Université de Montréal

Un système multi-paradigme pour la manipulation des connaissances utilisant la théorie des graphes conceptuels

par

Adil Kabbaj

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures
en vue de l'obtention du grade de
Philosophiæ Doctor (Ph.D.)
en informatique

Avril 1996

© Adil Kabbaj, 1996

QA
76
U54
1996
V.08

Nom KABBAJ Adil

Dissertation Abstracts International est organisé en catégories de sujets. Veuillez s.v.p. choisir le sujet qui décrit le mieux votre thèse et inscrivez le code numérique approprié dans l'espace réservé ci-dessous.

INFORMATIQUE — SUJET

0984 — CODE DE SUJET

U·M·I

Catégories par sujets

HUMANITÉS ET SCIENCES SOCIALES

COMMUNICATIONS ET LES ARTS
Architecture 0729
Beaux-arts 0357
Bibliothéconomie 0399
Cinéma 0900
Communication verbale 0459
Communications 0708
Danse 0378
Histoire de l'art 0377
Journalisme 0391
Musique 0413
Sciences de l'information 0723
Théâtre 0465

ÉDUCATION
Généralités 515
Administration 0514
Art 0273
Collèges communautaires 0275
Commerce 0688
Économie domestique 0278
Éducation permanente 0516
Éducation préscolaire 0518
Éducation sanitaire 0680
Enseignement agricole 0517
Enseignement bilingue et multiculturel 0282
Enseignement industriel 0521
Enseignement primaire. 0524
Enseignement professionnel 0747
Enseignement religieux 0527
Enseignement secondaire 0533
Enseignement spécial 0529
Enseignement supérieur 0745
Évaluation 0288
Finances 0277
Formation des enseignants 0530
Histoire de l'éducation 0520
Langues et littérature 0279
Lecture 0535
Mathématiques 0280
Musique 0522
Orientation et consultation 0519
Philosophie de l'éducation 0998
Physique 0523
Programmes d'études et enseignement 0727
Psychologie 0525
Sciences 0714
Sciences sociales 0534
Sociologie de l'éducation 0340
Technologie 0710

LANGUE, LITTÉRATURE ET LINGUISTIQUE
Langues
- Généralités 0679
- Anciennes 0289
- Linguistique 0290
- Modernes 0291

Littérature
- Généralités 0401
- Anciennes 0294
- Comparée 0295
- Médiévale 0297
- Moderne 0298
- Africaine 0316
- Américaine 0591
- Anglaise 0593
- Asiatique 0305
- Canadienne (Anglaise) 0352
- Canadienne (Française) 0355
- Germanique 0311
- Latino-américaine 0312
- Moyen-orientale 0315
- Romane 0313
- Slave et est-européenne 0314

PHILOSOPHIE, RELIGION ET THÉOLOGIE
Philosophie 0422
Religion
- Généralités 0318
- Clergé 0319
- Études bibliques 0321
- Histoire des religions 0320
- Philosophie de la religion 0322

Théologie 0469

SCIENCES SOCIALES
Anthropologie
- Archéologie 0324
- Culturelle 0326
- Physique 0327

Droit 0398
Économie
- Généralités 0501
- Commerce-Affaires 0505
- Économie agricole 0503
- Économie du travail 0510
- Finances 0508
- Histoire 0509
- Théorie 0511

Études américaines 0323
Études canadiennes 0385
Études féministes 0453
Folklore 0358
Géographie 0366
Gérontologie 0351
Gestion des affaires
- Généralités 0310
- Administration 0454
- Banques 0770
- Comptabilité 0272
- Marketing 0338

Histoire
- Histoire générale 0578
- Ancienne 0579
- Médiévale 0581
- Moderne 0582
- Histoire des noirs 0328
- Africaine 0331
- Canadienne 0334
- États-Unis 0337
- Européenne 0335
- Moyen-orientale 0333
- Latino-américaine 0336
- Asie, Australie et Océanie 0332

Histoire des sciences 0585
Loisirs 0814
Planification urbaine et régionale 0999
Science politique
- Généralités 0615
- Administration publique 0617
- Droit et relations internationales 0616

Sociologie
- Généralités 0626
- Aide et bien-âtre social 0630
- Criminologie et établissements pénitentiaires 0627
- Démographie 0938
- Études de l' individu et de la famille 0628
- Études des relations interethniques et des relations raciales 0631
- Structure et développement social 0700
- Théorie et méthodes. 0344
- Travail et relations industrielles 0629

Transports 0709
Travail social 0452

SCIENCES ET INGÉNIERIE

SCIENCES BIOLOGIQUES
Agriculture
- Généralités 0473
- Agronomie. 0285
- Alimentation et technologie alimentaire 0359
- Culture 0479
- Élevage et alimentation 0475
- Exploitation des pâturages 0777
- Pathologie animale 0476
- Pathologie végétale 0480
- Physiologie végétale 0817
- Sylviculture et faune 0478
- Technologie du bois 0746

Biologie
- Généralités 0306
- Anatomie 0287
- Biologie (Statistiques) 0308
- Biologie moléculaire 0307
- Botanique 0309
- Cellule 0379
- Écologie 0329
- Entomologie 0353
- Génétique 0369
- Limnologie 0793
- Microbiologie 0410
- Neurologie 0317
- Océanographie 0416
- Physiologie 0433
- Radiation 0821
- Science vétérinaire 0778
- Zoologie 0472

Biophysique
- Généralités 0786
- Medicale 0760

SCIENCES DE LA TERRE
Biogéochimie 0425
Géochimie 0996
Géodésie 0370
Géographie physique 0368
Géologie 0372
Géophysique 0373
Hydrologie 0388
Minéralogie 0411
Océanographie physique 0415
Paléobotanique 0345
Paléoécologie 0426
Paléontologie 0418
Paléozoologie 0985
Palynologie 0427

SCIENCES DE LA SANTÉ ET DE L'ENVIRONNEMENT
Économie domestique 0386
Sciences de l'environnement 0768
Sciences de la santé
- Généralités 0566
- Administration des hipitaux 0769
- Alimentation et nutrition 0570
- Audiologie 0300
- Chimiothérapie 0992
- Dentisterie 0567
- Développement humain 0758
- Enseignement 0350
- Immunologie 0982
- Loisirs 0575
- Médecine du travail et thérapie 0354
- Médecine et chirurgie 0564
- Obstétrique et gynécologie 0380
- Ophtalmologie 0381
- Orthophonie 0460
- Pathologie 0571
- Pharmacie 0572
- Pharmacologie 0419
- Physiothérapie 0382
- Radiologie 0574
- Santé mentale 0347
- Santé publique 0573
- Soins infirmiers 0569
- Toxicologie 0383

SCIENCES PHYSIQUES
Sciences Pures

Chimie
- Généralités 0485
- Biochimie 487
- Chimie agricole 0749
- Chimie analytique 0486
- Chimie minérale 0488
- Chimie nucléaire 0738
- Chimie organique 0490
- Chimie pharmaceutique 0491
- Physique 0494
- PolymÇres 0495
- Radiation 0754

Mathématiques 0405
Physique
- Généralités 0605
- Acoustique 0986
- Astronomie et astrophysique 0606
- Electronique et électricité 0607
- Fluides et plasma 0759
- Météorologie 0608
- Optique 0752
- Particules (Physique nucléaire) 0798
- Physique atomique 0748
- Physique de l'état solide 0611
- Physique moléculaire 0609
- Physique nucléaire 0610
- Radiation 0756

Statistiques 0463

Sciences Appliqués Et Technologie

Informatique 0984
Ingénierie
- Généralités 0537
- Agricole 0539
- Automobile 0540
- Biomédicale 0541
- Chaleur et ther modynamique 0348
- Conditionnement (Emballage) 0549
- Génie aérospatial 0538
- Génie chimique 0542
- Génie civil 0543
- Génie électronique et électrique 0544
- Génie industriel 0546
- Génie mécanique 0548
- Génie nucléaire 0552
- Ingénierie des systèmes 0790
- Mécanique navale 0547
- Métallurgie 0743
- Science des matériaux 0794
- Technique du pétrole 0765
- Technique minière 0551
- Techniques sanitaires et municipales 0554
- Technologie hydraulique 0545

Mécanique appliquée 0346
Géotechnologie 0428
Matières plastiques (Technologie) 0795
Recherche opérationnelle 0796
Textiles et tissus (Technologie) 0794

PSYCHOLOGIE
Généralités 0621
Personnalité 0625
Psychobiologie 0349
Psychologie clinique 0622
Psychologie du comportement 0384
Psychologie du développement 0620
Psychologie expérimentale 0623
Psychologie industrielle 0624
Psychologie physiologique 0989
Psychologie sociale 0451
Psychométrie 0632

Université de Montréal

Faculté des études supérieures

Cette thèse intitulée:

Un système multi-paradigme pour la manipulation des connaissances utilisant la théorie des graphes conceptuels

présentée par:

Adil Kabbaj

a été évaluée par un jury composé des personnes suivantes:

Paul Bratley,	président-rapporteur
Claude Frasson,	directeur de recherche
Guy Gouardères,	membre du jury
John F. Sowa,	examinateur externe

Thèse acceptée le 96-06-03

Sommaire

Dans des domaines comme le génie logiciel, l'intelligence artificielle, les systèmes tutoriels intelligents et les systèmes multi-agents, plusieurs types de connaissances sont manipulés dans un même système avec souvent des formalismes différents, rendant difficile l'intégration, la communication, l'utilisation et le partage des connaissances et des expertises, au sein du système et/ou entre les systèmes.
Un système multi-paradigme, basé sur un formalisme uniforme serait donc approprié.

John Sowa propose la théorie des Graphes Conceptuels (GC) comme un *formalisme universel de représentation des connaissances*. Cette théorie est présentée non seulement comme un fondement logique pour les réseaux sémantiques mais aussi comme une théorie du traitement de l'information chez l'humain et la machine. Une communauté de chercheurs s'est formée pour analyser, étendre et utiliser la théorie des GC dans différents domaines (base de données, base de connaissances, génie logiciel, système d'information, traitement du langage naturel, acquisition des connaissances, etc.).
Il convient de considérer la possibilité d'utiliser un système multi-paradigme basé sur la théorie des GC. Différents prototypes de systèmes de manipulation des GC ont été développés mais aucun système multi-paradigme n'a été conçu. Nous nous sommes ainsi consacré dans cette thèse à *concevoir et à développer un système multi-paradigme utilisant la théorie des GC.*
Le système est composé :

1) d'un *environnement graphique* comprenant *un langage parallèle* et *un modèle de formation incrémentale d'une mémoire dynamique.*

   Le langage, appelé *Synergy est basé sur l'activation des GC et intègre différents modèles de programmation* (en l'occurrence, le modèle procédural, le modèle fonctionnel et le modèle orienté-objet). Une application en *Synergy* est organisée en une base de connaissances appelée "mémoire à long terme" et une zone de travail appelée "mémoire de travail", les deux sont composées de GC. Le modèle de formation porte sur la "mémoire à long terme" et spécifie comment de nouvelles connaissances y sont intégrées automatiquement.

2) d'*une extension conceptuelle, contextuelle et orientée objet du langage PROLOG*, appelée *Prolog+CG.*

   L'extension conceptuelle se base sur les GC et l'extension orientée objet sur les deux premières extensions ainsi que sur une formulation logique de la

Liste des figures

programmation orientée objet. *Prolog+CG* étend PROLOG sans le "masquer" ; un programme Prolog est aussi un programme Prolog+CG.

3) d'une *hiérarchie d'opérations sur les GC*, incorporée dans *Synergy* et *Prolog+CG*.

Motivé par un souci de synthèse, le but principal de cette thèse est de concevoir et de développer le système dans sa totalité, en essayant pour chaque composante d'intégrer plusieurs approches.

Les solutions que nous proposons peuvent être utilisées dans différents domaines. De telles utilisations pourraient susciter de nouveaux développements permettant de compléter et d'enrichir les composantes du système ainsi que leur intégration.

Mots clés : représentation et manipulation des connaissances, réseaux sémantiques, Graphes Conceptuels (GC), opérations sur les GC, langage logique pour les GC, activation de GC, langage parallèle multi-paradigme pour les GC, formation dynamique d'une mémoire basée sur la généralisation, formation dynamique d'une base de connaissances selon la théorie des GC.

Table des matières

Chapitre 1

Introduction

1.1 Motivation

Dans des domaines tels le génie logiciel ([Pressman, 92], [Shlaer et Mellor, 92]), l'intelligence artificielle ([Jackson, 90], [Mattos, 91]), les systèmes tutoriels intelligents ([Wenger, 87], [Psotka et al., 88]) et les systèmes multi-agents ([Demazeau et Müller, 90, 91], [Wooldridge et Jennings, 95a]) plusieurs types de connaissances sont formulées et traitées.
Considérons à titre d'exemple le domaine des systèmes tutoriels intelligents (STI), un STI utilise des connaissances relatives au domaine, à l'expert du domaine, à l'apprenant, à la pédagogie, à la communication et à l'interface Homme-Machine ([Wenger, 87], [Psotka et al., 88], [Tiberghien et Mandl, 90], [Frasson et al., 92], [Larkin et Chabay, 92], [Cunnigham et Hubbold, 92]). Le domaine peut concerner aussi bien l'enseignement d'un langage de programmation que la formation sur un appareil, ou encore l'évaluation d'un malade. Ces connaissances sont de différents types impliquant souvent l'utilisation de plusieurs formalismes. Parmi ceux utilisés en STI, citons les systèmes à base de règles, les systèmes à base de "frames", les systèmes orientés objets, différentes formes de réseaux sémantiques et, plus récemment, les systèmes multi-agents et les systèmes à base de cas. D'autres formalismes, concernant plus particulièrement les connaissances procédurales sont aussi utilisés, par exemple, les graphes de dépendances fonctionnelles, les graphes de tâches, les diagrammes de transition d'états, les graphes de flot de données et les réseaux de pétri.
Différents formalismes sont souvent utilisés dans un même STI rendant difficile l'intégration, la communication, l'utilisation et le partage des connaissances et des expertises. Cette remarque s'applique aussi aux autres domaines (en l'occurrence, génie logiciel, base de connaissances et systèmes multi-agents).

