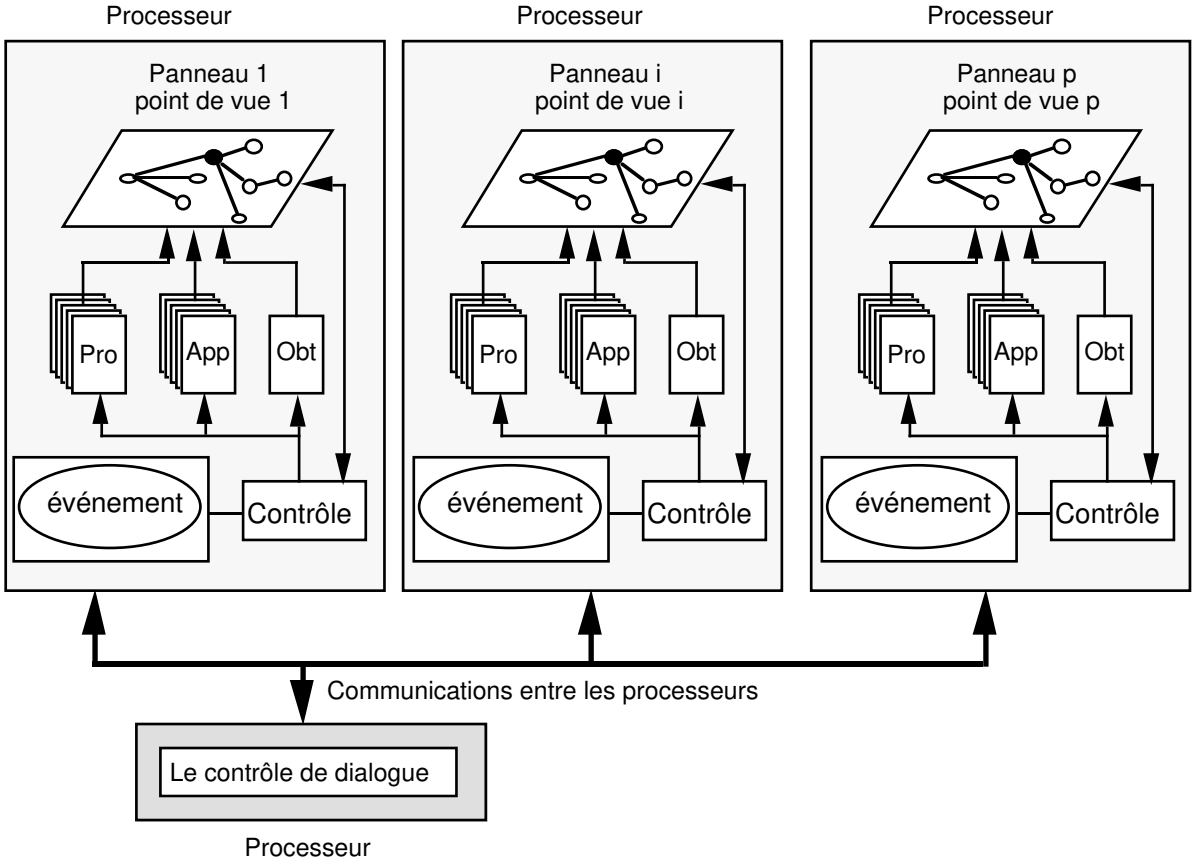


Figure 6.15 : Allocation finale de processus (instances d'une source de connaissance) par processeur, et communication entre eux, pour la mise en œuvre de l'algorithme parallèle de classification.

L'allocation des groupes de processus sur les processeurs se fait de telle sorte que deux tâches communicantes sont placées sur deux processeurs voisins ou sur un même processeur. Ainsi la fonction de routage de messages entre les processus est réduite. Une tâche ne communique avec une autre que s'il existe au moins une passerelle entre les points de vue des tâches (le point de vue est la structure de données d'une tâche).
Dans le mise en œuvre tous les processus qui interagissent entre eux, à travers un même panneau (les processus qui ont le même point de vue) sont réunis dans un même groupe. Chaque groupe de processus G est placé dans un même processeur afin de réduire le coût de communication, car toutes les communications entre les processus qui appartiennent à G sont locaux. Le coût d'une communication locale est considérée égale à zéro (formule 1, § 6.2.5). Donc, le coût de communication des processus, qui appartiennent au même groupe G est zéro.
D'autre part, nous avons utilisé une heuristique (§ 6.2.6) pour le placement des groupes de processus (figure 6.15), qui prend en compte les communications entre les différents contrôles. Deux groupes de processus ne communiquent entre eux que s'il existe une passerelle entre les deux points de vue associés aux groupes de processus.

Finalement, pour le mise en œuvre du système de classification en parallèle en utilisant le modèle de tableau noir nous avons utilisé le système DSBB. Nous allons exprimer d'abord les règles, puis les structures de données du tableau.

- Dans DSBB, chaque source de connaissances doit connaître les conditions sous lesquelles elle peut contribuer à la solution, et les préconditions nécessaires pour son déclenchement. Donc, une source de connaissances est composée de deux parties: la précondition pour son déclenchement et le corps qui constitue la partie action. Ces connaissances sont représentées de façon déclarative comme un ensemble de règles. Par exemple, la règle qui exprime la propagation interne de marques est écrite comme :

```
if ((l'état de la classe C a changé) et (nouveau état de C == SURE)) then
   propagationSure(conceptActuel,PVActuel,classeActuel,C)
elsf ((l'état de la classe C a changé) et (nouveau état de C == IMPOSSIBLE)) then
   propagationImpossible(conceptActuel,PVActuel,classeActuel,C)
endif
```

La règle qui exprime la propagation externe de marques est écrite comme :

```
if ((l'état de la classe C a changé) et (nouveau état de C == SURE)) then
   propagationSurePasserelle(conceptActuel,PVActuel,classeActuel,C)
elsf ((l'état de la classe C a changé) et (nouveau état de C == IMPOSSIBLE)) then
   propagationImpossiblePasserelle(conceptActuel,PVActuel,classeActuel,C)
endif
```

Nous rappelons que lors de la traduction ("précompilation") le système attache beaucoup d'informations aux structures de données du tableau. Parmi ces informations on trouve les règles qui doivent être déclenchées lors d'une modification de la structure. Par exemple, la règle de propagation de marques interne est associée à toutes les classes d'un point de vue, tandis que la règle de propagation de marques externe n'est associée qu'aux classes qui sont aussi des passerelles.

Une autre information associée aux structures de données indique si elles sont partagées par plusieurs règles en lecture ou en écriture. Cet espace partagé est situé dans la mémoire principale. Pour éviter les problèmes d'incohérence, le système associe aux structures de données l'information de "section critique". C'est-à-dire qu'il est associé à chacune de ces structures des fonctions pour la gestion d'exclusion mutuelle : mutex_lock et mutex_unlock.

Cette régle est traduite pour un compilateur en un programme C de la façon suivante (nous pouvons faire la comparaison avec les fonctions de propagation dans l'annexe 5) :

```
Règle(conceptActuel,PVActuel,classeActuel,C) {
/* Variables locales */
/* Appel à une fonction du système pour associer les paramètres de la règle aux structures de
données du tableau */
        …
        /* le système vérifie s'il est possible d'arriver à un interblocage */
        VérificationCycle(...);
        /* Section critique : variable partagée */
        mutex_lock(verrou);
        if ( ModifiéEtat(C) && (état(C) == SURE ) {
                propagationSure(conceptActuel,PVActuel,classeActuel,C);
        } else {
                if ( ModifiéEtat(C) && (état(C) == IMPOSSIBLE ) {
                        propagationImpossible(conceptActuel,PVActuel,classeActuel,C);
                }
        }
        mutex_unlock(verrou);
}
```

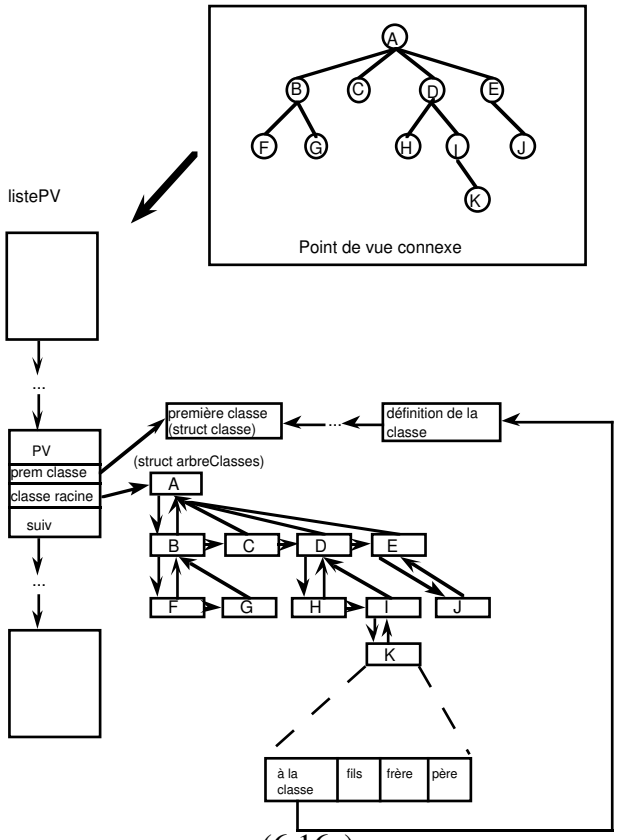
- Nous allons maintenant montrer un exemple de structures de données créées par le système lors de la compilation.

```
base
{
 exemple
 concepts      [qui] ;
  points_de_vue [premier, ... , ... ]
}
concept
{
 qui
   clef              [... ] ;
   points_de_vue     [premier, ... , ... ]
}
classe
{
        qui.A/premier
                …
}
classe
{
        qui.B/premier
        sorte_de      # [A] ;
                …
}
classe
{
        qui.C/premier
        sorte_de      # [A] ;
                …
}
classe
{
        qui.D/premier
        sorte_de      # [A] ;
                …
}
classe
{
        qui.E/premier
        sorte_de      # [A] ;
                …
}

…

pass
{
        …
}
```

Exemple 6.1 : Exemple d'une base de connaissances.