Un système multi-paradigme, basé sur un formalisme uniforme serait donc approprié pour ces domaines, où plusieurs types de connaissances sont représentées et traitées.
Pour répondre à ce besoin, nous avons considéré la théorie des Graphes Conceptuels (GC) proposée par John Sowa comme un *formalisme universel de représentation des connaissances* [Sowa, 92] ; la théorie est présentée non seulement comme un fondement

logique des réseaux sémantiques mais aussi comme une théorie du traitement de l'information chez l'humain et la machine [Sowa, 84]. La logique des GC est fondée sur l'algèbre relationnelle et la logique des graphes existentiels de Peirce [Robert, 73]. Sowa a montré également l'équivalence de la logique des GC avec la logique des prédicats [Sowa, 84] et une correspondance avec KIF (Knowledge Interchange Format) est en cours [Sowa, 93].

Depuis la parution de son ouvrage en 1984, une communauté de chercheurs à travers le monde s'est graduellement formée pour utiliser la théorie des GC comme un formalisme de représentation des connaissances dans des domaines aussi divers que les bases de données, les bases de connaissances, le traitement du langage naturel, l'acquisition des connaissances, le génie logiciel, les systèmes d'information, l'apprentissage, etc. ([Nagle et al., 92], [Pfeiffer and T. E. Nagle, 92], [Mineau et al., 93], [Tepfenhart et al., 94] et [Ellis et al., 95]).

Des ateliers annuels ont été organisés entre 1986 à 1992 et depuis 1993 une conférence internationale est organisée chaque année.

La théorie des GC pourrait donc constituer un formalisme pour un système multi-paradigme. Différents prototypes de systèmes de manipulation des GC ont été développés mais aucun système multi-paradigme n'a été conçu. Nous nous sommes ainsi consacré dans cette thèse à concevoir et à développer un système multi-paradigme utilisant la théorie des GC.

1.2 But de la thèse : Un système multi-paradigme utilisant la théorie des GC

La figure 1.1 décrit l'architecture générale du système que nous proposons. Ce dernier est composé :

→ d'un *environnement graphique* comprenant un *langage parallèle* et un *modèle de formation automatique et incrémentale d'une base de connaissances selon la théorie des GC*. Le langage, appelé *Synergy* est basé sur *l'activation des GC* et il *intègre différents modèles de programmation* (en l'occurrence, le modèle procédural, le modèle fonctionnel et le modèle orienté-objet). Dans Synergy, un GC peut représenter un *contexte de traitement* et son activation résulte de l'interprétation d'un concept comme une entité active ayant son propre cycle de vie et, aussi, de la capacité de certaines relations à propager l'activation dans le graphe.

Une application dans Synergy est organisée en une base de connaissances appelée "mémoire à long terme" et une zone de travail appelée "mémoire de travail", les deux sont composées de GC. Le modèle de formation d'une base de connaissances porte sur la "mémoire à long terme" et spécifie comment de nouvelles connaissances y sont intégrées automatiquement.

→ d'*une extension conceptuelle, contextuelle et orientée objet du langage PROLOG*, appelée *Prolog+CG*. L'extension conceptuelle se base sur les GC et l'extension orientée objet sur les deux premières extensions ainsi que sur une formulation logique de la programmation orientée objet. *Prolog+CG* étend PROLOG sans le "masquer" ; un programme Prolog est aussi un programme Prolog+CG.

→ d'une *hiérarchie d'opérations sur les GC*, incorporée dans Synergy et Prolog+CG.

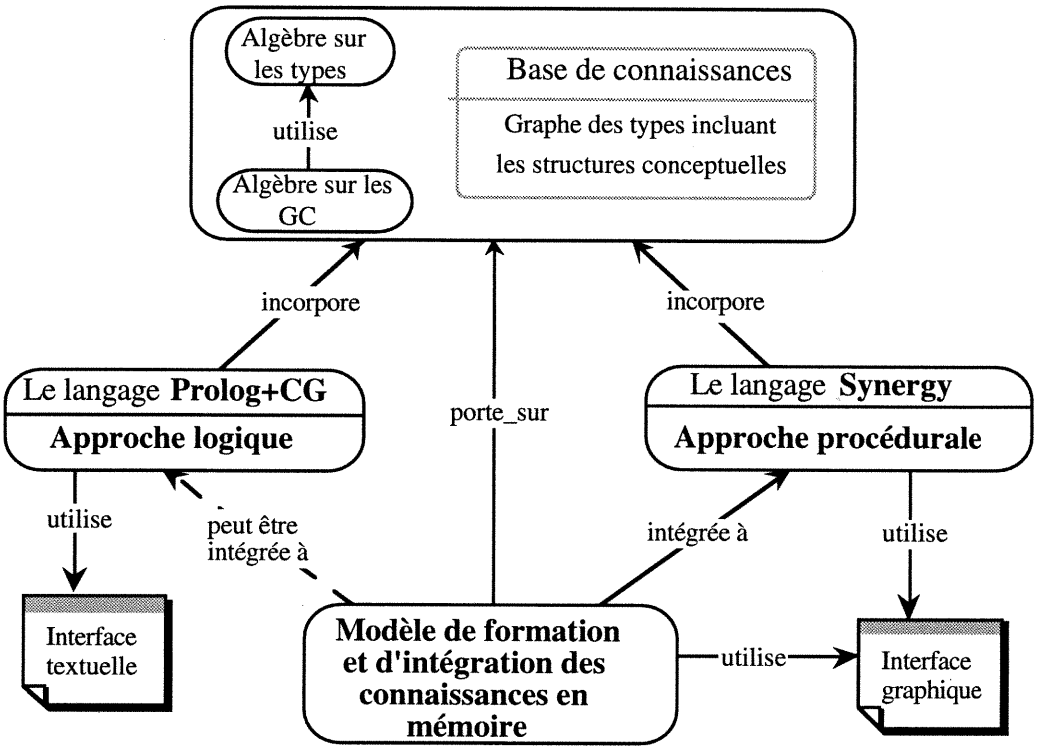


Figure 1.1 : Système multi-paradigme utilisant la théorie des GC

Prolog+CG et Synergy se basent sur deux approches complémentaires de conception et de développement de système de représentation et de manipulation des connaissances : approche logique à base de règles et approche "procédurale". Par ailleurs, Prolog+CG est un langage textuel et séquentiel alors que Synergy est un langage graphique et parallèle.
Les deux langages ont en commun : 1) l'utilisation et l'extension de la théorie des GC, 2) deux algèbres, une pour les types de concepts et l'autre pour les GC, 3) l'adoption d'une approche orientée objet et la conception d'une application (ou "programme") sous la forme d'une base de connaissances et d'un espace de travail permettant de formuler les requêtes. Pour les deux langages, la base de connaissances correspond au graphe de généralisation des types de concepts, enrichie par les structures conceptuelles associées aux types.

Notons que le système général (Figure 1.1) n'offre pas une "1" base de connaissances qui est exploitée par Synergy et/ou Prolog+CG; il offre plutôt deux langages qui permettent, chacun selon sa syntaxe et son interface, la formulation et l'exploitation d'une base de connaissances.
Actuellement, le modèle de formation de la mémoire est intégré à Synergy, mais il peut s'intégrer également au langage Prolog+CG. Enfin, une combinaison possible des deux langages serait de permettre des invocations de Prolog+CG à partir d'une application Synergy. Ces deux points font partie des travaux futurs !

1.3 Organisation de la thèse

Dans le chapitre 2, nous introduisons brièvement la théorie des GC, puis nous considérons des systèmes de manipulation des GC. Nous soulignons suite à cette revue, l'absence d'une hiérarchie des opérations sur les GC comme celle que nous proposons, l'absence d'un "Prolog étendu pour les GC", l'absence d'un langage parallèle multi-paradigme basé sur l'activation de GC et enfin, l'absence d'un modèle de formation incrémentale d'une base de connaissances selon la théorie des GC. Nous considérons ensuite le contexte "général" de notre étude ; des travaux qui ne sont pas effectués dans le cadre de la théorie des GC mais qui ont des éléments en commun avec notre thèse. Nous présentons ainsi des travaux sur les opérations de manipulation des réseaux sémantiques (RS) et les langages logiques pour les RS, puis des travaux sur l'activation de différents types de graphes; RS, "frames", graphes de flot de données, réseaux de Pétri, diagrammes de transition d'états et objets actifs. Enfin, nous considérons des travaux sur la formation incrémentale d'une hiérarchie de concepts ainsi que le modèle de mémoire dynamique basée sur la généralisation.

Les chapitres 3 à 5 portent sur le langage Synergy. Le chapitre 3 présente les éléments de base du langage, en particulier l'environnement conceptuel (composé d'une mémoire à long terme qui constitue la base de connaissances et d'une mémoire de travail), les "GC de Synergy", les mécanismes d'abstraction du langage, le mécanisme d'activation des GC, la boucle principale de l'interpréteur, la "syntaxe" et l'interface graphique du langage.
Les chapitres 4 et 5 montrent comment Synergy peut être utilisé pour des programmations fonctionnelle, procédurale, par accès, par événement, par processus concurrents, par objets actifs concurrents et par agents.
Le chapitre 6 présente le processus de formation incrémentale de la mémoire à long terme ainsi que son interface graphique. Nous y précisons tout d'abord le type de connaissances à intégrer en mémoire et nous décrivons ensuite comment l'intégration est effectuée automatiquement.

Le chapitre 7 introduit le langage Prolog+CG, après la définition des éléments de base du langage et en particulier la notion d'objet en Prolog+CG, nous présentons la syntaxe du langage suivie d'exemples, les notions d'héritage et d'instanciation sont introduites par la suite. Enfin, une brève description de l'implantation de Prolog+CG complète la présentation de ce dernier.

Le chapitre 8 discute des applications possibles, de l'originalité, des limites et des développements futurs de chacune des composantes de notre système.

Nous concluons la thèse en établissant la méthodologie que nous avons adopté, les points forts et les points faibles de notre système ainsi que les axes de recherche qu'il peut susciter.

Chapitre 2

Systèmes de manipulation des GC et autres travaux connexes

La première partie du chapitre présente le contexte "immédiat" pour notre thèse : la théorie des Graphes Conceptuels (GC) et les systèmes de manipulation des GC, ces derniers sont considérés selon les axes suivants : opérations sur les GC, langages logiques pour les GC, traitement des connaissances procédurales et activation de GC, et modèles de formation dynamique d'une base de connaissances selon la théorie des GC. La seconde partie du chapitre présente le contexte "général"; des travaux qui ne sont pas effectués dans le cadre de la théorie des GC mais qui ont des éléments en commun avec notre thèse. Nous présentons ainsi des travaux sur les opérations de manipulation des réseaux sémantiques (RS) et les langages logiques pour les RS, puis des travaux sur l'activation de différents types de graphes; RS, "frames", graphes de flot de données, réseaux de Pétri, diagrammes de transition d'états et objets actifs. Enfin, nous considérons des travaux sur la formation incrémentale d'une hiérarchie de concepts ainsi que le modèle de la mémoire dynamique basée sur la généralisation.

2.1 Introduction

Différentes formes de *Réseaux Sémantiques* (RS) ont été développées selon des besoins, motivations et objectifs divers ([Minsky, 68], [Schank et Colby, 73], [Norman et al., 75], [Findler, 79], [Brachman et Levesque, 85], [Lehmann, 92a,b], [Sowa, 92b]), ce qui a motivé plusieurs chercheurs à élaborer une formalisation des RS par correspondance avec : la logique des prédicats ([Hendrix, 79], [Schubert et al., 79], [Shapiro, 79], [McSkimin et Minker, 79], [Simmons et Chester, 77]), l'algèbre relationnelle ([Mylopoulos et al., 75], [Sowa, 76, 84]) et/ou la logique des graphes existentiels de Peirce [Sowa, 84, 95].

Les RS ont inspiré par ailleurs des travaux sur les algèbres des structures typées avec attributs et des extensions conceptuelles de Prolog ([Aït-Kaci, 84], [Aït-Kaci et al., 94], [Carpenter, 92]), la section 2.4 présente brièvement le travail du groupe de Aït-Kaci.

Les RS ont inspiré également les systèmes à base de "frames" ([Jackson, 90], [Mattos, 91], [Lehmann, 92a]) et les systèmes hybrides de la famille KL-ONE [Woods et Schmolze, 92] qui intègrent la programmation par classification [McGregor, 91] et la programmation en logique. Par ailleurs, les modèles de formation de la mémoire considère celle-ci comme un "immense" RS ([Rumelhart et al., 72], [Schank, 82], [Waltz, 90]). La section 2.6 considère cet aspect.

Les réseaux sémantiques sont généralement utilisés pour représenter des connaissances déclaratives, des chercheurs les ont toutefois utilisés pour représenter également des connaissances procédurales ([Norman et al., 75], [Rieger, 76], [Rieger et Grinberg, 77], [Schank et Abelson, 77], [Cercone et al., 92], [Mylopoulos, 92], [Shapiro et Rapaport, 92], [Sowa, 93], [Skuce, Lethbridge, 95]). En plus de représenter les connaissances procédurales avec des RS, le groupe de Norman, Rumelhart et al. [Norman et al., 75] considère également leur interprétation (évaluation).

La section 2.5.1 présente brièvement cette première tentative *d'activation conceptuelle des réseaux sémantiques*. Dans les sections 2.2.4 et 2.3.3 nous considérons le cas des Graphes Conceptuels (GC) et les formes d'activation des GC proposées dans la littérature.

D'autres formes d'activation des RS ont été proposées dans la littérature, par exemple :

→ La *propagation de marqueurs* ([Fahlman, 88], [Hendler, 92]) : des nœuds du réseau sont marqués par des marqueurs au départ et ensuite, pour trouver des connexions entre les nœuds, les marqueurs sont propagés sous certaines conditions à des nœuds voisins et ainsi de suite. On peut ainsi trouver les chemins qui peuvent connecter différents nœuds du réseaux.

La propagation de marqueurs dans un réseau sémantique a été utilisée pour traiter l'héritage et pour considérer certains aspects du traitement du langage naturel ou de la résolution de problèmes [Hendler, 92]. Notons qu'avec la propagation de marqueurs, les nœuds (concepts) ne sont pas évalués (ils n'incorporent pas un traitement) et ne produisent pas de résultat ni de changement dans le réseau.

→ *Propagation de données* [Bic, 85, 88] : Bic considère un réseau sémantique comme un graphe de flot de données, les concepts sont actifs (des acteurs), capables de recevoir, de traiter et d'émettre des messages (données), via certains liens du réseau. Par exemple, pour vérifier si une information, comme "a -r1-> b -r2-> c", existe dans le réseau, il "suffit" de l'envoyer comme message au nœud dans le réseau qui représente "a". Ce dernier va ensuite propager le message "b -r2-> c" via tout lien "a -r1-> X " tel que X et b puissent s'unifier. X peut ensuite transformer le message et envoyer le résultat à d'autres nœuds et ainsi de suite.

→ *Réseaux sémantiques connexionistes* [Shastri, 92] : les nœuds effectuent un traitement mais de type "connexioniste", comme dans les "réseaux connexionistes" (neural nets), les liens possèdent généralement des poids qui influencent l'activité des nœuds.

L'activation des réseaux sémantiques n'est elle même qu'une forme d'activation de graphes. La Figure 2.1 montre une hiérarchie de formalismes orientés activation de graphes (la hiérarchie n'est pas exhaustive). Notons que chaque forme de graphe, par exemple les réseaux de pétri ou les réseaux connexionistes, constitue une "famille" où différentes approches et extensions ont été proposées.

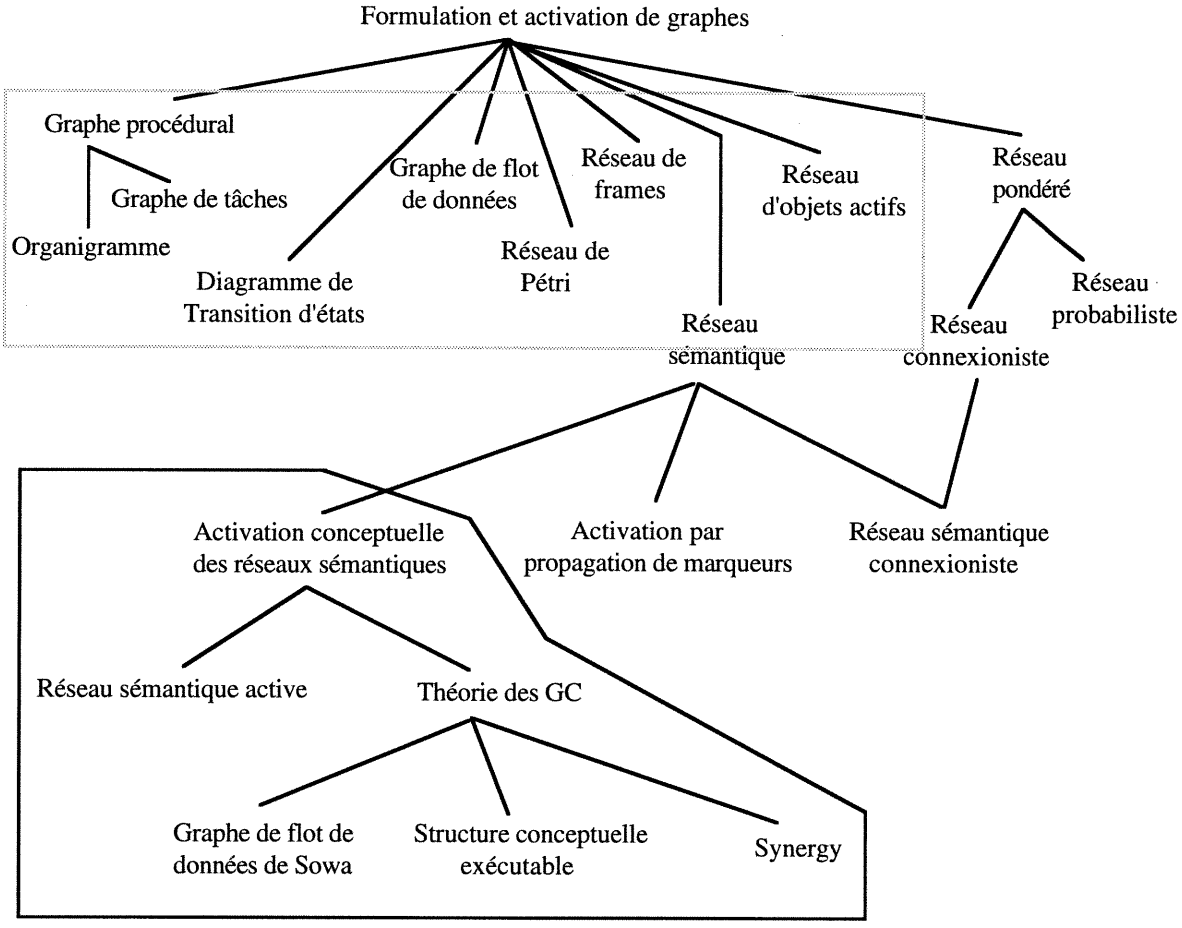


Figure 2.1 : Hiérarchie de formalismes orientés activation de graphes

Notre intérêt porte plus particulièrement sur l'activation conceptuelle des GC et des RS (surface encadrée par un trait continu dans la Figure 2.1). A cause toutefois du caractère multi-paradigme de notre approche, nous considérons aussi dans ce chapitre l'activation d'autres formes de graphes, en particulier les réseaux de frames, les graphes de flot de

données, les réseaux de Pétri, les diagrammes de transition d'états et les objets actifs (surface encadrée par un trait en pointillé dans la Figure 2.1 et traitée dans la section 2.5).

2.2 Théorie des Graphes Conceptuels (GC)

Un Graphe Conceptuel (GC) est un ensemble de concepts connectés par des relations conceptuelles n-adiques. Un concept est une spécification typée d'un référent : [TYPE: Referent] et un référent peut être une instance particulière, une variable, un ensemble ou un GC. Si un référent est une variable, on peut l'omettre et spécifier uniquement le type du concept.
Un ordre partiel de généralisation est défini sur l'ensemble des types des concepts. Dans la théorie des GC, les types forment un treillis. La figure 2.2 montre un exemple de GC qui représente la proposition "la femme Valerie aime l'homme Andre qui est un ami de Jo et Valerie est mariée à Jo".

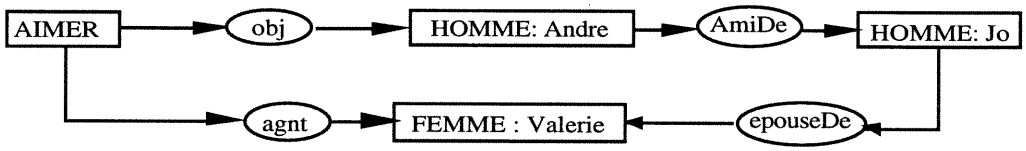


Figure 2.2 : Exemple de GC

2.2.1 GC et algèbre relationnelle

Une correspondance peut être établie entre une description intentionnelle d'une base de données et les GC [Sowa, 76, 84]. L'information intentionnelle dans une base de données concerne essentiellement le rôle des domaines dans une relation, les contraintes sur les valeurs d'un domaine et les dépendances fonctionnelles [Sowa, 76]. Un GC est composé de concepts (qui correspondent aux domaines des valeurs), de relations (qui correspondent aux rôles des domaines) et éventuellement de liens de dépendances fonctionnelles. La figure 2.3 montre un exemple d'une telle correspondance (adapté de [Sowa, 76]).

ENGAGER	EMPLOYE	GERANT	DATE
	Robert Lee	Jo Bart	3/1/74
	David Sowl	Jo Bart	3/1/72

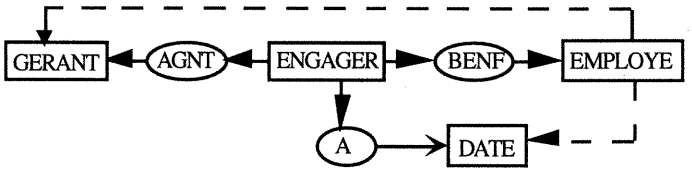


Figure 2.3 : Correspondance entre l'intention d'une relation et un GC

Un GC peut représenter aussi un schéma de la base de données, la sémantique d'une question et le graphe qui se forme graduellement lors de la recherche d'une réponse.

Sowa a défini des *règles de formation* qui permettent de dériver un nouveau GC à partir des GC existants. Il a souligné par ailleurs le rapport entre ces règles et celles de l'algèbre relationnelle [Sowa, 76, 84] :

→ *dérivation par copie d'un GC .*

→ *dérivation par spécialisation ou restriction d'un concept :*

obtenir un nouveau GC en remplacant le type d'un concept d'un GC par un sous-type et/ou en remplacant le référent générique d'un concept par un référent particulier.

Cette règle correspond à la restriction (ou sélection) en base de données où une relation est dérivée d'une autre en spécifiant une contrainte sur les valeurs d'un attribut de la relation.

→ *dérivation par jointure de concepts :*

deux concepts identiques c1 et c2 dans un même GC ou dans deux GC peuvent être joints pour former un nouveau GC en éliminant le concept c1 et en attachant ses relations au concept c2.

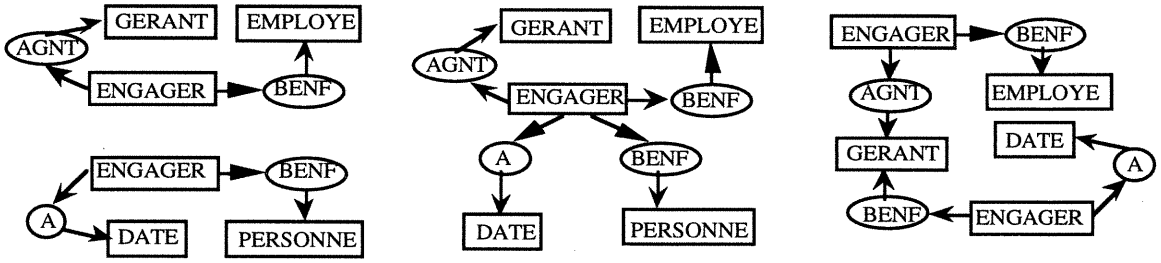
La dérivation par jointure correspond à la jointure de deux relations en base de données.

→ *dérivation par simplification :*

si les mêmes concepts sont reliés par deux relations identiques, alors l'une d'elles peut être éliminée.

La figure 2.4 montre un exemple de dérivation par restriction et jointure : les concepts [ENGAGER] dans les deux graphes sont restreints afin qu'ils aient un même type et un même référent. On y applique ensuite la jointure (figure 2.4b).

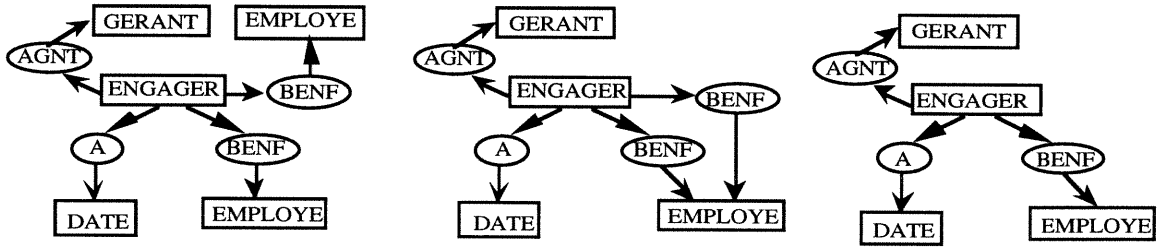
Deux GC peuvent se joindre de différentes façons; la Figure 2.4c montre une autre dérivation possible pour le même exemple : identification des concepts [GERANT] et [PERSONNE] (restriction des types suivie d'une restriction des référents) et ensuite jointure autour du résultat.



(a) deux GC (b) jointure autour de [ENGAGER] (c) jointure autour de [GERANT]

Figure 2.4 : Exemples de restriction et de jointure de concepts.

Si on considère maintenant le GC de la Figure 2.4b, on peut restreindre [EMPLOYE] et [PERSONNE] afin qu'ils deviennent identiques (Figure 2.5a) et y appliquer ensuite la jointure autour du résultat (Figure 2.5b), la relation BENF (bénificiaire) est dupliquée et l'une des deux est donc éliminée (Figure 2.5c). Notons qu'au lieu de rendre [PERSONNE] identique à [EMPLOYE], on aurait pu le rendre identique à [GERANT].



a: restriction d'un concept b: jointure de deux concepts d'un GC c: simplification

Figure 2.5 : Exemple d'une succession de dérivations.

Sowa [Sowa 76, 84] a défini en plus deux règles de dérivation qui correspondent à des applications successives des règles ci-dessus :

→ *projection d'un graphe g2 sur un graphe g3* :

elle correspond à la règle de restriction/spécialisation, appliquée toutefois aux graphes et non à un concept. Si le graphe g3 est plus spécifique que g2 alors g1, appelé la *projection* de g2 sur g3, est un sous-graphe de g3 qui correspond à une copie de g2 mais avec restriction de certains de ses concepts.

Cette règle correspond à la projection et/ou la restriction en algèbre relationnelle. En effet, la projection d'une relation consiste à ne retenir que certains attributs de la relation. La relation qui en résulte a pour schéma un sous-schéma de la relation initiale. On peut appliquer ensuite des restrictions sur des attributs de la relation obtenue.

→ *jointure de deux graphes g1 et g2* :

alors que la règle de jointure de concepts joint deux graphes à partir d'un concept identique, la présente règle joint deux graphes à partir d'un sous-graphe identique. Elle se base pour cela sur la projection.

La jointure des graphes correspond à la jointure des relations sur plusieurs attributs communs.

2.2.2 GC et graphes de flot de données

L'exemple de la Figure 2.3 présente une première formulation des dépendances fonctionnelles dans les GC [Sowa, 76]. Dans son ouvrage [Sowa, 84], Sowa propose une

formulation plus explicite, basée sur une forme particulière de graphe de flot de données (appelons la GFDS) qui peut être fusionné à un GC produisant un graphe "hybride".

Un GFDS est un graphe de dépendance fonctionnelle, composé de concepts (qui représentent les données) et d'acteurs (qui représentent les fonctions). Un acteur peut s'activer uniquement si tous ses arguments en entrée (qui sont des concepts) ont des valeurs. Contrairement aux acteurs dans les graphes de flot de données, un acteur dans un GFDS ne consomme pas les valeurs de ses arguments en entrée et il est bloqué si un de ses résultats est différent de la valeur de l'argument correspondant, ou si le résultat est non conforme au type de l'argument. Les GFDS se basent sur l'assignation unique (single-assignment) : une fois déterminée, la valeur d'un concept ne peut pas changer. L'activation d'un GFDS peut être dirigée par la "disponibilité" des données en entrée et/ou par la demande des résultats en sortie des acteurs.