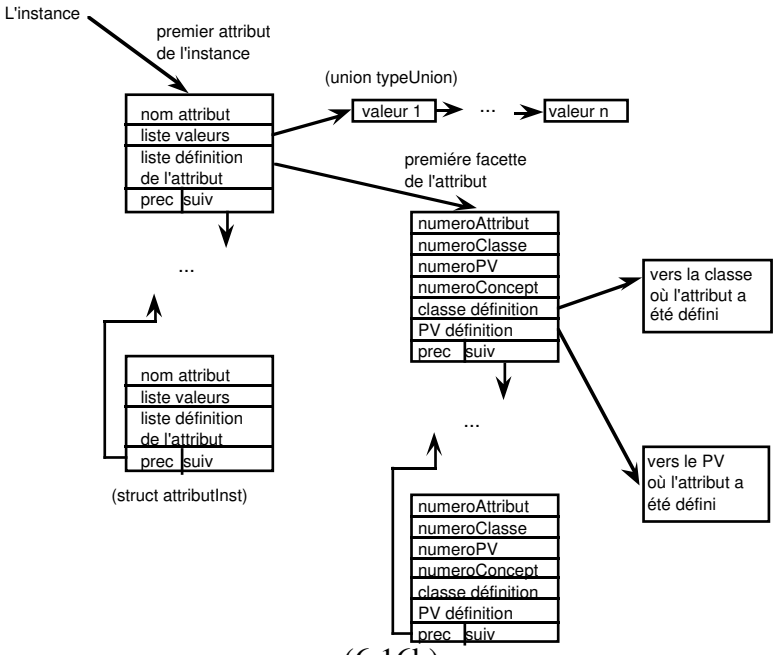
Les structures de données créées par le système sont (figure 6.16a, 6.16b, et 6.16c) :

Arbre de classes pour chaque point de vue PV



(6.16a)

Structure de données pour l'Instance



(6.16b)

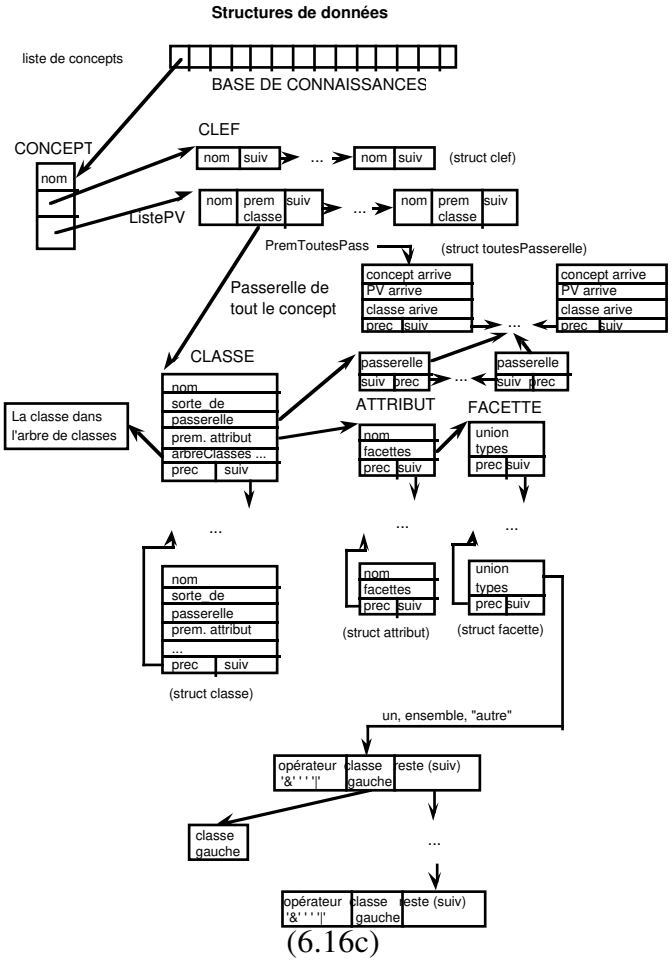


Figure 6.16 : Structures de données (de la base de connaissances de l'exemple 6.1 créées par le système pendant la précompilation.

6.4 Conclusions

Dans ce chapitre a été présentée la mise en œuvre du système de classification en parallèle pour TROPES que nous avons développée dans les chapitres 2 et 4. L'analogie faite entre le modèle de connaissances TROPES et le modèle de tableau noir a permis d'implémenter de façon naturelle l'algorithme parallèle.

Finalement, cette mise en œuvre a été faite sur un système réparti local basé sur un réseau à diffusion. Cette réalisation entraîne certains avantages comme la diminution des coûts de communication. Ces coûts peuvent être ignorés sur un réseau à diffusion. Un autre avantage de cette réalisation est l'utilisation d'un algorithme simple pour l'allocation de processus sur les processeurs, qui prend en compte les communications, mais pour lequel le critère le plus important est le partage de l'information entre les différentes sources de connaissances.

CONCLUSIONS

L'objectif principal de notre travail était d'étudier les possibilités de parallélisation dans le modèle de connaissances multi-points de vue TROPES. L'intérêt principal de l'étude est basé sur des systèmes répartis et des machines parallèles à mémoire distribuée. D'autre part, la classification des instances étant l'opération la plus utilisée, c'est naturellement sur celle-ci que nous avons décidé d'étudier les possibilités de parallélisation.

Bonnes performances de la parallélisation

La faisabilité d'un tel projet a pu être démontrée puisque notre prototype est aujourd'hui opérationnel sur des systèmes répartis et qu'il permet d'obtenir des performances supérieures à celles obtenues avec des machines classiques. De nombreuses optimisations sont encore possibles, en particulier la mise en œuvre du prototype sur une machine massivement parallèle du type Méganode [ARR 92].

Il est important de rappeler que ces performances ont été obtenues bien que le modèle TROPES ne possède pas de notion de parallélisme. Aucune indication relative à l'architecture cible (nombre de processeurs, topologie, ...) n'est fournie par l'utilisateur qui définit la base de connaissance ou qui l'utilise. La base de connaissances compilée pour être exécutée sur un seul processeur (en séquentiel) ou sur plusieurs processeurs (en parallèle) n'a pas besoin d'être modifiée. Ceci est possible car le parallélisme n'a pas été introduit au niveau du langage de TROPES mais au niveau de la machine abstraite cible du compilateur. Cela permet de suivre l'évolution des techniques de compilation en séquentiel sans remettre en cause les principes fondamentaux du modèle TROPES. Ce dernier atteint les objectifs de portabilité et d'extensibilité que nous nous étions fixés, en utilisant des modèles parallèles pour la résolution de problèmes en intelligence artificielle comme le tableau noir et le modèle acteur.

Un point d'ombre cependant pour les machines massivement parallèles : à problème constant, le modèle parallèle pour TROPES permet d'obtenir un gain de performances temporelles en fonction du nombre de processeurs, mais le coût en taille mémoire augmente également en fonction du nombre de processeurs. Ceci est dû à la technique de duplication physique de données logiquement partagées (les codes des sources de connaissances et du contrôle sont également dupliqués sur chaque nœud). Il est clair que ceci est un inconvénient du modèle : la parallélisation ne permet pas de résoudre un problème nécessitant plus de mémoire qu'il n'y a en de disponible sur un nœud alors que, en pratique, l'utilisation de machines parallèles correspond souvent à un volume de données à traiter plus important que ce qu'il est possible de traiter efficacement avec un machine séquentielle. Cependant, ce problème est résolu dans les systèmes répartis, qui normalement utilisent la mémoire virtuelle (comme MACH, pour lequel la mémoire virtuelle est un élément essentiel des abstractions du système).

Il existe aussi d'autres difficultés pour la réalisation d'un modèle parallèle pour la classification dans TROPES : la modélisation du problème, le placement de tâches, la communication entre les différentes tâches, le placement de données, le partage d'information, le routage de messages, etc. Cependant, une des difficultés majeures de cette approche est le risque pour le concepteur d'un tel système, de se trouver rapidement "perdu", face à la manipulation des paramètres très divers caractérisant cette technologie. En effet, cette approche offre des possibilités d'implantations et de choix techniques variés, ce qui exige un investissement de conception important, et conduit au risque pour le concepteur de se trouver confronté à un logiciel ingérable et lourd. Pour éviter ce risque, le premier prototype qui a été réalisé pour la classification d'instances dans TROPES a été abandonné.

Pour éviter ces situations d'impasse, il s'avère nécessaire de disposer de modèles formels provenant d'une étude approfondie des connaissances et des raisonnements, qui aident à cerner les besoins réels de l'application. C'est ainsi que deux modèles formels ont été proposés dans le chapitre 4 (le modèle de "tableau noir" et le modèle "acteur") s'avérant en effet très utiles pour la structuration des connaissances et des raisonnements. Les intérêts principaux dans l'utilisation de ces modèles sont les suivants :

- facilité d'identification et de transcription informatique des divers classes de connaissances et des raisonnements impliqués dans l'expertise à modéliser.
- appréhension des aspects intrinsèques au raisonnement à travers les diverses formes de connaissances manipulées.

- analyse des possibilités d'intégration des raisonnements non seulement à l'intérieur des tâches, mais aussi dans le cadre d'une coopération entre tâches
- possibilité d'une conception constructiviste de systèmes intégrant d'abord des raisonnements simples, puis des combinaisons complexes de raisonnements.
- spécification de comportements de haut niveau permettant la modélisation de démarches générales de résolution.

Le modèle de "tableau noir" et le modèle "acteur" peuvent fournir un moyen d'exploiter implicitement le parallélisme (vis-à-vis du programmeur), mais les problèmes de gestion du parallélisme ne disparaissent pas pour autant. Ils sont repoussés au niveau du concepteur des systèmes parallèles. Par exemple, pour la mise en œuvre du modèle "acteur" [QUI 92c] et du modèle de "tableau noir" [QUI 92d], il faut résoudre, entre autres, les problèmes de communication, de synchronisation et de routage. En particulier, la gestion des différentes ressources matérielles implique l'utilisation d'un langage à parallélisme explicite.
Néanmoins, il existe certains désavantages du modèle acteur par rapport au modèle de tableau noir pour la modélisation de la classification en parallèle des instances dans TROPES, notament l'impossibilité d'obtenir un état global du problème en un instant donné. En effet, le modèle acteur est un modèle asynchrone, tandis que le modèle tableau noir peut être vu comme un modèle synchrone (la synchronisation peut être faite à travers la mémoire partagée : le tableau).
L'expérience acquise lors du développement des nombreux outils mis en œuvre pour réaliser notre prototype (parmi ceux-ci, le modèle de tableau noir DSBB [QUI 92d] et le modèle acteur LDBA [QUI 92c]) sur un système réparti fait ressortir le manque de logiciel de base et de système pour ce genre de machine : peu de chose existet à l'heure actuelle et l'avenir de telles machines nécessitera un effort de recherche considérable (comme le projet HARNESS [HAR 91]). Les approches tableau noir et acteur s'avèrent des outils technologiques de haut niveau bien adaptés à la résolution des applications complexes, car elles permettent une représentation de la connaissance, la manipulation de schémas de raisonnement multi-formes, ainsi qu'une résolution coopérative des problèmes.
Les machines parallèles à mémoire distribuée et les systèmes répartis font apparaître beaucoup plus de problèmes liés à l'exploitation du parallélisme que les machines parallèles à mémoire partagée. Pour ces dernières, le matériel fournit "implicitement" des solutions à la plupart des problèmes de synchronisation (l'accès à la mémoire partagée est l'outil élémentaire de synchronisation). Bien que de nombreux résultats aient été obtenus en algorithmique parallèle (et pour l'intelligence artificielle en particulier, le nombre de résultats est très réduit), peu de solutions efficaces ont été mises en œuvre sur les machines commerciales. Par exemple, bien que de nombreux langages permettent de définir des processus légers/coroutines et des outils de synchronisation/communication, aucun compilateur actuel de ces langages (C//, ADA, OCCAM, ...) ne permet de s'abstraire de la frontière qu'il y a entre les nœuds d'un réseau de processeurs ou un réseau de machines sans mémoire commune. Actuellement, ces générateurs de code ne produisent que du code pseudo-parallèle (multi-programmation) pour un seul processeur et l'on doit recourir à d'autres outils, comme par exemple des langages de configuration, pour décrire le placement du graphe de processus sur le graphe de processeurs. Chaque utilisateur ou programmeur se pose le problème complexe de placement/routage pour chacun de ses programmes et pour chaque configuration de la machine (machines parallèles reconfigurables) ou des machines (systèmes répartis) utilisées : avec les outils actuels il ne peut pas faire abstraction des caractéristiques de la machine ou machines cibles, sauf pour quelques cas très particuliers (comme le cas d'un système réparti basé sur un réseau local à diffusion. Dans ce cas, il n'existe pas de routage et le placement n'est pas très important, car la distance entre les différentes tâches n'a aucune importance). Cela rend complexe la programmation des machines à mémoire distribuée.

Perspectives

Les résultats pratiques du projet de parallélisation de la classification d'instances dans TROPES sont :

- la mise en œuvre du modèle sur un système de tableau noir qui fournit une bonne base de travail pour affiner l'étude de la parallélisation implicite du modèle TROPES.
- de nombreux outils, sous-produits intermédiaires du projet, qui ont permis de mettre en évidence les problèmes que pose l'exploitation de parallélisme réel (parallélisme

physique) et qui apportent quelques éléments de solution. Parmi les outils : un langage acteur réparti (LDBA) et système de tableau noir concurrent (DSBB).

La complexité a été présentée pour le modèle séquentiel de classification en utilisant le modèle RAM [AHO 74] et pour le modèle parallèle en utilisant le modèle CRCW COMMUNE PRAM [KAR 90]. Après l'analyse de la complexité de deux modèles, nous pouvons conclure que l'algorithme parallèle réduit vraiment la complexité du processus de classification par rapport à celle du processus séquentiel. Dans le dernier cas, la complexité est fonction du nombre de points de vue du concept, du nombre de sous-classes d'une classe, du nombre d'attributs définis dans une classe, du nombre de contraintes définies pour un attribut et de la profondeur de l'arbre de classes. Pour l'algorithme parallèle, cette complexité est une fonction qui ne dépend que de la hauteur du point de vue (table 1).

Cependant, c'est le modèle EREW PRAM qui a été utilisé pour la mise en œuvre de l'algorithme parallèle, avec un coût de log(nombre de processeurs) pour cette simulation. La comparaison des complexités des algorithmes séquentiel et parallèle fait apparaître les avantages de ce dernier.

h	= hauteur maximale de l'arbre de classes (point de vue)
n	= nombre maximal de sous-classes de la classe actuelle
an	= nombre maximum d'attributs définis dans une classe
ct	= nombre maximal de contraintes d'un attribut.
f0	= complexité maximale de la validation des valeurs des attributs par rapport aux contraintes
pa	= nombre de passerelles actives dans le point de vue actuel
p	= nombre de points de vue du concept
O(classification séquentielle)	**$= O(p*(h*n*an)+(n+pa*h^2))$.**
$O_{//}$(classification en parallèle)	**$= O_{//}(h)$**

Table 1 : Complexité du modèle séquentiel et parallèle pour la classification d'une instance dans TROPES.

Le succès du projet permet d'envisager différents types de continuation :

- La mise en œuvre en parallèle de tout le modèle TROPES : la classification de classes, l'évaluation de tâches, le filtrage, le raisonnement hypothétique, etc. Enfin, l'introduction du parallélisme dans TROPES, grâce aux recherches menées actuellement dans le laboratoire IRIMAG-LIFIA (Grenoble) sur les approches réparties, permettra d'enrichir le modèle proposé dans ce travail en introduisant de nouveaux paramètres pour l'étude du raisonnement en mode d'exécution parallèle.
- La mise en œuvre du modèle parallèle sur une machine massivement parallèle du type Méganode.

ANNEXE 1 : LE SYSTEME D'EXPLOITATION MACH

Mach[14] est supporté par le Département de la Défense aux Etats-Unis, comme une partie du projet DARPA, et a été choisi comme système d'exploitation par l'Open Software Foundation.
Mach est un système intégré dont le noyau a été conçu comme une sorte de noyau Unix auquel a été rajouté le support nécessaire à l'exécution de plusieurs processus légers ("threads") au sein d'un même processus Unix, ainsi qu'une gestion de mémoire virtuelle innovatrice, élément essentiel des abstractions du système (figure 1). Ses objectifs sont de supporter aussi bien les machines multi-processeurs que les systèmes répartis et la compatibilité avec l'interface de 4.3BSD.

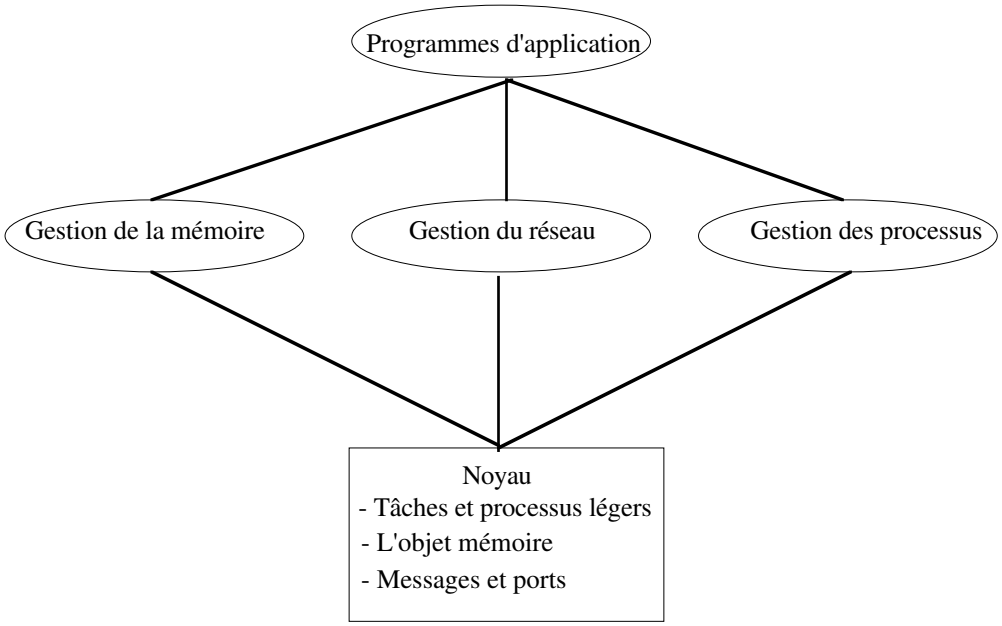


Figure 1 : L'environnement de Mach

Le modèle d'exécution est basé sur deux abstractions qui subdivisent un processus Unix en deux niveaux : la tâche et le processus léger. Une tâche est l'environnement dans lequel les processus légers peuvent s'exécuter ; elle fournit un espace d'adressage virtuel et l'accès protégé aux ressources du système. Une tâche comporte des objets de communication et comptabilise les ressources qu'elle utilise. Un processus léger représente l'exécution d'un flot d'instructions; chacun s'exécute dans le contexte d'une seule tâche jusqu'à terminaison. Une tâche peut contenir plusieurs processus légers et ceux-ci peuvent s'exécuter en parallélisme vrai sur un site multi-processeurs. L'ordonnancement des processus légers s'effectue par tranches avec différentes variantes. Mach offre un modèle d'exécution plat, la relation traditionnelle d'Unix étant remplacée par des droits d'accès sur une tâche.
La relation entre tâches, processus légers et processeurs pour des systèmes multi-processeurs et pour les systèmes répartis peut être visualisée sur la figure 2 :

[14] La présentation du Mach a été repris de [RAS 86], [GOS 91] et [TEV 87].

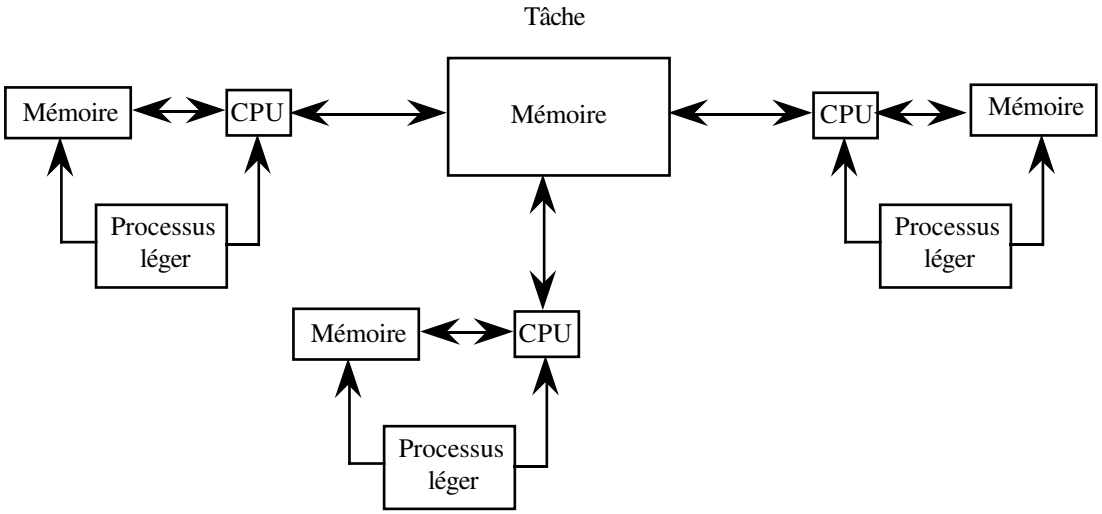


Figure 2a : La relation entre les tâches, les processus légers et les processeurs dans un système multi-processeur.