Le flot de données dans un GFDS peut être contrôlé par les types de concepts et les contraintes sur les types, par exemple : dans <A> --> [Int >0] --> <B>, l'acteur A ne peut affecter une valeur au concept [Int >0] que si la valeur est supérieure à 0.

Le contrôle de l'activation demeure néanmoins problématique dans certaines cas; comment formuler par exemple la condition suivante (sans ajouter ou modifier la formulation initiale) : "**if** a < b **then** c := f(a) **else** c := g(b) " ?

2.2.3 GC et structures conceptuelles

Différentes structures conceptuelles ont été proposées dans la littérature sur les réseaux sémantiques : la hiérarchie des types, la définition d'un type, la description d'une instance, la description d'un schéma, la description d'un canon, etc. ([Norman et al., 75], [Findler, 79], [Sowa, 84]). Par ailleurs, en logique des prédicats, l'opération d'abstraction est utilisée pour formuler des "Lambda-expressions" dont le corps correspond à un prédicat ou à une formule logique. Dans sa logique des GC, Sowa utilise l'abstraction pour la même raison, avec les GC comme mode de représentation (au lieu des prédicats). L'abstraction est utilisée pour spécifier les différentes structures conceptuelles précitées ou d'autres structures comme, par exemple, la définition d'une relation.

Exemple :

→ Définition d'un type de concept :

type PARISIEN(X) **est** [PERSONNE: X] <-(obj)- [HABITER] -(lieu)-> [VILLE : Paris].

→ Spécification d'un schéma pour le type PARISIEN :

schéma PARISIEN(X) **est**

[PARISIEN: X]<-(agent)- [AIMER]-(obj)->[CULTURE] -(attr)->[STYLE: "HG"].

→ Définition du type d'une relation :

relation fournisseur(X, Y) **est**

[SOCIETE: Y] <-(agent)-[FOURNIR] -(obj)-> [MARCHANDISE: X].

2.2.4 GC, logique des prédicats et logique de Peirce

Tout GC peut être traduit en une conjonction de prédicats ; chaque concept est représenté par un prédicat monadique TYPE(Referent) et chaque relation n-adique est représentée par un prédicat n-adique [Sowa, 84]. Pour les règles d'inférences, Sowa adopte la logique des graphes existentiels de Peirce [Robert, 73] qui est équivalente à la logique des prédicats.
Dans un second ouvrage à paraître, Sowa présente une formulation plus approfondie de sa théorie et il précise davantage le lien avec la logique des prédicats et en particulier avec KIF, ainsi qu'avec la logique de Peirce.

2.3 Systèmes de manipulation des GC

Cette section présente des systèmes de manipulation de GC selon les axes suivants : opérations sur les GC, langage logique pour les GC, traitement de connaissances procédurales et activation des GC et, modèles de formation dynamique d'une base de connaissances selon la théorie des GC.

2.3.1 Opérations sur les GC

Sowa [Sowa, 84] a proposé une définition axiomatique des opérations (essentiellement la projection et la jointure maximale), basée sur la dérivation par les règles de formation. Parallèlement à cette définition formelle, il a formulé un "algorithme" pour la jointure maximale :

"one algorithm for computing a maximal join is to start by joining two graphs on a single concept. Then extend the join by one conceptual relation at a time by looking for potential candidates along the boundaries of the part that has already been joined."

[Sowa, 84 p. 102],

une autre formulation de cet "algorithme" est proposée dans [Sowa et Way, 86 p. 66] :

"A maximal join is a sequence of joins and simplifications applied to the matching nodes of two graphs. Once the starting place for the maximal join is determined, a simple join is performed on the matching concepts. Next, all the nodes adjacent to the joined concept are checked to see if any of the relations from one graph match those of the other. If a match is found, then the procedure continues around the graph locating matching concepts, restricting their types, joining and simplifying until no further matches are detected.".

Les deux versions adoptent donc, implicitement, le cadre général de l'opération d'appariement et ce, afin de *guider* l'application des règles de formation.
Par ailleurs, les opérations définies par Sowa (en particulier la projection et la jointure maximale) sont caractérisées comme étant NP-Complet ([Mugnier et Chein, 92, 93], [Cogis et Guinaldo, 95]). Un compromis entre la généralité des GC et l'efficacité des opérations est à rechercher. Dans [Kabbaj, 87], nous avons défini et utilisé les "GC fonctionnels" (un concept dans un GC fonctionnel ne peut avoir des relations en sortie de même type et il en est de même pour les relations en entrée) afin d'avoir des opérations polynomiales. Il en est de même pour Liquière et Brissac [Liquière et Brissac, 94]. L'importance et l'impact du caractère fonctionnel sur les opérations a été souligné également par Mineau [Mineau, 94] et Fall [Fall, 95]. Pour avoir des opérations polynomiales, Mugnier et Chein [Mugnier et Chein, 92] ont considéré le cas où les GC sont des arbres.
Par ailleurs, parmi les autres restrictions communement adoptées par les chercheurs, citons : le référent d'un concept est soit une instance soit une variable avec quantificateur existentiel par défaut, les relations sont dy-adiques et la hiérarchie des types des relations n'est pas considérée.

Dans la communauté des GC, plusieurs groupes de chercheurs ont été/sont concernés par les opérations sur les GC. Deux directions complémentaires co-existent : étude théorique des opérations et en particulier la projection ([Mugnier et Chein, 92, 93], [Cogis et Guinaldo, 95]) et étude procédurale en vue d'une implantation ([Fargues et al., 86], [Garner et Tsui, 86], [Kabbaj, 87] et [Rao and Foo, 87] pour ne citer que les premiers systèmes). Ces systèmes diffèrent par la définition des GC et des opérations qui les manipulent et aussi par le nombre des opérations offertes.
Dans le cadre de notre maîtrise ([Kabbaj, 87], [Moulin et Kabbaj, 90]), nous avons réalisé un système de manipulation des GC qui comprend les opérations jointure maximale, EstPlusGeneral et généralisation, les trois définies comme des spécialisations de l'opération d'appariement. Les opérations recoivent en entrée des GC fonctionnels, simples ou composés avec co-référents, ainsi que les points d'entrée associés aux GC. Les co-référents sont utilisés pour déterminer les points d'entrée. D'autres opérations sont offertes par le système : expansion de concepts et de relations et contraction de définition de concept ou de relation.
Fargues et al. [Fargues et al., 86] ont réalisé la jointure, la généralisation, la projection et l'unification pour des GC simples, Garner et al. [Garner et Tsui, 86] ont réalisé la jointure, la projection et l'unification pour des GC simples.
Depuis 1987, d'autres systèmes ont été développés (en particulier [Kocura and Kwong, 92]). Le système du groupe de Kocura offre les règles de formation de Sowa et la contraction.

Recemment, un groupe de chercheurs a proposé le système CGKEE [Munday et al., 1995], composé d'une interface, d'un éditeur, d'un processeur, d'une mémoire de travail et d'une base de connaissances. Les opérations offertes par le processeur, le module principal de CGKEE, sont les règles de formation définies sur des GC simples.

Peu de chercheurs ont proposé une définition des opérations sur les GC composés. A part notre système [Kabbaj, 87] et la définition actuelle des opérations (annexe 1), Chan et al. [Chan et al., 88] ont proposé une définition récursive de l'unification afin de considérer des GC composés et Leishman [Leishman, 92] a proposé également une définition de la généralisation qui porte sur des GC composés. Recemment, Esch et Levinson [Esch et Levinson, 95] ont proposé d'étendre la définition des règles de formation afin de considérer les GC composés.
La majorité des autres chercheurs ont considéré uniquement des GC simples ([Fargues et al., 86], [Sowa et Way, 86], [Myaeng et Lopez-Lopez, 91], [Mugnier et Chein, 92], [Ellis, 92], [Mineau, 92], [Yang et al., 92], [Liquière et Brissac, 94], [Fall, 95], [Ghosh et Wuwongse, 95], [Willems, 95], [Munday et al., 95]).
Aussi, différentes opérations ont été proposées par plusieurs chercheurs (appariement, projection, différentes formes de jointure, unification, généralisation, analogie, contraction, canonicité, etc). Il n'est souvent pas clair le rapport et le lien entre ces différentes opérations. Pour surmonter cette lacune, nous définissons une hiérarchie des opérations sur les GC dont la racine est l'opération d'appariement (les autres opérations sont des spécialisations de l'appariement). La hiérarchie fait ressortir ce qui est commun et ce qui caractérise les diverses opérations. Une autre lacune que nous tentons de surmonter dans notre définition des opérations est de considérer les GC simples aussi bien que les GC composés avec les liens de co-références.
Enfin et au lieu d'offrir un paquetage d'opérations sur les GC "hors contexte", il est plus approprié de les fournir dans le cadre d'un langage de programmation afin de pouvoir formuler des traitements plus complexes qui utiliseraient les opérations comme des primitives. Le système du groupe de Kocura [Kocura and Kwong, 92] offre un langage pour formuler des "scripts" (procédures) composés essentiellement de leurs opérations sur les GC. La puissance d'expression du langage proposé est toutefois limitée. Pour répondre au besoin d'avoir les opérations comme primitives dans un langage, nous avons incorporé notre hiérarchie des opérations sur les GC dans les langages Synergy et Prolog+CG.

2.3.2 Langages logique pour les GC

Sowa a utilisé la logique de Peirce comme base pour son système d'inférence pour les GC [Sowa, 84]. Des chercheurs tentent de réaliser un tel système ([Heaton et Kocura, 93], [Emond, 93]) alors que d'autres travaillent sur des formes plus "conventionnelles", des règles à la Prolog. En effet, plusieurs projets comprennent une composante "inférence pour les GC" (par exemple [Fargues et al., 86], [Garner et Tsui, 88], [Garner et al., 92], [Rao et Foo, 87]) et utilisent une(des) méthode(s) d'inférence et des règles de la forme :

GC1 -> GC2 Et ... Et GCn.

Aussi, un des modules du projet PEIRCE est *prévu* pour un langage "CG with constraints" (CGC) [Ellis, 92]. Un programme CGC est une collection de règles de la forme indiquée ci-dessus. Un programme CGC est donc similaire à un programme Prolog, sauf qu'au lieu des prédicats, les buts sont représentés par des GC et l'unification des prédicats est donc remplacée par l'unification des GC. Dans une série d'articles, Ghosh et Wuwongse [Ghosh et Wuwongse, 93, 94, 95] ont proposé une sémantique formelle pour un tel langage mais aucune réalisation n'a toutefois été effectuée. En supposant qu'ils réalisent un tel langage (appelons le CGC), ce dernier ne répondrait pas aux besoins suivants (besoins qui concernent l'organisation et l'efficacité "pratique" du langage) :

→ Besoin d'indexation des règles : les règles dans CGC ne sont pas partitionnées contrairement même au Prolog "standard" où les règles sont indexées selon le nom et l'arité du prédicat-tête d'une règle. Pour satisfaire un but dans CGC, il faudrait parcourir toutes les règles du programme. Un tel parcours peut être envisagable si on considère uniquement quelques règles (comme c'est le cas dans le projet KALIPSOS [Fargues et al., 86]) mais un grand nombre de règles rendrait un tel parcours trop lourd.
Des chercheurs en programmation logique ont proposé une extension contextuelle de Prolog afin d'obtenir des programmes plus modulaires ([Kauffmann et Grumbach, 86], [Monteiro et Porto, 89], [Dichev, 93]). D'autres chercheurs ont proposé une intégration de la programmation logique et de la programmation orientée objet ([Fukunaga and Hirose, 86], [McCabe, 92], [Moss, 93], [Davison, 93]). CGC ne tient pas compte de ces extensions.

→ CGC est présenté comme un "PrologCG pur" qui permet (uniquement) de déduire la véracité d'une proposition, formulée en GC, à partir d'une collection de règles et de faits. Tout le côté procédural (comme l'appel à des primitives) est ignoré. Aussi, aucune algèbre pour les GC n'est offerte par CGC (du moins, dans sa conception actuelle).

→ En complément au point précédent, CGC n'est pas une extension du langage Prolog (comme C++ l'est pour C) ; un programme Prolog n'est pas un programme CGC.

Un programme en CGC n'inclut pas de règles en Prolog et une règle en CGC ne peut contenir un but qui soit satisfait par des règles Prolog.
L'ouverture de CGC au Prolog standard serait appropriée afin d'utiliser et intégrer ce qui existe déjà et aussi afin de ne pas restreindre le programmeur à tout écrire en GC, certains traitements pouvant être formulés en Prolog.

→ Les buts qui composent une règle CGC sont représentés par des GC simples, ce qui limite la puissance d'expression du langage.

Nous voulons concevoir et développer un langage "Prolog pour les GC" qui puisse surmonter les limitations ci-dessus.

2.3.3 Traitement des connaissances procédurales et activation de GC

→ Les graphes de flot de données de Sowa (GFDS) ont été utilisés par plusieurs chercheurs de la communauté des GC ([Hines et al., 90], [Hines et Hines, 90], [Wuwongse et Ghosh, 92], [Delugach, 92], [Whipple, 93]). Dans leur formulation de la programmation orientée-objet en termes de GC, Hines et al. ainsi que Wuwongse et Ghosh proposent l'utilisation des GFDS comme formalisme pour les méthodes des classes/objets.
Pour son système de résolution de problèmes, le groupe de Pfeiffer et Hartley ([Coombs et Hartley, 87], [Fields et al., 91], [Pfeiffer et Hartley, 92]) a développé un environnement de programmation, appelé "Conceptual Programming , CP" qui se base sur les GC et les GFDS (avec quelques variantes comme la possibilité de relier des relations aux acteurs). Les acteurs sont utilisés durant la jointure pour tester la conformité du(des) graphe(s) résultat(s) et pour représenter des contraintes spatio-temporelles. Les acteurs dans CP sont uniquement des primitifs (ils ne peuvent être définis en termes de GFDS) et peuvent correspondre à des tests ou à des fonctions.