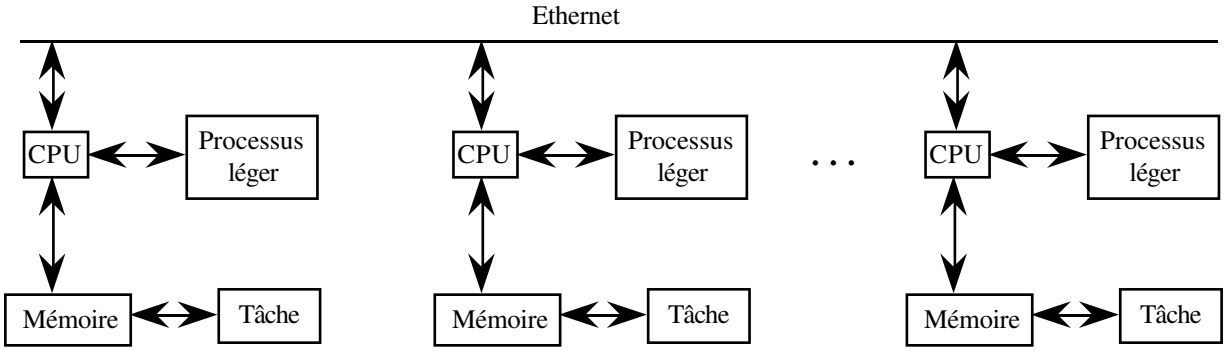


Figure 2b : La relation entre les tâches, les processus légers et les processeurs dans un système réparti.

Une troisième abstraction introduite dans Mach est l'objet mémoire, qui permet la gestion de la mémoire secondaire et notamment la gestion de la mémoire virtuelle (figure 3). L'espace d'adressage est géré par pages. Une tâche peut allouer des régions; une région est un intervalle d'adresses consécutives. Des gestionnaires de pages sont associés aux pages physiques. Ils reçoivent un message lors de chaque faute de page concernant l'une des pages dont ils ont la charge, ou lorsqu'il faut les écrire en mémoire secondaire. Chaque page peut être protégée en lecture, écriture et exécution.

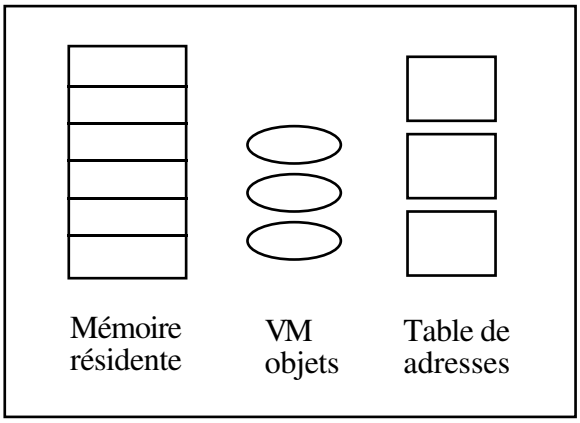


Figure 3 : Dans Mach, l'implémentation de la mémoire virtuelle est indépendante de la machine.

Les objets de communication sont appelés “ports”, mais ils sont définis comme étant de simples canaux de communication unidirectionnels implantés comme des files d’attente de messages gérées et protégées par le noyau.
Les primitives de communication correspondent aux opérations d’émission (“send”) sur un port, auquel cas le message est rajouté à la file d’attente, ainsi qu’aux opérations de réception (“receive”) sur un port, auquel cas le message en tête de la file d’attente est délivré. Si aucun message n’est disponible, la primitive “receive” est bloquante. Les opérations d’émission et de réception peuvent être combinées dans un seul appel système.
Plusieurs processus légers peuvent avoir le droit d’émettre sur un port, mais un seul a le droit de recevoir. Lors de l’émission d’un message, un processus léger peut, soit décider d’attendre que le message soit déposé dans la file d’attente du port, soit continuer son exécution et être prévenu a posteriori par un message généré par le noyau. Le premier cas correspond à une communication synchrone et le second à une communication asynchrone.
Les ports de Mach permettent la désignation de tâches indépendamment de leur localisation physique. Ils permettent aux processus légers d’échanger des messages quelle que soit leur localisation physique et sans qu’ils aient à se préoccuper de celle de leur correspondant[15], même si leur correspondant se trouve physiquement dans une autre machine (figure 4).

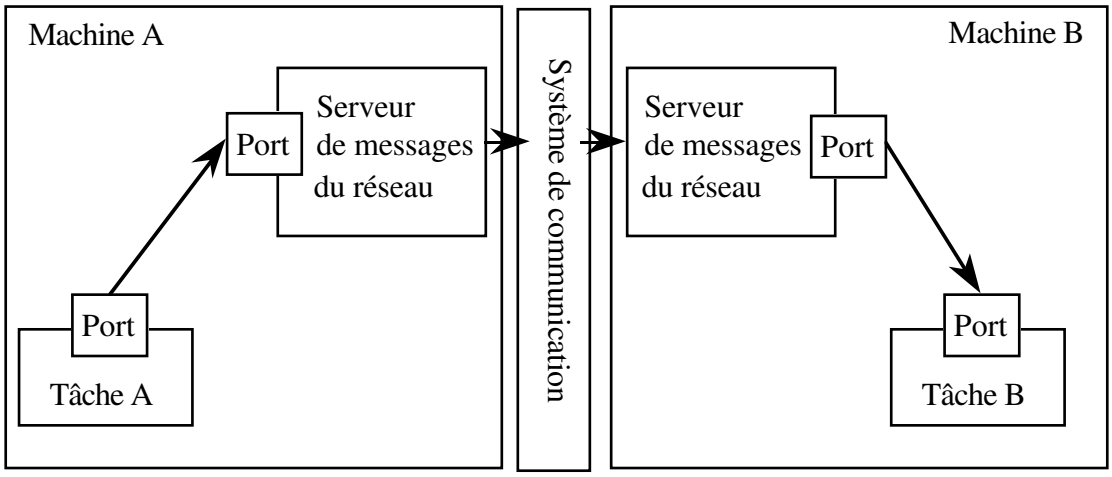


Figure 4 : Communication de deux tâches placées dans des machines différentes à travers un port.
Des droits sont requis pour opérer sur des ports ; ils sont collectifs pour tous les processus légers d’une même tâche. Une seule tâche possède le droit de réception sur un port, mais plusieurs de ses processus légers peuvent demander à recevoir simultanément. Les droits d’accès aux ports peuvent être transmis dans un message. Un port peut être restreint ou non. Dans le premier cas, il doit être explicitement désigné par son nom ; dans le second, il fait partie de l’ensemble de ports par défaut sur lesquels sont attendus des messages.

[15] Il existe une certaine similitude entre la façon d'envoyer les messages dans le modèle Acteur et la communication à travers de ports de Mach [COB 92].

ANNEXE 2 : PARCOURS EN PARALLELE D'UN GRAPHE

Il existe plusieurs méthodes pour faire un parcours d'un graphe orienté, et ces méthodes dépendent de la structure, et de l'opération qui est faite : n-profondeur, largeur-profondeur, largeur d'abord. Ces parcours sont faits en supposant qu'il y a n processeurs.

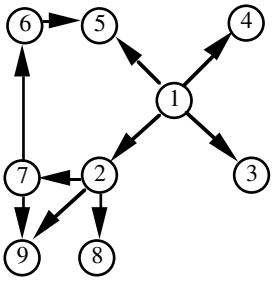


Figure 1 : Graphe à parcourir

a) N - profondeur (figure 2): dans un parcours n - profondeur, n sommets peuvent être visités à la fois. Puis un des sommets qui viennent d'être visités est choisi comme le prochain point de départ pour continuer le parcours.

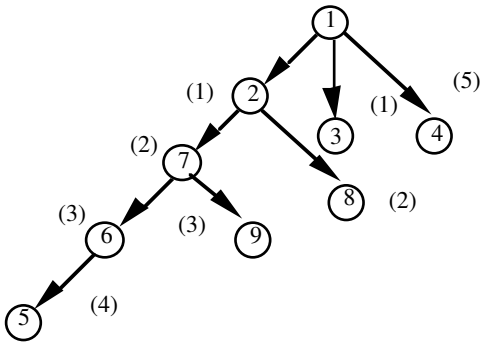


Figure 2 : Parcours du graphe (figure 1) en n-profondeur. Le numéro entre parenthèses est l'ordre du parcours

b) Largueur-profondeur (figure 3) : le parcours largueur-profondeur visite tous les sommets qui sont adjacents au sommet courant; puis le sommet le plus à gauche est choisi comme le prochain point de départ pour continuer le parcours; après, il se fait de même avec ce sommet.

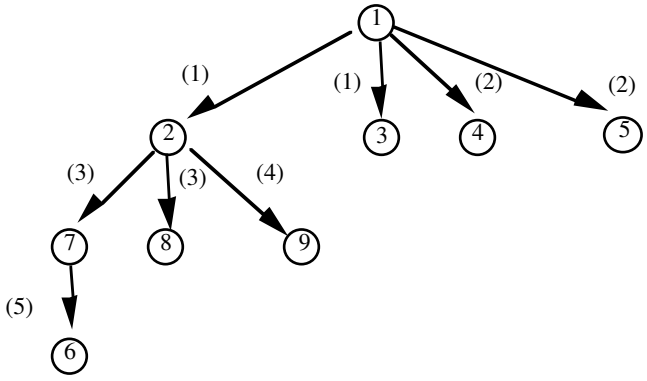


Figure 3 : Parcours du graphe (figure 1) en largeur-profondeur.

c) Largueur d'abord (figure 4) : le parcours largueur d'abord visite tous les sommets adjacents au sommet courant. Puis tous les sommets du niveau suivant sont visités, etc.

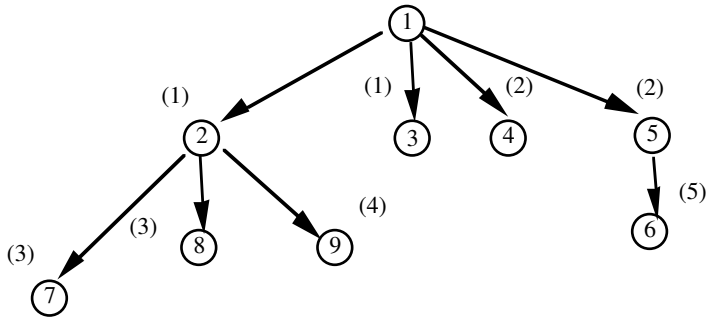


Figure 4 : Parcours du graphe (figure 1) en largeur d'abord.

ANNEXE 3 : CLASSIFICATION DES ARCHITECTURES PARALLELES

La grand diversité d'architectures parallèles a donné lieu à de nombreuses propositions de classification; parmi elles, la méthode devenue la plus populaire et la plus utilisée dans la littérature, à tel point que l'on pourrait difficilement parler d'un ordinateur parallèle et de son architecture sans y faire mention, est celle proposée par Flynn (table 1) [FLY 72]. Il existe aussi d'autres types de classification comme la taxonomie de Kuck (table 2) et la taxonomie de Treleaven (table 3) [ALM 89].

		Number of data Streams	
		Single	Multiple
Number of Instruction Streams	Single	SISD (von Neumann)	SIMD (vector, array)
	Multiple	MISD (pipeline?)	MIMD

Table 1 : Taxonomie de Flynn. Le tableau a été repris sans modification de [ALM 89]. Le modèle séquentiel de Von Neumann correspond à la classe SISD. Parmi les quatre classes, seulement SIMD et MIMD s'appliquent aux architectures parallèles. La classe MISD ne s'applique à aucune architecture actuelle.

	Execution Stream number and type			
Execution Stream number and type	Single, scalar	Single, array	Multiple, scalar	Multiple, array
Single, scalar	SISSES	SISSEA		
Single, array		SIASEA		
Multiple, scalar			MISMES	MISMEA
Multiple, array				

Table 2 : Taxonomie de Kuck. Le tableau a été repris sans modification de [ALM 89]. La taxonomie de Flynn a été élargie de plusieurs manières; Kuck, par exemple, remplace les flots de données par flots d'exécution et divise les flots (d'instruction et d'exécution) selon quantité et type. La taxonomie de Kuck vise plutôt le niveau matériel, tandis que celle de Flynn vise l'architecture et le jeu d'instructions de la machine.

	DATA MECHANISM	
Control Mechanism	Shared Memory	Private Memory (message passing)
control driven	von Neumann	cummunicating processes
pattern driven	logic	actors
demand driven	graph reduction	string reduction
data driven	dataflow (I-structure)	dataflow (tokens)

Table 3: Taxonomie de Treleaven pour architectures MIMD. Le tableau a été repris sans modification de [ALM 89]. Les architectures MIMD sont classifiées de façons diverses; Treleaven propose une classification selon les mécanismes utilisés pour le contrôle de l'exécution et les accès aux données.

ANNEXE 4 : SYNTAXE DU LANGAGE DE TROPES

D'abord nous présentons un exemple d'une base de connaissances : Le concept "voiture". Puis la syntaxis générale du langage de TROPES.

```
base
{
 automobilistes
 concepts      [voiture] ;
 points_de_vue [physique, mecanique, electrique]
}
concept
{
 voiture
   clef              [marque , date, prix ] ;
   points_de_vue     [physique, mecanique, electrique]
}
classe
{
      voiture.mecanique/mecanique
            carrosserie     un      chaine ;
            roues           ensemble_de entier ;
                        card    4 ;
            portiers        ensemble_de portiere_avant | portiere_arrier ;
                        card    2 ;
}
classe
{
      voiture.portes/mecanique
      sorte_de        # [mecanique] ;
      fenetre         un entier ;
      roues           ensemble_de entier ;
                  card 2 ;
}

classe
{
      voiture.fenetres/mecanique
      sorte_de        # [mecanique] ;
      fenetre1        un entier ;
      roues1          ensemble_de entier ;
                  card 2 ;
}

classe
{
      voiture.portes_nouvelles/mecanique
      sorte_de        # [portes] ;
      fenetre2        un entier ;
      roues2          ensemble_de entier ;
                  card 2 ;
}

classe
{
      voiture.electrique/electrique
            carrosserie1    un      chaine ;
```

```
        roues1          ensemble_de entier ;
                        card    4 ;
        portiers1       ensemble_de portiere_avant | portiere_arrier ;
                        card    2 ;
}
```

```
classe
{
      voiture.portes_electrique/electrique
            sorte_de       #[electrique] ;
            carrosserie1   un      chaine ;
            roues1         ensemble_de entier ;
                           card    4 ;
            portiers1      ensemble_de portiere_avant | portiere_arrier ;
                           card    2 ;
}

pass
{
      de = voiture.portes_electrique/electrique
      vers = voiture.portes_nouvelles/mecanique
}

pass
{
      de = voiture.portes_nouvelles/mecanique
      vers = voiture.portes_electrique/electrique
}
```

La syntaxis du langage de TROPES est :

```
/* ********************************************************************** */
/*                définition de la base                                   */
/* ********************************************************************** */

baseConnaissances : base concept attributsExternes classes passerelle ;

base     :       TBase   '{' nomident
                         TConcepts listeC ';'
                         TPoints   listePV '}'
                ;

nomident :       TIdent
                ;

listeC           :       '[' listevirgC ']'
            ;

listevirgC :     TIdent ',' listevirgC
          |      TIdent
            ;

listePV          :       '[' listevirgPV ']'
            ;

listevirgPV :            TIdent ',' listevirgPV
          |       TIdent
            ;

/* ********************************************************************** */
/*                définition de la liste de concepts                      */
/* ********************************************************************** */

concept          :
```

```
                /* vide */
        |
                unconcept concept
        ;

unconcept :     TConcept '{' TIdent listeClef listePVCon '}'
        ;

listeClef       :       TClef '[' nomClef ']' ';'
        ;

nomClef  :      TIdent ',' nomClef
        |
                TIdent
        ;

listePVCon :    TPoints '[' nomPVs ']'
          ;

nomPVs   :      TIdent ',' nomPVs
        |
                TIdent
        ;

/* ********************************************************************** */
/*                 Définition des attributs externes                      */
/* ********************************************************************** */

attributsExternes  :
                /* vide */
              |
                unAttributExterne attributsExternes
              ;

unAttributExterne  :
                TAttribut '{' TIdent TConcept '#' '[' "" TIdent "" ']' ';' resteAttributExterne '}'
             ;

resteAttributExterne :
             TCommentaire TIdent TNature TIdent
            |
             TNature TIdent
            ;

/* ********************************************************************** */
/*                 Définition des classes                                 */
/* ********************************************************************** */

classes         :       /* vide */
        |
                unClasse classes
        ;

/* ********************************************************************
        */
/* La définition de une classe est : nom du concept.nom de la classe/PV          */
/* ********************************************************************
        */
```

```
unClasse :       TClasse '{' TIdent '.' TIdent '/' TIdent defClasse '}'
          ;

/* ************************************************************************
          */
/* Définition de la sur classe et des attributs    :  TIdent = sorte_de                                  */
/* ************************************************************************
          */

defClasse :      TSorteDe '#' '[' TIdent ']' ';' lesAttributs
          |
                   lesAttributs
          ;

lesAttributs :
                   /* vide */
                   |
                   unAttribut lesAttributs
                   ;

unAttribut :     TIdent  facettes
                   ;

facettes         :
                    /* vide */
                   |
                   unFacette facettes
                   ;

unFacette  :      TUn reste ';'
          |
                   TEns reste ';'
                   |
                   TDom listeValues ';'
          |
                   TSauf listeValues ';'
                   |
                   TCard listeCard ';'
                   |
                   TDefaut listeValues ';'
                   |
                   TValeur listeValues ';'
          ;

listeValues :
                    /* vide */
                   |
                   TIdent ',' listeValues
                   ;

listeCard :
                   /* vide */
                   |
                   TIdent
                   |
                   TIdent ',' listeCard
                   ;
```

```
/* ********************************************************************** */
/*            Definition de la liste de type UN et Ensemble          */
/* ********************************************************************** */
reste    :          unType
         |
                    unType '|' reste
          |
                     unType '&' reste
         ;
unType           :          TIdent
         |
                    TIdent '.' TIdent '/' TIdent
          ;
/* ********************************************************************** */
/*            Définition de passerelles                                   */
/* ********************************************************************** */
passerelle :
                    /* vide */
                    |
                    unPasserelle passerelle
                    ;
/* ********************************************************************** */
/* Définition d'une passerelle :                                          */
/*            pass {  de   = classeDepart                                 */
/*                    vers = classeArrive }                               */
/*     defintion de classeDepart et classeArrive:                                 */
/*            classeDepart ::= concept.classe/point de vue                */
/*            classeArrive ::= concept.classe/point de vue                */
/* ********************************************************************** */
unPasserelle :
                    TPass '{' TDe    '=' TIdent '.' TIdent '/' TIdent
                            TVers '=' TIdent '.' TIdent '/' TIdent '}'
```