→ Des chercheurs ont utilisé les GC dans le cadre des méthodes d'analyse et de génie logiciel ([Feller et Rucker, 92], [Delugach, 91, 92], [Delugach et Hinke, 92]). En particulier, Delugach a proposé une extension de la notion d'acteur, appelée "démon". Il a introduit cette notion pour formuler et manipuler en GC des connaissances temporelles, en particulier pour ajouter/détruire des connaissances au cours d'un traitement, comme on peut le faire avec le "assert/retract" en Prolog.
Un "démon" consomme ses concepts en entrée et crée des concepts en sortie. Différentes questions se posent toutefois concernant cette notion de "démon" : en particulier quel serait le traitement des liens connectés aux concepts à détruire ?, quelle est la signification du concept qui sera créé alors qu'il est déjà lié par des relations à d'autres concepts du GC ?, etc.

Plutot que de créer/détruire des concepts, Synergy offre la possibilité de créer/détruire les valeurs des concepts (les concepts ne sont pas détruits). Nous avons ainsi les avantages de la notion de démon, sans ses inconvénients.

→ *Structure conceptuelle exécutable*

Pour leur système de résolution de problèmes, Garner, Lukose et al. ([Garner et al., 92], [Lukose, 92, 93], [Lukose et al., 95]) ont proposé la notion de "structure conceptuelle exécutable" qui peut être un "graphe d'acteur" (actor graph) ou un (problem map). Le premier type représente une action alors que le second représente un plan. Un graphe d'acteur est composé d'un GC et d'un acteur, le premier définit la "sémantique" de l'action (en spécifiant les propriétés essentielles de l'action) alors que le second décrit le "script" qui peut la réaliser. La sémantique d'une action (le GC et l'acteur qui la réalise) est celle proposée dans le système de résolution de problèmes STRIPS [Fikes et Nilsson, 71] : un acteur possède une *mémoire à court terme* utilisée pour enregistrer les valeurs actuelles associées au domaine et une *mémoire à long terme* qui contient les listes de préconditions, postconditions et à-éliminer (delete list); les trois correspondent à des listes de GC. Enfin, un acteur possède une *méthode principale* qui représente l'ensemble des méthodes auxquelles l'acteur peut répondre. Cette implantation "objet" d'un acteur est formulée comme une structure Prolog.
L'exécution d'un graphe d'acteur est initiée par un envoi de message à son acteur, le message est alors traité par sa méthode principale : elle teste si les pré-conditions sont vérifiées, si oui le corps de la méthode est exécuté et les post-conditions sont ensuite produites. Une séquence de graphes d'acteurs peut former un "problem map". Voici un exemple fourni par Lukose [Lukose, 93] :

```
[plan : [node : [obtain]->(obj)->[job_names]
                ->(srce)->[job_database]
                ->(rcpt)->[system]
                ->(agnt)->[system]
        ] ->(follow_by)->
        [node :  [display]->(obj)->[job_names]
                 ->(dest)->[stdout]
                 ->(agnt)->[system]
        ]
].
```

Un nœud peut être lui même une séquence d'actions.

→ Des tentatives ont été effectuées pour *décrire* des tâches/processus en termes de GC ([Cyre, 91], [Yen et Lee, 91], [Feller et Rucker, 92], [Sowa, 93]). Pour exécuter la description, Feller et Rucker utilisent les Réseaux de Pétri. Sowa propose soit de développer des axiomes et des règles d'inférences qui pourraient simuler le traitement (on demeure ainsi

dans un cadre logique et formel), soit de "compiler" la description dans un langage exécutable (comme KIF, Lisp, Pascal, C++, ...).

→ Dans le cadre de la programmation concurrente orientée-objet, les GC sont une option appropriée pour une représentation plus sémantique du contenu d'un message. C'est ce qui a été proposé par différents chercheurs ([Nagle, 89], [Haemmerlé, 95], [Pfeiffer et Waltar, 95]).

Aucune des approches proposées dans la littérature n'a utilisé les GC comme formalisme de base pour un langage parallèle multi-paradigme; intégrant les modèles procédural, fonctionnel et orienté objet.

2.3.4 Modèles de formation dynamique d'une base de connaissances selon la théorie des GC

→ Dans le cadre de la théorie des GC, Levinson ([Levinson, 84], [Levinson, 92]) et Ellis ([Ellis, 92], [Ellis, 93]) proposent des algorithmes de classification, similaires à (mais plus efficaces que) ceux développés dans la famille KL-ONE ([Woods, 79, 91], [Schmolze et Lipkis, 83], [Woods and Schmolze, 92]). Le but de la classification est de trier un ensemble de descriptions, selon l'opération "EstPlusGeneral" (subsume).
La classification d'un GC T (ou de la définition d'un nouveau terme T pour la famille KL-ONE) consiste généralement à trouver les "pères" et les "fils" de T, la place de T dans le graphe de généralisation est ainsi déterminée; il suffit alors d'ajouter T au graphe en le connectant à ses pères et à ses fils. Pour déterminer les pères de T, l'algorithme de classification commence par la racine du graphe de généralisation (ou la taxonomie des termes pour la famille KL-ONE) et "descend" ensuite dans le graphe à la recherche des pères de T. La descente se fait selon une stratégie de parcours (par exemple en profondeur, en largeur ou "topologique" [Ellis, 93]). Lors de la descente, l'algorithme teste si la description du nœud courant est plus générale que celle de T, si oui, T continue sa descente vers les fils du nœud courant. L'algorithme s'arrête lorsqu'aucun des fils du nœud courant n'est plus général que T, le nœud courant constitue alors un père pour T. Notons que si un des fils du nœud courant possède uniquement des éléments en communs avec T, le processus de classification ne fait pas ressortir cette similitude; il n'effectue pas de généralisation.
Pour la recherche des fils de T, l'algorithme de classification procède généralement comme suit : à partir de tout père de T, l'algorithme poursuit la descente, dès qu'il rencontre un terme Tf qui est plus spécifique que T et qu'aucun autre terme T' n'est plus spécifique que T et en même temps plus général que Tf, il conclut que Tf est un fils de T.

→ Godin et al. ([Godin et al., 86], [Mineau, 90], [Godin et al., 95]) proposent un algorithme de "catégorisation conceptuelle" (conceptual clustering) basé sur la théorie des treillis ([Salton et McGill, 83], [Atzeni et Stott Parker, 90]). A partir de la description d'un ensemble d'objets, l'algorithme de Godin et al. génère un treillis qui fait ressortir toutes les similitudes possibles entre tous les objets de l'ensemble. Godin et al. spécifient que leur approche est indépendante de la forme de représentation (les descriptions peuvent être des ensembles de mots-clés, des listes d'attributs-valeurs ou d'autres formes plus complexes). Dans leur article, Godin et al. [Godin et al., 86] ont illustré leur approche pour les deux premières formes, Mineau [Mineau, 90] l'a illustré pour la troisième forme en utilisant les GC. Dans Godin et al. [Godin et al., 95], les auteurs montrent une variante de l'algorithme permettant une formation incrémentale de la hiérarchie des GC. En effet, dans la formulation initiale, tous les objets sont spécifiés au départ. Avec la version incrémentale, un nouveau objet peut être intégré à la hiérarchie.
Bournaud et Ganascia [Bournaud et Ganascia, 95] utilisent l'algorithme de Mineau pour extraire, à partir du graphe de généralisation de GC, une hiérarchie de GC (une vue partielle du graphe initial).

→ Parmi d'autres travaux sur la formation (ou le raffinement) d'une hiérarchie de GC, citons celui de Liquière [Liquière, 93] qui propose aussi (comme le groupe de Godin et Mineau) un algorithme de "catégorisation conceptuelle", le travail de Champesme [Champesme, 95] qui est concerné par l'apprentissage à partir d'exemples positifs et d'exemples négatifs et enfin, le travail de Aïmeur ([Aïmeur et Ganascia, 93], [Aïmeur, 94]) en acquisition des connaissances où elle utilise des techniques de descrimination pour raffiner la taxonomie des classes.

Le problème de la formation incrémentale d'*une base de connaissances organisée selon la théorie des GC* n'a pas été toutefois considéré; les approches précitées ont été concernées par la formation d'une hiérarchie (éventuellement un graphe) de GC.
Sowa n'a pas fourni une spécification pour l'organisation d'une base de connaissances selon la théorie des GC, mais les structures conceptuelles qu'il a proposé peuvent être utilisées pour constituer une telle spécification. Par exemple, la hiérarchie des types peut être enrichie en associant à chaque type défini sa définition (certains types peuvent être non-définis, temporairement ou définitivement) ainsi que les instances du type, son(ses) canon(s) et les schémas où le type est utilisé. Considérer la formation incrémentale d'une telle base de connaissances consisterait à étudier le processus d'intégration d'une définition, d'un canon, d'un schéma ou d'une instance, à étudier le rapport entre ces structures dans l'organisation de la base ainsi que dans la formulation du processus d'intégration, à étudier comment de

nouvelles définitions et de nouveaux canons ou schémas peuvent être induits à partir des informations qui ont été intégrées, etc.

C'est cette problématique que nous tentons d'analyser dans le chapitre sur le modèle de la mémoire (en l'occurrence, la mémoire à long terme d'une application Synergy).
Dans sa revue des travaux en apprentissage, Sowa [Sowa, 84 p. 329-334] a relevé l'importance d'un modèle qui puisse tenir compte de l'intégration des connaissances en mémoire et de la formation et "l'évolution" des concepts et des schémas associés. Par ailleurs, la problématique qui nous concerne est en rapport avec le modèle de Schank [Schank 82, 91] d'une *mémoire dynamique basée sur la généralisation.* Dans la section 2.6 nous présentons brièvement ce modèle ainsi que le processus de formation incrémentale d'une hiérarchie de concepts.

2.3.5 PEIRCE: vers une plate-forme pour la communauté des GC

Pour unifier et intégrer leurs travaux, des chercheurs dans la "communauté des GC" ont proposé, depuis 1992, la plate-forme (workbench) PEIRCE [Ellis et Levinson, 92]. Des ateliers annuels sont tenus "en annexe" à la conférence internationale sur les GC. L'objectif principal étant de "standardiser" la notation textuelle de la théorie des GC, l'interface graphique et l'implantation des GC afin de faciliter le partage et la communication entre les différents groupes de chercheurs, selon leurs domaines de recherche.
En annoncant le projet en 1992, Ellis et Levinson notent que PEIRCE *va* contenir les modules suivants :

→ *Interfaces textuelle et graphique.* Depuis 1992, les travaux sont en cours pour standardiser la
notation et améliorer l'interface graphique (un prototype est disponible de ce dernier).

→ *Traitement du langage naturel.* Toujours en état de projet, bien que plusieurs chercheurs ont travaillé séparement sur cet axe de recherche ([Nagle et al., 92], [Pfeiffer and T. E. Nagle, 92], [Mineau et al., 93], [Tepfenhart et al., 94] et [Ellis et al., 95]).

→ *Ajout et recherche dans une base de GC.* Ellis et Levinson notent que ce module constitue le centre (core) de PEIRCE, il se base sur l'algorithme de classification proposé par les deux auteurs (il est aussi le sujet de thèse de Ellis). Un prototype est disponible.

→ *Catalogue conceptuel.* Le but est de constituer une ontologie commune à la communauté des GC. Les travaux sont en cours.

→ *Programmer en GC avec contraintes* (CGC). Selon Ellis et Levinson, CGC *va* intégrer les GC avec en plus des notations pour spécifier le focus et les contraintes fonctionnelles, il aura une structure de contrôle similaire à Prolog mais les GC vont remplacer les prédicats et l'unification des GC, l'unification des prédicats. Comme nous l'avons noté

précédemment, Ghosh et Wuwongse [Ghosh et Wuwongse, 93, 94, 95] ont proposé une sémantique formelle pour un tel langage (sans considérer les notations supplémentaires pour "spécifier le focus" et les contraintes fonctionnelles) mais aucune réalisation n'a toutefois été effectuée. Le langage CGC est toujours en état de projet.

→ *Mécanismes d'inférence et de démonstration de théorèmes.* Le module va réaliser les règles d'inférence de la logique de Peirce. Le groupe de Kocura travaille sur cet aspect [Heaton et Kocura, 93].

→ *Mécanismes d'apprentissage.* Ce module comprend en particulier le travail de Levinson sur son système "fully domain-independent adaptive search and game-playing system" [Levinson, 93, 94].

→ Au centre du projet PEIRCE, il y aura une hiérarchie de structures de données abstraites (abstract data types, ADT). En particulier, le ADT pour les GC comprend les différentes règles de formation (copie, jointure, restriction, etc.) et autres opérations comme la lecture et l'écriture d'un GC.

Ellis et Levinson soulignent l'intérêt d'avoir un autre module dans PEIRCE, concernant un système de vision. Depuis 1993-1994, PEIRCE comprend deux autres modules, visant à utiliser les GC dans les systèmes d'information et les bases de données relationnelles ([Levinson et Ellis, 93], [Ellis, 94]).

Le système que nous proposons dans cette thèse est complémentaire au projet PEIRCE : le langage Synergy peut être envisagé comme un "autre" module concerné par un traitement, par activation, de connaissances procédurales, le langage Prolog+CG comme une "réalisation" particulière dans le cadre du module CGC, notre modèle de la mémoire comme une "autre approche" dans le cadre du module "Apprentissage" et enfin notre hiérarchie des opérations sur les GC comme un module "Algèbres pour les GC".

2.4 Opérations et langages logiques pour les réseaux sémantiques

Plusieurs extensions conceptuelles de Prolog ont été proposées en littérature, basées sur l'utilisation d'une structure plus complexe (et plus "riche") que les prédicats ([Aït-Kaci, 84], [Aït-Kaci et Nasr, 86], [Aït-Kaci et al., 94], [Ferber et Volle, 88], [Voinov, 92], [Pletat and von Luck, 90], [Herzog and Rollinger, 91]).

Aït-Kaci a proposé une algèbre pour une forme de réseaux sémantiques (appelés Ψ-termes) ([Aït-Kaci, 84], [Aït-Kaci et Nasr, 86]). L'algèbre est basée sur la définition d'un treillis de Ψ-termes et comprend l'opération EstPlusGeneral (subsumption), l'unification et la généralisation.