ANNEXE 5 : MISE EN ŒUVRE DE LA CLASSIFICATION EN PARALLELE

Définitions de quelques fonctions de la premièere version de la classification en parallèle d'objets dans TROPES. Pour cette réalisation, nous n'avons pas utilisé ni le tableau noir, ni le modèle acteur.

```
/* ************************************************************** */
/* Cette fonction l'ecrit au controle de dialogue que la classification */
/* dans le point de vue actuel a finie, et que le point de vue devient */
/* inactif                                                        */
/*            - jPV est le numero du point de vue actuel PVactuel */
/*             conceptActuel le numero du concept actuel          */
/* ************************************************************** */

direCDclassificationFinie()
{
        struct  attributLire    unAttribut;

        unAttribut.attribut     = 0;
        unAttribut.classe       = 0;
        unAttribut.PV           = jPV;
        unAttribut.concept      = conceptActuel;
        unAttribut.qui          = jPV;
        unAttribut.valider      = 0;

        write(pipeCD[1],&unAttribut,(sizeof(struct attributLire)));

}

/* ************************************************************** */
/* Cette fonction commence la classification de l'instance dans un concept */
/* La premiere choses a faire est choisir un point de vue, qui sera le PV */
/* actuel dans le processeur.                                     */
/*                 Entre : - le numero du concept                 */
/* ************************************************************** */

classification()
{
        struct attributs    *attributActuel ;

         PVactuel       = pointVueProcesseur(conceptActuel,jPV);

        classeActuel    = PVactuel->premClasse;
        iClasse         = 1;

      /* commence la classification */

        sortir = FAUX ;
        while ( !sortir) {
                typeClasseActuel        = SURE ;
                classeActuel->typeClasse = SURE ;
                obtenirInformation(conceptActuel,PVactuel,jPV,classeActuel,iClasse,
                        attributActuel,kAttribut);
                classeActuel = appariement(conceptActuel,PVactuel,jPV,classeActuel,
                                iClasse);
                if (classeActuel == (struct classe *)NULL) {
                        PVactuel->typePointVue = INACTIF;
                        PVactuel->nombrePassActifs = 0;
                        actualiserInformation();
                        sortir = VRAI;
                } else {
```

```
                        iClasse = posClasse(conceptActuel,PVactuel,jPV,classeActuel);
                        propagationMarques(conceptActuel,PVactuel,jPV,classeActuel,
                                iClasse,SURE);
                        actualiserInformation();
                }
        }
        direCDclassificationFinie();
                        /* le processus actuel l'ecrit au controle de
                           dialogue que la classification dans le PV
                           actuel a finie */
        exit(1);
           /* finalisation de la classification dans le point de vue
              actuel */
}

/* ************************************************************** */
/* Cette fonction demande les valeurs pour un attribut au controle de */
/* dialogue.                                                      */
/* ************************************************************** */

union typeUnion *demanderCDvaleurAttribut(int conceptActuel,struct listePV PVactuel,
int jPV,int iClasse,struct attributs attributActuel,int kAttribut,int valider)
{
        struct attributLire     unAttribut;
        struct reponse      unReponse;
        struct classe           *classePropagation;
        union  typeUnion        *valeurs;
        struct listeChar    *premCh,*totoCh,*dernCh;
        struct listeReel    *premRe,*totoRe,*dernRe;
        struct listeBool    *premBo,*totoBo,*dernBo;
        struct listeInt     *premIn,*totoIn,*dernIn;
        int                     fin,i;

        fin = FAUX;
        while (!fin) {
                read(pipePV[jPV][0],&unReponse,(sizeof(struct reponse)));
                if (unReponse.attribut == 0) { /* c'est une propagation */
                 classePropagation = adresseClasse(PVactuel,unReponse.classe);
                 if (unReponse.etat == SURE)
                   propagationSurePass(conceptActuel,PVactuel,jPV,
                                classePropagation);
                 else
                   propagationImpossible(conceptActuel,PVactuel,jPV,
                                        classePropagation);
                } else {
                 *valider= unReponse.valide;
                 fin     = VRAI;
                 valeurs = memoireTypeUnion();
                 i       = 1;
                 /* initialiser structures de données */
                 while (i <= unReponse.nombre) {
                   switch (unReponse.type) {
                        case CHAINE :
                                …
                                break;
                        case ENTIER :
                                …
                                break;
                        case BOOLEAN:
                                …
                                break;
                        case REEL   :
                                …
```

```
                                        break;
                                …
                                default     :
                                        break;
                            }
                            i++;
                        }
                    }
            }
            return(valeurs);
}


struct classe *appariement(int conceptActuel,struct listePV PVactuel,int jPV,struct classe classeActuel,int
iClasse)
{
            /* structures de données locales */

            classesPossibles            = 0;
            fils                        = (classeActuel->classePV)->fils;
            derniereClassePossible      = (struct classe *)NULL;
            while (fils != (struct arbreClasses *)NULL) {
              if ((fils->classeMeme)->typeClasse == POSSIBLE) {
                        classesPossibles++;
                        if (classesPossibles  == 1)
                          dernieClassePossible = fils->classeMeme;
               }
              fils = fils->frere;
            }
            if (classesPossibles  == 0)
                        return((struct classe *)NULL); /* point de vue inactif */
            if (classesPossibles  == 1) {
                        dernieClassePossible->typeClasse = SURE;
                        return(dernieClassePossible);/* la seule classe possible*/
            }
            nouvelleClasseActuel        = (struct classe *)NULL;
            fils                        = (classeActuel->classePV)->fils;
            existeClasseSure            = FAUX;
            while (!existeClasseSure) {
                         if (classesPossibles  == 1) {
                                    existeClasseSure     = VRAI;
                                    nouvelleClasseActuel = dernieClassePossible;
                        } else {
                                    totoClasse                  = premiereClassePossible(fils);
                                    dernieClassePossible        = totoClasse->classeMeme;
                                    attributActuel              = (totoClasse->classeMeme)->premAttrib;
                                     kAttribut                  = 1;
                                    typeClasse                  = POSSIBLE;
                                    valider                     = FAUX;
                                    while ((typeClasse != POSSIBLE) ||
                                                (attributActuel != (struct attributs *)NULL)) {
                                       if (attributActuel->valeursAttribut!=(union typeUnion *)NULL) {
                                                valeurs = valeurAttribValue(attributsValues,attributActuel);
                                                if ( valeurs == (union typeUnion *)NULL) {
                                                   valeurs = demanderCDvaleurAttribut(conceptActuel,
                                                    PVactuel,jPV,iClasse,attributActuel,kAttribut,&valider);
                                                   if (valider == PASVALIDE) {
                                                   (totoClasse->classeMeme)->typeClasse = IMPOSSIBLE;
                                                            if ((totoClasse->classeMeme)->premPass !=
                                                                        (struct passerelle *)NULL) {
                                                                                propaguerPasserelle(conceptActuel,
                                                                      totoClasse->classeMeme,IMPOSSIBLE);
                                                            }
```

```
                                        typeClasse = IMPOSSIBLE;
                                        classesPossibles--;
                                    } else {
                                        attributActuel->valeursAttribut = valeurs;
                                    }
                                    attributsValues = insererAttributValue(attributsValues,
                                                attributActuel,valeurs);
                                    } else {
                                     if (validerValeurAP(valeurs,attributActuel)==VALIDE){
                                        attributActuel->valeursAttribut = valeurs;
                                      } else {
                                     (totoClasse->classeMeme)->typeClasse=IMPOSSIBLE;
                                     if ((totoClasse->classeMeme)->premPass !=
                                        (struct passerelle *)NULL) {
                                                propaguerPasserelle(conceptActuel,
                                                totoClasse->classeMeme,IMPOSSIBLE);
                                     }
                                     typeClasse = IMPOSSIBLE;
                                     classesPossibles--;
                                     }
                                   }
                                }
                                attributActuel = attributActuel->suiv;
                                kAttribut ++;
                            }
                           if (typeClasse == POSSIBLE) { /* la classe devient SURE */
                                (totoClasse->classeMeme)->typeClasse = SURE;
                                 if ((totoClasse->classeMeme)->premPass !=
                                    (struct passerelle *)NULL) {
                                        propaguerPasserelle(conceptActuel,
                                            totoClasse->classeMeme,SURE);
                                }
                                derniereClassePossible          = totoClasse->classeMeme;
                                nouvelleClasseActuel            = totoClasse->classeMeme;
                                existeClasseSure                = SURE;
                            }
                        }
                }

                if (nouvelleClasseActuel != (struct classe *)NULL) {
                        nouvelleClasseActuel->typeClasse = SURE;
                if (nouvelleClasseActuel->premPass != (struct passerelle *)NULL) {
                        propaguerPasserelle(conceptActuel,nouvelleClasseActuel,SURE);
                }
        }
        return(nouvelleClasseActuel);
}

propagationImpossible(int  conceptActuel,struct listePV *PVactuel,int jPV,struct classe *classeActuel)
{
        struct arbreClasses     *fils;

        classeActuel->typeClasse = IMPOSSIBLE;
        if (classeActuel->premPass != (struct passerelle *)NULL) {
          propaguerPasserelle(conceptActuel,classeActuel,IMPOSSIBLE);
        }
        fils = (classeActuel->classePV)->fils;
        while (fils != (struct arbreClasses *)NULL ) {
       propagationImpossible(conceptActuel,PVactuel,jPV,fils->classeMeme);
          fils = fils->frere;
        }
}
```

```
propagationSure(int  conceptActuel,struct listePV *PVactuel,int jPV,struct classe *classeActuel)
{
        struct arbreClasses         *fils,*pere;

        classeActuel->typeClasse = SURE;
        if (classeActuel->premPass != (struct passerelle *)NULL) {
          propaguerPasserelle(conceptActuel,classeActuel,SURE);
        }
        pere  = (classeActuel->classePV)->pere;
        fils    = pere->fils;
        while (fils != (struct arbreClasses *)NULL) {
         if ((fils->classeMeme)->typeClasse != POSSIBLE) { /* deja marquee */
            fils = fils->frere;
         } else {
        propagationImpossible(conceptActuel,PVactuel,jPV,fils->classeMeme);
            fils = fils->frere;
         }
        }
}

propagationSurePass1(int conceptActuel,struct listePV *PVactuel,int jPV,struct classe *classeActuel)
{
        struct arbreClasses         *pere;

      propagationSure(conceptActuel,PVactuel,jPV,classeActuel);
        pere = (classeActuel->classePV)->pere;
        while ((pere != (struct arbreClasses *)NULL) &&
             ((pere->classeMeme)->typeClasse != POSSIBLE)) {
         propagationSurePass1(conceptActuel,PVactuel,jPV,pere->classeMeme);
            pere = (classeActuel->classePV)->pere;
        }
}

propagationSurePass(int  conceptActuel,struct listePV *PVactuel,int jPV,struct classe *classeActuel)
{
        struct arbreClasses         *pere,*fils;

        if (classeActuel->typeClasse != POSSIBLE) {
         classeActuel->typeClasse = SURE;
         pere                = (classeActuel->classePV)->pere;
         fils                = pere->fils;
         while (fils != (struct arbreClasses *)NULL) {
          if ((fils->classeMeme)->typeClasse != POSSIBLE) { /* deja marquee */
                fils = fils->frere;
           } else {
                propagationImpossible(conceptActuel,PVactuel,jPV,fils->classeMeme);
                fils = fils->frere;
           }
         }
         if ((pere != (struct arbreClasses *)NULL) &&
              ((pere->classeMeme)->typeClasse != POSSIBLE)) {
                propagationSurePass1(conceptActuel,PVactuel,jPV,pere->classeMeme);
         }
        }
}

propagationMarques(int   conceptActuel,struct listePV *PVactuel,int jPV,struct classe *classeActuel,int
iClasse,int typeMarque)
{
        if (typeMarque == SURE)
        propagationSure(conceptActuel,PVactuel,jPV,classeActuel);
        else
        propagationImpossible(conceptActuel,PVactuel,jPV,classeActuel);
```

```
}

/* ********************************************************************** */
/* Fonction du controle de dialogue                                       */
/* ********************************************************************** */

dialogue(unAttributActuel)
struct unAttribut *unAttributActuel;
{
        union typeUnion                 *valeur;
        int                             fin,valide;
        struct tousAttribLu     *ilEst;

        if (unAttributActuel->attribut != 0) {
          trouverAttribut(unAttributActuel);
          if ((ilEst= existe(unAttributActuel))!=(struct tousAttribLu *)NULL) {
                valeur = valeurAttribut(ilEst,unAttributActuel);
                valide = validerValeur(valeur,unAttributActuel);
                unAttributActuel->type = (ilEst->attribut)->type;
           }
          else {
                fin = PASVALIDE;
                valide = PASVALIDE;
                while (!fin) {
                        ecrireNom(unAttributActuel);
                        valeur = lireValeur(unAttributActuel);
                        insererValeurAt(valeur,unAttributActuel);
                        if (valeur != (union typeUnion *)NULL) {
                          valide = validerValeur(valeur,unAttributActuel);
                          if (unAttributActuel->valider == FAUX)
                            fin = VALIDE;
                          else
                            fin = valide;
                        } else
                            fin = VALIDE;
                }
           }
          retourValeur(valeur,unAttributActuel,valide);
        }
}

/* ********************************************************************** */
/* Fonction pour gerer les processeurs, et la classification d'une        */
/* instance dans la base de connaissances.                                */
/*              - creation de processus dont chacun a son propre          */
/*                point de vue.                                           */
/*              - gerer le probleme de termination du parallelisme        */
/* ********************************************************************** */


commencerClassification()
{
      int                       i;
      int                       fin = FAUX;
      int                       processusFin = 0; /* pour savoir combien du
                                                     de processus ils sont fini*/
      int                       numeroPV ;
      struct unAttribut         unAttributActuel;
      int                       nombrePV;

                                /* creation du pipe pour le controle de dialogue */
      pipe(pipeCD);
```

```
    for (conceptActuel = 0; conceptActuel < nomC1; conceptActuel++) {
          nombrePV        = listeConcep[conceptActuel].nombrePV;
       for (numeroPV   = 0; numeroPV <= nombrePV ; numeroPV++)
            pipe(pipePV[numeroPV]);
          PVactuel           = listeConcep[conceptActuel].premPV;
          jPV                = 1;
          numeroProcessus = 1;

                             /* Dans ce while le system fait le placement
                                de taches (classification d'une instance
                                dans un point de vue) dans les differentes
                                processus. Pour chaque point de vue il y a
                                un processus. */

          while (PVactuel != (struct listePV *)NULL) {
             PVactifs[jPV]    = ACTIF ;
             if (fork() == 0) {
                   for (i = 0; i <= nombrePV; i++) {
                     if ( i != jPV )
                        close(pipePV[i][0]);
                   }
                   close(pipeCD[0]);
                   classification();
             }
             PVactuel = PVactuel->suiv;
             jPV ++;
             numeroProcessus ++;
          }
     }
    while (!fin) {
          if (quelAttribut(pipeCD[0],&unAttributActuel) == 1) {
             if (unAttributActuel.attribut == 0)  {
                   processusFin ++;
                   PVactifs[unAttributActuel.qui] = INACTIF;
                   if (processusFin == nombrePV)
                      fin = VRAI;              /* fin de la classification */
             }
             else {
                   dialogue(&unAttributActuel);
             }
          }
     }
     exit(0);
    /* la classification de l'instance a finie dans tous les points de
          vue, et le systeme peut finir. */
}
```