Un Ψ-terme est un graphe avec une seule racine (le nom du prédicat), un nœud du graphe peut être une variable ou un identificateur, un arc représente une relation orientée et un nœud ne peut avoir des relations en sortie identiques. La racine d'un Ψ-terme constitue son point d'entrée, ce dernier est donc fixé au départ dans la structure.

Notre définition des opérations sur les GC est similaire à celle de Aït-Kaci pour lesΨ-termes, ces derniers constituent toutefois une forme plus restreinte que les GC sur lesquels sont définis nos opérations; un GC peut avoir plusieurs racines, aucun point d'entrée n'est fixé au départ dans la structure, un concept peut avoir un GC comme référent et les concepts peuvent être liés par des liens de co-références. L'imbrication des GC ainsi que les co-références permettent la représentation et le traitement de la notion de contexte (contrairement aux Ψ-terme). Enfin, notre algèbre fournit un plus grand nombre d'opérations.

Aït-Kaci et Nasr [Aït-Kaci et Nasr, 86] ont proposé un langage logique, appelé Login, basé sur les Ψ-termes. Login est une extension conceptuelle à Prolog ; au lieu des prédicats et de l'unification des prédicats, Login utilise les Ψ-termes et l'unification des Ψ-termes. Enfin, le groupe de Aït-Kaci a proposé recemment une extension à Login, appelée Life ([Aït-Kaci et Podelski, 92], [Aït-Kaci et al., 94]). Life intègre la programmation logique, la programmation fonctionnelle et une forme de programmation orientée objets ; un Ψ-terme est utilisé pour décrire une classe (et plus précisement, ses attributs qui constituent la partie descriptive de la classe) et le programmeur peut définir une hiérarchie de classes avec possibilité d'héritage.

2.5 Traitements de connaissances basés sur l'activation de graphes

2.5.1 Réseaux sémantiques actifs

Norman et al. [Norman et al., 75] ont proposé une forme de réseau sémantique, appelée *active structural network ; ASN* qui est un graphe composé de nœuds-concepts connectés par des arcs-relations orientés, un concept peut représenter un type, appelé *primary node* (comme "person" et "move"), une instance, appelée *secondary node* (comme <Mary>), ou une variable. Un ASN peut représenter un prédicat (par exemple, give[agent, object, recipient, time]) ou une proposition qui est une instanciation d'un prédicat (par exemple, give[<Mary>,<dollar>,<John>,time]). Un concept peut, par exemple, être une proposition (Figure 2.6a, extraite de [Norman et al., 75, p. 43]).

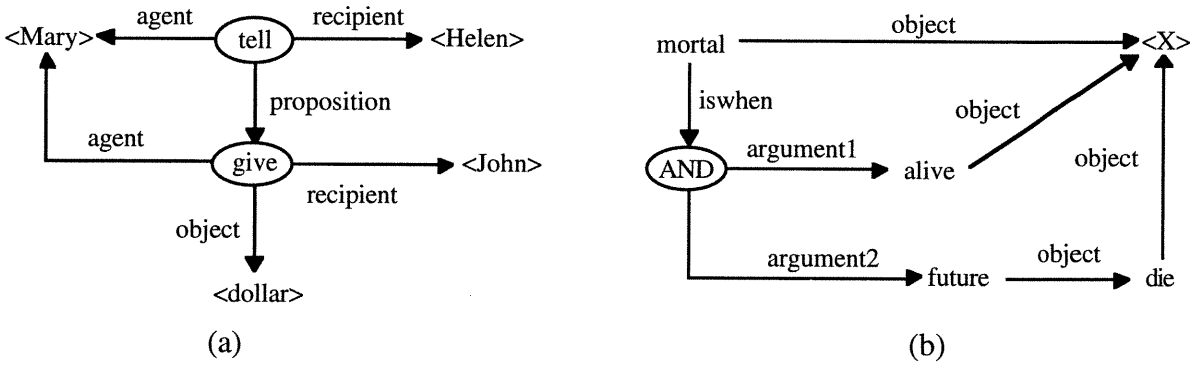


Figure 2.6 : (a) Réseau sémantique avec une proposition comme concept,
(b) Définition d'un type

Les auteurs considèrent également les quantificateurs, les structures conceptuelles comme la définition, un prototype et un schéma. La Figure 2.6b montre la définition de "mortal".

Une des caractéristiques des ASN est "qu'un ASN représente aussi bien une donnée qu'un processus" [Norman et al., 75, p. 35]. Considérons par exemple la définition d'un sens du type "son" (Figure 2.7, extraite de [Norman et al., 75, p. 168]).

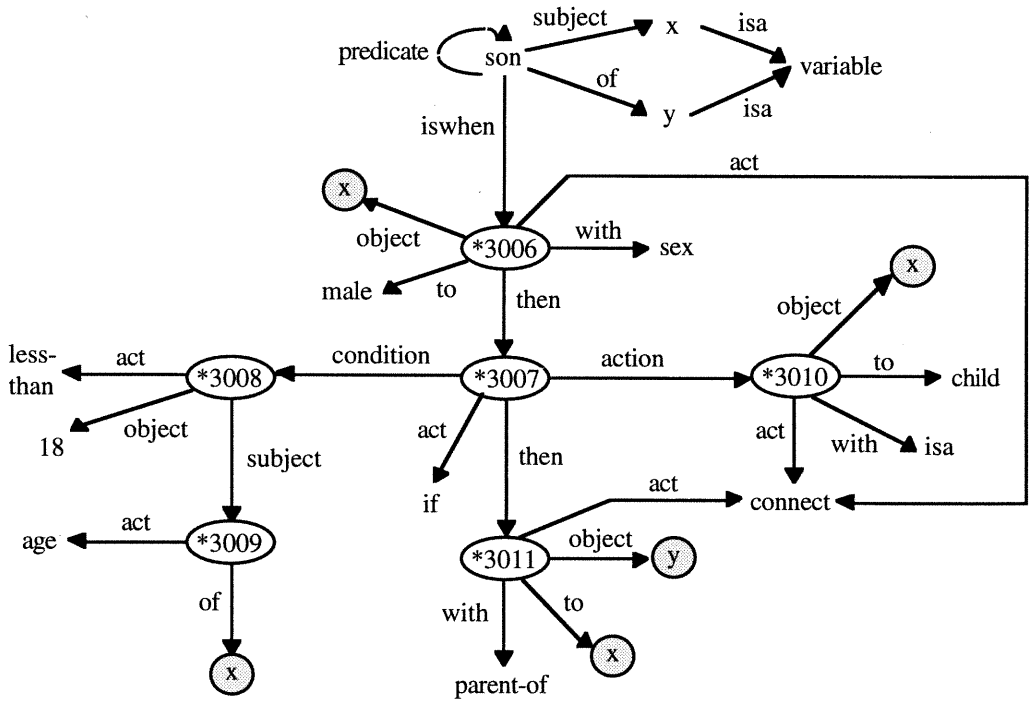


Figure 2.7 : Définition du concept son

La spécification textuelle de la définition fait mieux ressortir son contenu procédural (le réseau de la Figure 2.7 est en fait généré à partir de la spécification suivante) :

Define son as predicate.
The definition frame for son is : x son of y.
The definition is :
Connect x to male with sex.
If age of x is less than 18, then connect x to child with isa.
Connect y to x with parent-of.
##

Le corps de la définition est donc une séquence de trois "instructions", décrivant le traitement à effectuer lorsqu'on affirme le prédicat "son" à propos de deux entités. En particulier, des relations seront établies, en tenant compte de certaines conditions (ici "l'age de x < 18").
Un nœud de la structure ASN peut être une action particulière, reliée à son type par le lien "act" (comme pour différents nœuds de la Figure 2.7). Le type peut correspondre à une opération primitive ou à un type défini en ASN (comme "son"). Dans le premier cas, le type est relié à la primitive par le lien "prim" et dans le second cas, par le lien "iswhen" (comme pour "son" dans Figure 2.7). Les deux "structures de contrôle" de base sont la séquence, formulée avec le lien "then", et l'alternative.
Considérons maintenant l'interpréteur des ASN qui est initié par la demande d'évaluation d'un nœud X en mémoire (qui est un ASN). L'évaluation est réalisée comme suit ([Norman et al., 75] p. 168) :

1. Évaluer tout argument de X qui est susceptible de l'être. Remplacer ensuite l'argument par sa valeur.
2. Suivre le lien "act" pour arriver au type du nœud X.
3. Transférer les valeurs des arguments aux paramètres correspondants, qui sont associés au type.
4. Si le type du nœud est relié par "prim" à une primitive alors exécuter la primitive,
   sinon évaluer le nœud Y pointé par la relation "iswhen" : X -iswhen-> Y.
5. Si le nœud X est relié par "then" à un nœud Z : X -then-> Z, alors évaluer Z.

Remarque : Notons la similitude entre les "structures conceptuelles exécutables ; SCE" de Lukose et al. [Lukose, 93] et les ASN de Norman et al.; dans un SCE un envoi de message à l'action (par exemple à obtain) déclenche l'acteur associé (qui est une "primitive", par rapport à la structure conceptuelle), dans les ASN une demande d'évaluation d'un nœud (par exemple obtain) déclenche l'interpréteur. La préparation de l'exécution de l'acteur consiste à considérer les préconditions et à effectuer les associations (binding) nécessaires au niveau de la mémoire à court terme. Dans les ASN, la préparation correspond à l'évaluation des arguments et à l'association (binding) entre les arguments et les paramètres. Les deux approches sont concernées par un traitement séquentiel, avec une formulation explicite de la séquence (la relation follow_by dans la structure de Lukose et al. et la relation then dans les ASN).
Parmi les différences citons les suivantes : d'une part, les ASN fournissent l'alternative pour exprimer une "instruction conditionnelle" et la possibilité de définir des actions en ASN, ce

qui n'est pas le cas des structures de Lukose (tout acteur est une primitive; son corps n'est pas décrit par un GC mais par une structure Prolog) et d'autre part, un nœud (concept) dans une structure de Lukose peut être une séquence d'actions ou une collection de règles et la sémantique d'une action primitive correspond à celle proposée dans STRIPS.

Norman et al. ont proposé le langage SOL (Semantic Operating Language) pour leur système MEMOD (memory model) qui comprend une mémoire (appelée "node space") représentée comme un ASN contenant toute la connaissance du système, un analyseur du langage naturel ("parser") et un interpréteur. SOL est en fait un sous ensemble du langage naturel (la spécification ci-dessus de la définition de "son" est en SOL), utilisé pour ajouter de l'information dans la mémoire et aussi pour poser des questions qui sont ensuite évaluées par l'interpréteur. Le système MEMOD a été utilisé comme un *système cognitif* capable de différentes tâches cognitives : compréhesion du langage naturel, question/réponse, perception, résolution de problèmes et raisonnement [Norman et al., 75].
La mémoire dans MEMOD est un ASN organisé comme un graphe de généralisation, selon le lien de généralisation entre les types. Aux types, sont associés les définitions et autres structures conceptuelles.
Dans Rumelhart et al. [Rumelhart et al., 72], les auteurs proposent "les grandes lignes" d'un modèle de formation dynamique de la mémoire, basé sur *la généralisation des concepts* (les attributs communs à certains concepts sont "déplacés" au concept père) et *la spécialisation par subdivision des concepts* (des attributs communs à certains concepts et non à d'autres vont impliquer une subdivision et éventuellement la création de concepts intermédiares). Le système MEMOD n'incorpore toutefois pas ce modèle.

SOL est une tentative intéréssante vers une *programmation par activation conceptuelle de réseaux sémantiques*, il constitue toutefois une première tentative qui n'a pas été approfondie et développée par ses auteurs (ces derniers ont initié, dans les années 80, le mouvement des "réseaux connexionistes" [Rumelhart et al., 86]). Parmi les limitations du langage SOL citons les suivantes : les ASN ne fournissent pas un moyen de formuler et de traiter les notions de contextes et de co-référents (cela a un impact entres autres sur l'organisation d'un "programme" SOL), l'évaluation d'un ASN est séquentielle, SOL adopte une sémantique procédurale limitée (il ne peut formuler des processus concurrents) et il n'est pas concerné par d'autres formes de programmation comme la programmation par accès, la programmation fonctionnelle ou la programmation orientée objet (rappelons qu'il a été proposé en 1975).
La remarque sur SOL s'applique également au modèle de formation de la mémoire [Rumelhart et al., 72] ; une tentative intéréssante qui n'a pas été approfondie et développée par ses auteurs (on la retrouve, sous d'autres formes, dans les travaux en apprentissage par

induction [Michalski, 83] et dans le modèle de la mémoire dynamique de Schank [Schank, 82]). En fait et comme le note Rumelhart et al. [Rumelhart et al., 72], l'article ne présente pas un modèle clairement défini, mais plutôt quelques éléments de base (comme les opérations de généralisation et de subdivision des concepts) qu'un modèle de la mémoire devrait inclure ainsi qu'une discussion générale sur l'impact d'un tel modèle.

2.5.2 Réseaux de "frames"

La programmation par accès a été introduite avec la notion de "frame" et ensuite avec les langages orientés frames [Masini et al., 90]. Comme les classes dans la programmation orientée objet, les frames sont généralement organisés en une hiérarchie d'héritage, un cadre ("frame") est composé généralement d'attributs ("slots"), chacun peut avoir plusieurs facettes ("facets"), une facette peut être "statique" (comme le type ou la valeur par défaut) ou "dynamique" (comme si-present, si-besoin, verifieType) ; la valeur d'une facette dynamique correspond à un traitement (appelé *attachement procédural* ou *demon*) :

```
(entite
    (attribut1
            (facette11  valeur11)
            (facette12  valeur12)
            ... )
    (attribut2
            (facette21  valeur21)
            (facette22  valeur22)
            ... )
    ...)
```

Parmi les facettes dynamiques, citons les trois suivants : 1) *si-présent* (si la valeur de l'attribut vient d'être calculée alors exécuter le traitement associé à "si-présent"), 2) *si-besoin* (si la valeur de l'attribut est demandée alors exécuter le traitement associé à "si-besoin") et 3) *verifieType* (exécuter le traitement si l'attribut reçoit une nouvelle valeur).
Les facettes si-present et verifieType sont déclenchées par chaînage avant : une mise à jour de la valeur de l'attribut va les déclencher. La facette si-besoin est déclenchée par chaînage arrière : une demande de valeur déclencherait si-besoin.
A noter que l'exécution d'une facette dynamique d'un attribut peut changer ou demander la valeur d'un autre attribut du cadre courant ou d'un autre cadre, constituant ainsi un graphe de dépendance sous-jacent aux cadres "actifs".

Des combinaisons de classes et de frames ont été proposées en littérature [Masini et al., 90] : un frame peut avoir en plus des méthodes, intégrant ainsi la programmation par accès et la programmation par envoi de messages. Enfin, des systèmes intègrent également des règles de production ([Mattos, 90], [Jackson, 90]) : la valeur d'un attributs peut être une collection

(une petite base) de règles. Les règles sont déclenchées par chaînage avant (la présence des prémisses d'une règle déclenche cette dernière pour affirmer sa conclusion) et/ou arrière (le besoin de vérifier la conclusion d'une règle déclenche cette dernière afin de vérifier ses premisses). Comme pour les attributs et les facettes des frames, notons qu'une conclusion d'une règle est généralement une prémisse dans une autre règle, constituant ainsi un graphe d'inférence.

Note : Comme le montre la prochaine section, les graphes de flot de données sont également évalués en chaînage avant (par le flot de données) et/ou en chaînage arrière (par le flot de demande de valeurs).

2.5.3 Graphes de flot de données

Nous considérons en premier le modèle de graphe de flot de données et ensuite des travaux d'intégration avec d'autres modèles.

→ Différents modèles et langages de graphes de flot de données (GFD) (appelés aussi "graphe de dépendances fonctionnelles") ont été proposés dans la littérature ([Ackerman, 82], [Myers, 82], [Filman et Friedman, 84], [Glauert, 84], [Thakkar, 87], [Arvind et Nikhil, 87], [Gaudiot, 89]). Les GFD sont une formulation orientée graphe du modèle fonctionnel et sont appropriés pour une *programmation visuelle* ([Davis et Keller, 82], [Hils, 92], [Harvey et Morris, 95]).

Un GFD est généralement composé d'acteurs (fonctions); un acteur reçoit des données des autres acteurs, les consomme et produit, après évaluation de son corps, des sorties qui sont envoyées à d'autres acteurs. Nous avons ainsi un flot de données qui anime le graphe. A l'inverse de la programmation procédurale, une zone mémoire n'est pas associée à une donnée.

Un acteur se déclenche dès que toutes les données en entrée sont disponibles et tous les arcs en sortie sont libres (ils ne contiennent pas des données non encore consommées).

Les acteurs peuvent être définis et l'activation peut être dirigée par le flot de données et/ou par le flot de "marqueurs de demande (demand token)", les deux flots correspondent aux chaînages avant et/ou arrière, utilisés par les mécanismes d'inférence dans les bases de connaissances (et aussi dans les systèmes orientés "frames" concernant l'activation des "démons").

Toutes les entrées/sorties d'un acteur dans un GFD correspondent à ses arguments. L'acteur est contrôlé uniquement par le flot de données, ce flot peut être toutefois lui-même contrôlé par des acteurs spéciaux (figure 2.8) :

- un *prédicat* consomme ses arguments en entrée et produit un booléen (jeton-booléen de contrôle) qui ne peut être en entrée que pour les quatre autres types d'acteurs (le jeton de contrôle est véhiculé par le lien de contrôle ; lien discontinu).
- Les acteurs *porte-T* et *porte-F* laissent passer la donnée en entrée si le booléen est T(rue) ou F(alse) respectivement.
- L'acteur *sélecteur* laisse passer la donnée arrivant à la partie T ou F du sélecteur selon la valeur T ou F du booléen.
- L'acteur *distributeur* achemine sa donnée dans sa partie T ou F selon la valeur T ou F du booléen.

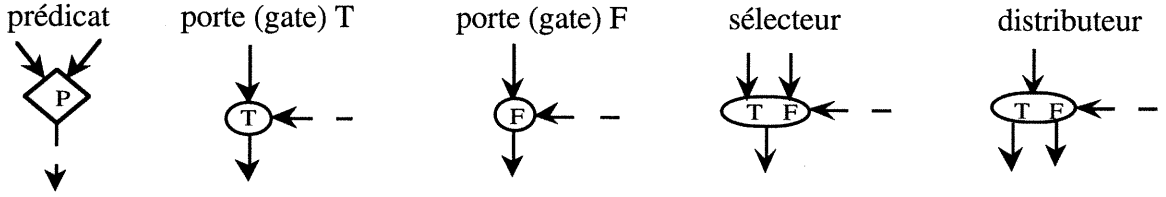


Figure 2.8 : Acteurs de contrôle dans un GFD

Pour établir une dépendance de contrôle entre deux acteurs qui ne sont pas en dépendance fonctionnelle, il faudrait élaborer et imposer une *dépendance entre les flots de données* associés aux deux acteurs. La Figure 2.9 en montre un exemple : la sortie de l'acteur opr1 est dupliquée par l'acteur de duplication représenté par le point noir, une des copies est envoyée au prédicat Pa qui ne fait que consommer la valeur (c'est le moyen de détecter que opr1 a terminé son exécution), sa sortie contrôle la porte-T qui a été introduite pour contrôler le flot de données de opr2. En effet, opr2 ne peut pas s'exécuter car la porte-T doit attendre le booléen en sortie de Pa.

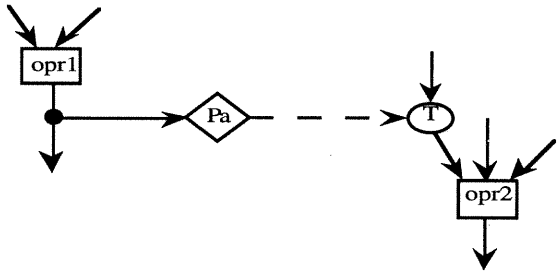


Figure 2.9 : Synchronisation d'acteurs dans un GFD

L'exécution/activation des GFD est contrôlée par le flot de données, la synchronisation entre acteurs doit donc être reformulée en termes de flot de données et d'acteurs primitifs. Aussi et à cause du modèle fonctionnel sous-jacent à la famille GFD, cette dernière ne peut formuler

une connaissance "purement" procédurale ; un acteur doit avoir au moins une donnée en entrée et en principe, au moins une donnée en sortie [Arvind et Nikhil, 87]. Ainsi, on ne peut pas reformuler en GFD un graphe de tâches où le contrôle sur les flots de données associés aux tâches ne peut pas se faire, pour la simple raison qu'il peut ne pas y avoir de flot de données ! Il faut "bricoler" le graphe et attribuer aux tâches des "données fictives". Enfin et comme pour la synchronisation, la notion "classique" d'objet ayant une valeur qui peut changer durant l'exécution du programme ne peut être formulée directement en GFD, il faut la simuler.

→ Des chercheurs ont proposé l'intégration du modèle fonctionnel sous-jacent à la famille GFD et celle du modèle procédural sous-jacent à la programmation procédurale séquentielle/parallèle ([Treleaven et al., 82], [Schreiner, 92]). Amamiya et Hasegawa [Amamiya et Hasegawa, 84] ont considéré la transmission par-valeur/par-référence des données et les évaluations stricte/paresseuse et par-donnée/par-demande des acteurs dans le cadre du traitement flot de données. Enfin, Kluge [Kluge, 86] utilise les réseaux de Pétri pour une analyse des éléments de base des modèles de traitement orienté données (graphes de flot de données), traitement orienté contrôle (traitement procédural) et traitement par réduction (modèle fonctionnel basé sur l'activation par la demande).
Considérons par exemple le modèle combiné de Treleaven et al.; deux flots co-existent : flot de données (des jetons-données) et flot de contrôle (des jetons-contrôles). De même, les deux formes de données co-existent : données avec consommation (comme dans les GFD) et données avec assignation-multiple (la notion classique de variable; une référence à une zone mémoire). Pour ce dernier cas, c'est le nom de la variable qui est envoyée comme jeton. En tant que données transmises à un acteur, la première forme correspond à un transfert par valeur et la seconde à un transfert par adresse.
Une instruction (un acteur) peut avoir en entrée et produire en sortie des jetons-données (valeur et/ou nom de variable) et/ou jetons-contrôle. Une instruction peut aussi accéder à des valeurs et/ou variables spécifiées directement dans son corps. Une instruction est déclenchée dès que toutes ses entrées sont disponibles. La figure 2.10 (une adaptation d'un exemple fourni dans [Treleaven et al., 82]) illustre cette forme d'intégration des deux modes de traitement (fonctionnel et procédural) : les instructions I1 et I2 n'ont pas de données en entrée et elles sont déclenchées par des jetons-contrôle. L'instruction I1 a accès dans son corps à une valeur (2) et à une variable (y). Les deux valeurs en sortie de ces deux instructions sont les entrées de l'instruction I3, celle-ci est donc déclenchée par des jetons-données, elle affecte une valeur à la variable x et produit en sortie un jeton-contrôle.

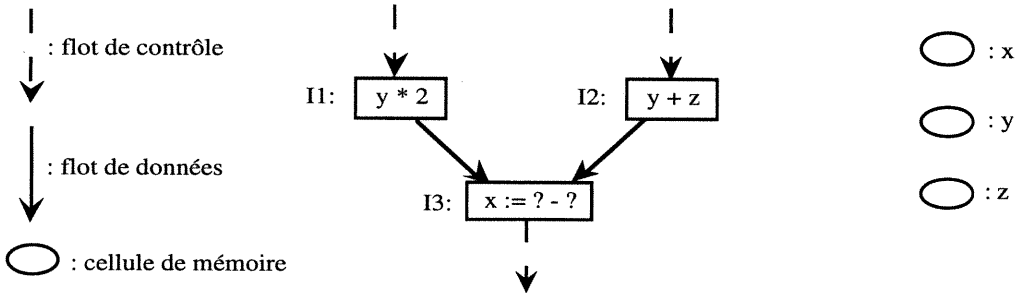


Figure 2.10 : Exemple de traitement dans le modèle combiné de Treleaven et al.

L'intégration des deux modèles (procédural et fonctionnel) permet la modélisation d'une grande variété d'applications ([Treleaven et al., 82], [Schreiner, 92]). Cette combinaison est appropriée, entres autres, pour le génie logiciel ([Pressman, 92], [Shlaer et Mellor, 92]) et elle le serait plus si on y intègre un formalisme de modélisation conceptuelle des données. Les GC peuvent constituer un tel formalisme ([Nagle et al., 92], [Pfeiffer and T. E. Nagle, 92], [Mineau et al., 93], [Tepfenhart et al., 94] et [Ellis et al., 95]). Les graphes de flot de données de Sowa constituent un modèle "purement" fonctionnel. Dans [Kabbaj, 93], nous avons proposé des modifications et des ajouts à la proposition de Sowa afin de considérer également le traitement procédural. Le langage *CAL* incorpore notre proposition [Kabbaj et Frasson, 94]. Nous avons proposé par la suite une extension orientée objet de CAL [Kabbaj et Frasson, 95a]. CAL "hérite" des graphes de flot de données de Sowa la distinction entre graphe d'acteurs et graphe conceptuel, avec la possibilité de les combiner en un graphe "hybride". Nous avons constaté par ailleurs que les acteurs sont (après tout) des concepts (une procédure, une fonction ou une tâche sont aussi des concepts) et que les GC peuvent jouer le rôle de graphe d'acteurs CAL. Une reformulation (et non une traduction) en GC des éléments de base du langage CAL a été donc effectuée [Kabbaj et Frasson, 95b]. Une dernière étape, dont le résultat est le langage Synergy, a été de compléter l'extension orienté-objet, d'améliorer l'intégration des éléments du langage, d'établir un lien entre le langage et notre modèle d'organisation et de formation de la mémoire et de rendre le langage aussi simple et uniforme que possible. L'implantation du langage et son utilisation ont influencé cette dernière étape.

2.5.4 Réseaux de Pétri (RP)

Comme pour les graphes de flot de données, nous introduisons brièvement le modèle réseau de Pétri et ensuite des travaux d'intégration avec d'autres modèles.

→ Les Réseaux de Pétri (RP) ([Brauer, 80], [Peterson, 81]) permettent la modélisation parallèle d'un système en terme de places (qui représentent les états du système) et de transitions. Une place P peut contenir des jetons et un jeton signale la *présence* d'une ressource dans la place P. Les jetons et les transitions sont "non-analysables" : un jeton ne possède ni type ni valeur et les transitions n'utilisent pas les jetons et ne font aucun traitement.

Pour une transition dans un RP, on spécifie les entrées et les sorties sans distinguer entre les données, qui correspondent aux arguments de l'opération, des booléens de contrôle, qui représentent des pré- ou post- conditions de l'opération. On ne peut pas donc assimiler (sans risque de confusion) une opération avec ses arguments en entrée et en sortie à une transition avec ses entrées et ses sorties. En d'autres termes, avec les RP on ne distingue pas la dépendance fonctionnelle (dûe au flot de données) de la dépendance de contrôle (flot de contrôle); un jeton peut représenter aussi bien une donnée qu'un booléen de contrôle.

Une transition est déclenchée si toutes les places en entrée ont des jetons, l'exécution d'une transition est alors instantanée : elle consomme un jeton de chacune de ses places en entrée et produit un jeton dans chacune de ses places en sortie. L'activation d'un RP résulte du flot de jetons.

→ Les RP de base constituent une forme très abstraite pour décrire un traitement parallèle. L'utilisation ou non de ces réseaux pour la modélisation d'un système dépend du niveau d'abstraction désiré. Dans certains cas, le type d'abstraction incorporé dans les RP suffit; dans d'autres cas il ne l'est pas et il faut plus de précision sur les jetons, les liens et les transitions.