BLIBLIOGRAPHIE

[AHO 74] Aho Alfred, Hopcroft John, Ullman Jeffrey
The design and analysis of computer algorithms
Addison-Wesley, 1974

[AKL 89] Akl Selim G.
The Design and Analysis of Parallel Algorithms
Prentice Hall International, Inc, 1989

[ALM 89] Almasi G.S., Gottlieb A.
Highly Parallel Computing
The Benjamin/Cummings Publishing Company, Inc. 1989

[AGH 86] Agha Gul
A Model of Concurrent Computation in Distributed Systems
MIT Press, Cambridge, MA, 1986

[AND 88] André Françoise, Pazat Jean-Louis
Le placement de tâches sur des architectures parallèles
T.S.I. Technique et Science Informatiques, no. 7,
pp 385-401, 1988

[ARR 92] Arrouye Y. et al.
Manuel du Méganode
Rapport Technique, RT 79, LMC - IMAG, Grenoble,
Avril 1992

[ATH 88] Athas William C., Seitz Charles L.
Multicomputers: Message-passing concurrent computers
IEEE transaction on computers, Août 1988

[ATH 87] Athas William C.
Fine grain concurrent computation
Tech Rep. No. 5242:TR/87, California Institute of Technology.
Mai 1987

[BAI 87] Bailey Duane A., Cuny Janice E., MacLeod Bruce
Reducing Communication Overhead: A parallel Code Optimization
Journal of parallel and distributed computing no. 4,
pp 505-520, 1987

[BAL 80] Balzer R., Erman L.D., London P.E., Williams C.
Hearsay-III: A domain-independant framework for expert systems
First National Conference on Artificial Intelligence. AAAI - 1980

[BAR 90] Baron R, Black D, Colosky W. et al.
MACH kernel interface Manual
Computer Science, Carnegie Mellon University, 1990

[BLA 90] Black David
Scheduling Support for Concurrency and Parallelism in the Mach
Operating System
IEEE Computer, Mai 1990

[BER 87] Berman Francine, Snyder Lawrence
On mapping parallel algorithms into parallel architectures

Journal of parallel and distributed computing no. 4,
pp 439-458, 1987

[BIR 84] Birrel, A.D., Nelson B.D.
Implementing Remote Procedure Calls
ACM Transactions on Computer Systems, vol 2, pp 39-59. Février 84

[BIS 89] Bisiani Roberto, Forin A.
Parallelization of Blackboard Architectures and the Agora System
dans Blackboard Architectures and Applications
Academic Press, Inc. 1989

[BOI 90] Boissier Olivier
La cooperation entre systèmes à base de connaissances
Rapport de Recherche 811-I-IMAG-96-LIFIA, Grenoble, Février 1990

[CAP 93] Capponi Cécile, Chaillot Mathias
Construction incrémentale d'une base de classes correcte du point de vue des types
Journées Acquisition - Validation - Apprentissage du PRC-GDR-IA du CNRS, Saint-Raphaël, Mars 1993

[CAR 88] Caromel Denis
A Generalized Model for Concurrent and Distributed Object-Oriented Programming
Proceedings of the ACM SIGPLAN
Workshop on Object-Based Concurrent Programming, 1988, SIGPLAN Notices, 24, 4, Avril 1989

[CHA 92] Chaillot Mathias
Une approche client/serveur pour la conception d'un système blackboard
Premieres rencontres nationales des Jeunes Chercheurs en Intelligence Artificielle
Rennes, Septembre 1992

[CLA 85] Clancey W.J.
Heuristic Classification
AI Journal 27, No. 4, 1985

[COM 89] Comon Hubert
Notes du Cours de Systèmes de Réécriture
ENSIMAG, Grenoble, 1989

[COR 89a] Corkill Daniel D.
Advanced architectures: Concurrency and Parallelism
dans Blackboard Architectures and Applications
Academic Press, Inc. 1989

[COR 89b] Corkill Daniel D.
Design Alternatives for Parallel and Distributed
Blackboard Systems. Academic Press, Inc. 1989

[COR 83] Corkill Daniel D., Lesser V.R.
The Distributed Vehicle Monitoring Testbed: a tool for investigating Distributed Problem Solving Network
AI Magazine. pp 15-33, 1983

[COU 88] Coulouris G.F., Dollimore J.

Distributed Systems Concepts qnd Design
Reading, MA, Addison Wesley, 1988

[ENG 88] Engelmore Robert, Morgan Tony
Blackboard Systems.
Addison-Wesley Publishing Company Inc., Menlo Park, CA 1988

[ERI 88] Eriksen Ole
A termination detection protocol and its formal verification
Journal of parallel and distributed computing no. 5,
pp 82-91, 1988

[ERM 81] Erman L.D., London P.E., Fickas S.F.
The design and example use of Hearsay III
IJCAI, pp 409-415, Vancouver, 1981

[ERM 80] Erman L.D., Hayes-Roth F., Lesser V.R., Reddy D.R.
The Hearsay-II speech understanding system: integrating knowledge to resolve uncertainty
ACM Computing Surveys 12(2), pp. 213-53, 1980

[ERM 75] Erman L.D., Lesser V.R.
A Multi-level Organization for Problem
Solving using many diverse Cooperating Sources of Knowledge
Proceedings of the 4th IJCAI Conference pp 483-490, 1975

[FAH 89] Fahlman Scott E.
What Can Massively Parallel Architectures Bring to AI
Lectures Notes in Artificial Intelligence. AI 88. 2nd. Australian Joint Artificial Intelligence. Adelaide, Australia, Novembre 1988

[FEN 77] Fennel R.D., Lesser V.R.
Parallelism in Artificial Intelligence
Problem Solving: A Case Study of Hersay II
IEEE Transactions on Computers, Vol C-26 (2) pp 98-111, 1977

[FIK 85] Fikes R., Kehler T.
The Role of Frame-Based Representation in Reasoning
Communications of the ACM, Vol 28, No. 9, 1985

[FLY 72] Flynn Michel
Some Computer Organizations and their Effectiveness
IEEE Transactions on Computers, Vol C-21, 1972

[GAS 88] Gasser L., Bond A. H.
Readings in Distributed Artificial Intelligence, 1988

[GON 91] Gonzalez Néstor
PARX : Noyau du système pour les ordinateurs massivement parallèles
Thèse de Docteur, I.N.P.G., 1991

[GOS 91] Goscinski A.
Distributed Operating Systems - The Logical Design
Addison-Wesley, 1991

[HAE 86] Hayes John P et al.
A microprocessor-based Hypercube Supercomputer
IEEE Micro Octobre 1986

[HAG 92] Haghjoo Mostafa, Papazoglou Mike
TrActorS : A Transactional Actor System for Distributed Query Processing
Transaction on parallel and distributed Systems, IEEE, 1992

[HAR 91] HARNESS, European Distributed System Integration Project.
ESPRIT Project 5279. HARNESS/ACC/MI/CGI/001/1.2 1991

[HAY 86] Hayes-Roth B., Garvey A., Johnson M.V., Hewett M.
A Layered Environment for Reasoning about Action
Report KSL-86-38, Comp. Science Dpt, Stanford Univ. Octobre 1986

[HAY 84] Hayes-Roth B.
BB1: An Architecture for Blackboard Systems that
Control, Explain, and Learn about their own Behavior
Knowledge Systems Laboratory Technical report HPP-84-16,
Stanford, 1984

[HEW 77] Hewitt Carl
Viewing Control Structures as Patterns of Passing Messages
Journal of Artificial Intelligence, 323-364, Juin, 1977

[HEW 73] Hewitt Carl, Bishop P., Steiger R.
A universal modular ACTOR formalism for artificial intelligence
Proceedings of 3th IJCAI, Stanford, California, 1973

[HIl 86] Hillis W. Daniel, Steele Guy
Data parallel algorithms
Communications of the ACM, vol. 29, no. 12
Decembre 1986

[HOC 84] Hockney R.W., Jesshope C.R.
Parallel computers
Adam Hilger Ltd., Bristol, 1984

[HWA 87] Hwang Kai et al.
Computer architectures for Artificial Intelligence Processing
IEEE Computer, 19-27, Janvier 1987

[HWA 86] Hwang Kai, Briggs Faye A.
Computer Architecture and Parallel Processing
McGraw Hill, 1986

[INM 88] Inmos Ltda.
Transputer Development System
Pentice Hall, 1988

[JAG 89] Jagannathan Vasudevan
Realizing the Concurrent Blackboard Model
dans Blackboard Architectures and Applications
Academic Press, Inc. 1989

[KAF 90] Kafura Dennis, Lee Keung Hae
ACT++: Building a Concurrent C++ using Actors
Journal of Object-Oriented Programming, 1990

[KAR 90] Karp Richard, Ramachandran Vijaya
Parallel Algorithms for shared-memory machines
Elsevier Science Publisher, chapitre 17, 1990

[KAR 87] Karp Alan H.
Programming for parallelism
IEEE Transaction on Computers, Mai 1987

[KAY 86] Kayser D.
Représentation des Connaissances en Intelligence Artificielle
Ecole d'été de l'ARC, Bonas, 1986

[KOE 89] Koenig Anne

TRAM: une architecture blackboard dediée à la robotique mobile
Thèse d'Informatique. Institut National Polytechnique de Grenoble, 1989

[KOE 88] Koenig Anne, Crochon Elisabeth
TRAM: tableau noir pour robotique mobile
8èmes jounées internationales sur les systèmes experts et leurs applications. Avignon, France, Mai 1988

[KON 80] Konolige K., Nilsson N.
Multiple Agent Planning Systems
AAAI Conference, pp 138-142, 1980

[KRA 91] Kraus Sarit, Wilkenfild Jonathan
Negotiation over time in multi-agents environment : preliminary report
12th. Inter. Joint Conference on Artificial Intelligence, Syndney, Australie, 1991

[LAM 83] Lamport Leslie
Specifying concurrent program modules
ACM Transaction on Programming Languages and Systems, vol. 5, no. 2, pp 190-222, Avril 1983

[LAN 91] Langue Tsobngy , Bertrand Yves
PARX : Architecture de noyau de système d'explotation parallèle
Thèse de Docteur, I.N.P.G., 1991

[LAX 84] Laxmi N. Bhuyan, Dharma P. Agrawal
Generalized Hypercube and Hypercubes Structures for a Computer Network
IEEE Transaction on computers, vol. c-33, no. 4, 1984

[LEN 91] Lenzerini Maurizio, Nardi Daniele
Inheritance Hierarchies in Knowledge Representation and Programming languages
pp 257-274, John Wiley & Sons, Chichester, 1991

[LIE 87] Lieberman Henry
Concurrent Object-Oriented Programming in ACT 1,
dans : Object-Oriented Concurrent Programming
A. Yonezawa and M. Tokoro (eds.), MIT Press, Cambridge, MA, 1987

[MAN 85] Mani Chandy, Lamport Leslie
Distributed Snapshots: Determining global states of distributed systems
ACM transaction on computer systems, vol. 3, no. 1, pp 63-75, Février 1985

[MAR 89] Mariño Olga
Classification d'objets dans un modèle multi-points-de-vue
D.E.A. d'Informatique. INPG, Grenoble, 1989

[MAS 89] Masini Gérald, Napoli Amedeo, Colnet Dominique, Léonard Daniel, Tombre Karl
Les langages à objets
InterEditions, Paris, 1989

[MEN 91] Mendieta Francisco
LDBA: Lenguaje Distribuído Basado en Actores
Universidad de los Andes, SantaFe de Bogotá, 1991

[MIN 75] Minsky M.
A framework for Representing Knowledge
The Psychology of Computer Vision, P.H. Winston Ed., Mac Graw Hill 1975

[MOL 92] Moldovan Dan, Lee Wing, Lin Changhwa
SNAP / A marker-propagation architecture for knowledge processing
IEEE Transactions on Parallel and Distributed Systems
Vol. 3, No. 4, pp 397-410, Julliet 1992

[MUN 91] Muninda P. Singh
Towards a formal theory of communication for multiagents systems
12th. Inter. Joint Conference on Artificial Intelligence, Sydney, Australie, 1991

[NEW 72] Newell A., Simon H.A.
Human Problem Solving.
Englewood Cliffs, 1972

[NII 89] Nii H.P., Aiello N., Rice J.P.
Experiments on Cage and Poligon: Mesuring the Perfomance of Parallel Blackboard Systems
dans Distributed Artificial Intelligence II.
Pitman Publishing Ltd. and Morgan Kauffmann 1989

[NII 86a] Nii H.P.
CAGE and POLIGON: two frameworks for blackboard-based concurrent problem solving
Technical Report KSL-86-41, Stanford, 1986

[NII 86b] Nii H.P.
Balckboard Systems: The blackboard model of problem solving and the evolution of blackboard architectures. Part One
Artificial Intelligence Magazine, 38-53, Août 1986

[NII 86c] Nii H.P.
Balckboard Systems: The blackboard model of problem solving and the evolution of blackboard architectures. Part Two
Artificial Intelligence Magazine, 82-106, Août 1986

[NII 79] Nii H.P., Nelleke Aiello
AGE : A knowledge-based program for building knowledge-based programs
Proceedings of 6th International Joint Conference on Artificial Intelligence, 1979

[QUI 92a] Quintero Alejandro
Los sistemas de Blackboard
Facultad de Ingenieria, Universidad de los Andes, Bogota, Colombia

1992

[QUI 92c] Quintero Alejandro, Alvarez Gloria
Diseño de un Sistema de Blackboard sobre GEMINIS
Facultad de Ingenieria, Universidad de los Andes, Bogota, Colombia
1992

[QUI 92d] Quintero Alejandro, Mendieta Francisco
LDBA: Lenguaje Distribuído Basado en Actores
Congreso LatioAmericano de Informática, islas Canarias, España,
Août, 1992

[QUI 92e] Quintero Alejandro, Mendieta Francisco
GEMINIS : Conjunto de Herramientas para la Implantación de
Sistemas Distribuidos bajo UNIX
Facultad de Ingenieria, Universidad de los Andes, Bogotá, Colombia
1992

[QUI 92f] Quintero Alejandro, Cobo Luis
Diseño de un conjunto de herramientas para el desarrollo de sistemas
Distribuidos bajo UNIX
Facultad de Ingenieria, Universidad de los Andes, Bogota, Colombia
1992

[QUI 89] Quintero Alejandro
Parallelisation de la maintenance de la vérité tirant parti des composantes
fortement connexes
D.E.A. d'informatique INPG, Grenoble 1989

[QUI 85] Quinton Patrice
Les hyper ordinateurs
La Recherche, no. 167, juin 1985

[RAS 86] Rashid R.F.
Threads of a New System
UNIX review, 1986

[REC 88] Rechenmann François
SHIRKA : système de gestion de bases de connaissances centrées-objet
Manual d'utilisation 1988

[RIC 89] Rice James
The design and implementation of Poligon, a high-
performance, concurrent Blackboard system shell
Report KSL-89-37, Stanford, novembre 1989

[RIC 86] Rice James
The Poligon User's Manual
Report KSL-86-10, Stanford, 1986

[ROC 84] Roche C.
EAQUE-LRO Génération de systèmes experts. Application à des
problèmes d'ordonnancement
Thèse de 3ème cycle, INPG Grenoble, 1984

[SAN 90] Sandberg Jonathan
Design of the PRAM Nerwork
IEEE, Transac. on Parallel and distributed systems, 1990

[SEI 85] Seitz Charles
The Cosmic Cube
Communication of the ACM, Vol. 28, No. 1, Janvier 1985

[SHA 88] Shasha Dennis, Snir Marc
Efficient and correct execution of parallel programs
that share memory
ACM Transaction on Programming Languages and
Systems, vol. 10, no. 2, pp 282-312, Avril 1988

[SIM 77] Simon H.A.
Scientific Discovery and the Psychology of Problem Solving
Models of Discovery, D. Reidel Publishing Company, 1977

[STR 86] Stroustrup Bjarne
The C++ Programming Language
Addison-Wesley, Menlo-Park, CA, 1986

[TAL 91] Talbi E-G, Muntean T.
Méthodes de placement statique des processus
sur architectures parallèles
Technique et Science Informatique, AFCET Bordas, 1991

[TAN 92] Tanenbaum A.S.
Modern Operating Systems
Prentice Hall, Inc 1992

[TEN 87] Teng Shang-Hua, Wang Bin
Parallel algorithms for message decomposition
Journal of parallel and distributed computing no. 4,
pp 231-249, 1987

[TEV 87] Tevanian A., Rashid R.
Mach : A basis for Future UNIX Development
Carnegie-Mellon University, Departament of Computer Science
1987

[TUC 88] Tucker Lewis, Roberston Georges
Architecture and Applications of the Connection Machine
IEEE Computer, Août 1988

[WAI 90] Waille Ph., Muntean T.
Introduction à l'architecture des machines supernode
La lettre du transputer No. 7, 1990

[YON 87] Yonezawa, Akinori, Shibayama, Etsuya, Tokada, T., Honda, Y.
Modelling and Programming in an Object-Oriented Concurrent Language
ABCL/1, dans Object-Oriented Concurrent Programming
A. Yonezawa y M. Tokoro (eds.) MIT Press,
Cambridge, MA, pp. 55 - 89, 1987

[ZIE 90] Ziébelin Danielle
Raisonnement sur les contraintes; application en ordonnancement
Thèse d'Informatique. Université Joseph Fourier
Grenoble 1, 1990