Cela explique les diverses extensions apportées aux RP de base, par exemple : rendre les jetons typés et valués ([Peterson, 80], [Jensen, 86, 91], [Genrich, 86], [Baldassari et Bruno, 88], [Bail et al., 92]), diversifier les types de liens, attacher des expressions aux arcs, introduire de nouveaux liens comme les liens de tests et liens d'inhibition ([Peterson, 81], [Törn, 81, 85, 88], [Jensen, 86, 91], [Christensen et Hansen, 93]), diversifier les types de transitions [Evans, 93], associer un traitement à une transition [Jensen, 91], rendre la formulation plus abstraite ([Malec, 91], [Hee et al., 93]), etc. Considérons en particulier les Réseaux de Pétri Colorés et Hiérarchisés et les extensions objets des RP :

→ Dans un Réseau de Pétri Coloré et Hiérarchisé (Hierarchical Coloured Petri Net; HCPN [Jensen, 91], [Malhotra et Shapiro, 93]), une place possède une couleur de l'ensemble ("color set", l'équivalent du type) et un marquage ("marking", l'équivalent d'une ou plusieurs valeurs). Des expressions sont associées aux arcs en entrée d'une transition et suite à leur évaluation, elles déterminent les jetons que la transition doit

consommer. Il en est de même pour les arcs en sortie d'une transition; les expressions déterminent les jetons que la transition doit produire. Par ailleurs, une transition possède un corps composé d'une partie condition ("guard") qui est une liste d'expressions booléennes et d'un segment code qui est exécuté à chaque activation de la transition. Enfin, une transition peut être définie en terme d'un RP.
Christensen et Hansen [Christensen et Hansen, 93] étendent les RPCH avec les arcs tests et les arcs inhibiteurs.

→ Un intérêt considérable a été consacré ces dernières années à l'intégration des RP avec les techniques des modèles orientés objets. Dans *ExSpect* [Hee et al., 93], les jetons correspondent à des objets structurés et les transitions à des processus. Un objet possède un "time-stamp" qui indique à partir de quand l'objet peut être consommé par un processus. Les objets sont décrits avec une extension du modèle entité-relation ("binary data model with object-oriented ideas" selon les auteurs) et les processus par un langage fonctionnel. Lakos [Lakos, 95] présente, de façon formelle, comment les Réseaux de Pétri Colorés (Coloured Petri Nets; CPN) peuvent être étendus à des Réseaux de Pétri Objets (Object Petri Nets; OPN).

2.5.5 Diagrammes de transition d'états

Les diagrammes de transition d'états sont très utilisés en génie logiciel, en conjonction avec d'autres formalismes orienté-graphes (comme les diagrammes de flot de données et les diagrammes entité-relation) [Pressman, 92]. Considérons par exemple les méthodes d'analyse orientées objets ([Coad et Yourdon, 91], [Jacobson, 92], [Rumbaugh et al., 91], [Shlaer et Mellor, 92]). Ces dernières produisent un *modèle des objets*, basé en général sur une extension du modèle entité-relation, un *modèle dynamique* décrivant le comportement du système en termes de diagrammes de transition d'états et un *modèle fonctionnel* où les méthodes des objets ainsi que les actions dans le modèle dynamique sont définies en termes de diagrammes de flot de données.
Pour l'activation des diagrammes de flot de données, considérons à titre d'exemple les "actions data flow diagrams; ADFD" de Shlaer et Mellor : toute donnée disponible est marquée par un jeton, si tous les arguments en entrée et les liens de contrôle d'un processus sont marqués et si le processus n'a pas encore été exécuté, le processus est alors exécuté et à sa terminaison, ses sorties sont marquées (aussi bien les données que les contrôles). L'activation continue ainsi jusqu'à ce qu'aucun processus ne puisse être exécuté.

Une des remarques souvent citée dans la littérature concernant les méthodes orientées-objets mentionnées ci-dessus est la multiplicité des modèles et la tendance à focaliser plus sur le

modèle des objets que sur les modèles dynamique et fonctionnel ([Moreira et Clark, 94], [Liddle et al., 94], [Ebert et Engels, 94]).

Un formalisme/langage qui permettrait une intégration des modèles est donc souhaitable. Des propositions ont été rapportées dans la littérature, basées généralement sur un modèle concurrent orienté objet avec parallélisme inter- et intra- objet et héritage (éventuellement). C'est le cas par exemple des deux approches suivantes :

→ Moreira et Clark [Moreira et Clark, 94] proposent l'utilisation de LOTOS comme unique formalisme pour couvrir les différents aspects de l'analyse.

→ Le modèle "Object-oriented System Modeling" [Liddle et al., 94] en est un autre exemple. OSM est utilisé pour les différentes phases du développement d'un système. Les objets sont structurés en une hiérarchie et un objet possède une structure et un comportement : la structure décrit les composantes de l'objet (par exemple "Person has Name ; Person has Address") et le comportement d'un objet est décrit par un réseau d'états/transitions appelé "state net". Une transition peut avoir plusieurs états en entrée (comme pré-conditions), plusieurs états en sortie et elle est composée d'une partie condition (appelée "trigger") et d'une partie action (une procédure). Un état peut être "on" ou "off". Une transition est déclenchée quand des états en entrée sont à "on" et si en plus la partie condition est vérifiée, la transition est alors activée.

Notons la similitude du "state net" avec certains réseaux de pétri de haut niveau : un état à "on/off" correspond à une place avec/sans jeton et la transition avec les deux parties condition/action est commune à divers modèles de réseaux de pétri de haut niveau (en particulier les RPCH [Jensen, 91]). Comme d'autres extensions objets des RP [Lakos, 95], "state net" permet la communication entre objets : une action dans une transition (qui se trouve dans le "state net" d'un objet) peut contenir un envoi (ou une réception) de message à/d'un autre objet.

2.5.6 Réseau dynamique d'objets actifs : programmation concurrente orientée objet

La programmation concurrente orientée objet (PCOO) peut être considérée comme une intégration de la programmation concurrente par processus et la programmation orientée objet.

Une multitude de modèles et de langages ont été développés pour la programmation, l'analyse et les base de données concurrentes orientées objets ([Yonezawa et Tokoro, 87], [Treleaven, 90], [Yonezawa, 90], [Tokoro et a., 91], [Agha et a., 93], [Nishio et Yonezawa, 93], [Madsen et al., 93], [Ciancarini et al., 94], [Bertino et Urban, 94], [Olthoff, 95]).

Synergy incorpore un modèle de PCOO. Pour y fournir un "background" général, nous introduisons en premier un exemple de langage pour la PCOO et nous concluons ensuite avec une description générale de la PCOO. Le chapitre 6 complète cette introduction et présente le modèle de la PCOO dans Synergy.

Le langage ABCL/1 (An object-Based Concurrent Language) [Yonezawa, 90] est représentatif pour ce type de programmation. Un agent dans ABCL/1 est modélisé comme un objet actif caractérisé par une mémoire locale, un comportement et une modalité décrivant le cycle de vie de l'objet. Un objet possède une queue jouant le rôle d'un tampon pour sauvegarder les messages reçus par l'objet. La mémoire locale décrit l'état de l'objet à un moment donné de sa vie; elle correspond à un ensemble de variables. Le comportement (appelé aussi "script") correspond à un ensemble de procédures (appelées méthodes), chacune est décrite par un modèle de message (message pattern), des contraintes et une suite d'actions. Une action peut correspondre à un traitement sur la mémoire de l'objet, un envoi de message, une attente de réception de message, ou la création d'un objet.
Un objet peut être à l'état *dormant*, *actif* ou *attente* : initialement l'objet est dormant ; inactif. Si sa queue des messages est vide, l'objet demeure dans son état dormant, sinon la queue est parcourue séquentiellement et tout message qui ne peut être accepté est rejeté avec éventuellement un traitement d'erreur. Dès qu'un message satisfait le modèle de message et les contraintes d'une des méthodes du script, l'objet devient actif et commence l'exécution du corps de la méthode (les actions). S'il termine l'exécution et qu'il n'y a aucun autre message, l'objet retourne dans le mode dormant. Si un objet à l'état actif exécute l'action d'attente de réception de messages et si la queue des messages contient un message qui peut s'apparier à un des modèles de messages de l'action d'attente, alors les actions associées au modèle seront exécutées, sinon l'objet se mettra dans le mode attente.
Un envoi de message est caractérisé par son type et son mode. Trois types d'envois sont considérés : 1) envoi de type *passé* qui correspond à un envoi asynchrone : l'objet envoit le message et continue son exécution sans attendre de réponse, 2) envoi de type *présent* qui correspond à un envoi synchrone : l'objet envoitle message et attend une réponse de l'objet receveur ou d'un objet délégué, 3) envoi de type *futur* : suite à l'envoi du message, l'objet spécifie une variable où le receveur du message peut entreposer la réponse et il poursuit ensuite son exécution. Si plus tard, l'objet a besoin du contenu de la variable et si cette dernière n'a pas encore de valeur, il se mettra alors en attente.
Un envoi de messages peut être en mode *normal* ou *express* : dans le premier cas, si l'objet reçoit un message pendant qu'il est actif, le message est alors enregistré dans la queue des messages, dans le second cas, l'objet peut être interrompu par la réception d'un message.

ABCL a été utilisé pour développer des applications très variées : un résolveur de problème distribué (distributed problem solving), la modélisation de processus cognitifs humain, des systèmes à temps réel et des systèmes d'exploitation, des systèmes d'information, la simulation distribuée, le traitement scientifique, etc. ([Yonezawa et al., 87], [Shibayama et Yonezawa, 87], [Yonezawa, 90]).

Comme nous l'avons souligné au début de la section, d'autres modèles de PCOO ont été proposés en littérature. En général, un programme en PCOO est composé d'objets actifs qui s'exécutent en parallèle et qui intéragissent par envoi/réception de messages. Un objet actif possède en général une boîte de message (appelée aussi "queue de message"), une partie déclarative composée de variables et une partie procédurale composée de méthodes. Une méthode est composée généralement d'une partie condition sur le message ("message pattern") et d'une partie traitement conditionné éventuellement par des contraintes. Un programme concurrent orienté objet est composé généralement d'une collection d'objets actifs qui s'envoient des messages, ces derniers établissent des liens "dynamiques" entre les objets et l'exécution du programme produit un réseau dynamique où des nœuds (objets actifs) et des liens (de communication) sont continuellement ajoutés/éliminés.
L'exécution au sein d'un objet est souvent séquentielle (comme dans ABCL) bien que certains modèles de PCOO supportent un parallélisme intra-objet ([Saleh et Gautron, 91], [Kaiser et al., 93], [Wakita, 93], [Sargeant, 93], [Liddle et al., 94], [Guerraoui, 95]), en permettant par exemple à plusieurs méthodes de s'exécuter en parallèle. Enfin, certains modèles de PCOO supportent en plus les notions de classes, hiérarchie de classes et héritage ([Papathomas, 91], [Matsuoka et Yonezawa, 93]).

La PCOO constitue un paradigme très important, utilisée dans différents domaines et intégrée à plusieurs autres paradigmes (procédural, fonctionnel, graphes de flot de données, réseaux de Pétri, diagrammes de transition, etc.). Aucune tentative n'a toutefois été effectuée pour utiliser les GC (ou une extension des GC) comme "langage" de PCOO. La PCOO bénéficierait aussi d'une telle intégration puisque les GC constituent un formalisme approprié pour la modélisation conceptuelle des connaissances (déclaratives).
Nous voulons combler ce "manque" en proposant un langage qui utilise les GC pour intégrer différents modèles de programmation, y compris la PCOO.

2.6 Formation incrémentale d'une hiérarchie de concepts et mémoire dynamique basée sur la généralisation

Cette section complète la section 2.3.4; nous introduisons brièvement des modèles de *formation incrémentale d'une hiérarchie de concepts* puis le modèle d'une *mémoire dynamique basée sur la généralisation* proposé par Schank [Schank, 82, 91].

2.6.1 Formation incrémentale d'une hiérarchie de concepts

Notre présentation est basée sur l'étude de Gennari et al. [Gennari et al., 89] qui proposent une étude comparative de trois modèles de formation incrémentale d'une hiérarchie de concepts : EPAM qui est basé sur la notion de réseau de descrimination ([Feigenbaum, 63], [Feigenbaum et Simon, 84]), UNIMEM qui est basé sur le modèle de Schank d'une mémoire dynamique basée sur la généralisation [Lebowitz, 85, 86] et COBWEB qui est basé sur une approche probabiliste [Fisher, 87]. Les auteurs présentent ensuite CLASSIT, une extension "plus probabiliste" de COBWEB. Nous considérons brièvement les trois premiers modèles.

Le processus de formation de concepts correspond à une *catégorisation conceptuelle* (conceptual clustering [Michalski et Stepp, 83]) *incrémentale* [Gennari et al., 89]. La catégorisation conceptuelle consiste à former une hierarchie de concepts à partir d'un ensemble d'instances (qui peuvent décrire des objets ou des événements). Par incrémentale, Gennari et al. réfèrent non seulement au fait que les instances sont accéptées par l'agent une à la fois mais aussi que le processus de formation n'ajoute pas la nouvelle instance à l'ensemble des instances déjà traitées pour "recalculer" la hiérarchie à partir du nouveau ensemble.

La formation incrémentale d'une hiérarchie de concepts est décrite comme un processus d'apprentissage qui construit une hiérarchie de concepts suite à une classification descendante des instances : partant de la racine de la hiérarchie (le concept le plus général), le processus cherche où insérer l'instance, comme dans un trie selon un arbre de décision, le choix de la branche à suivre ne dépend toutefois pas de la valeur d'un seul attribut. Aussi, l'instance *peut* suivre plusieurs branches à la fois et elle *peut* être considérée comme une instance d'un concept placé à un niveau supérieur dans la hiérarchie (il n'est donc pas nécessaire de descendre jusqu'au niveau "feuille" comme dans le cas avec un arbre de décision). Une des extensions de CLASSIT par rapport à COBWEB est d'incorporer justement cette dernière possibilité.

La classification descendante des instances construit la hiérarchie par *subdivision* (divisive manner), d'autres approches se basent sur une classification ascendante des instances

([Hanson et Bauer, 89], [Danyluk, 95]) ; la hiérarchie est construite par *agglomération* (agglomerative approach).

A part la classification descendante des instances, le processus de formation incrémentale d'une hiérarchie de concepts est caractérisé par un apprentissage efficace, non-supervisé (les instances n'ont pas été pré-classées au départ) et qui peut être considéré comme une recherche dans un espace de hiérarchies de concepts selon une stratégie "hill climbing" [Gennari et al., 89]. Les auteurs notent que chacune des caractéristiques ci-dessus est utilisée dans d'autres approches d'apprentissage de concepts, c'est l'intégration de ces caractéristiques dans une seule approche qui caractérise la formation incrémentale de concepts.

Parmi les autres approches d'apprentissage de concepts, citons l'apprentissage d'un (ou peu de) concept(s) à partir d'exemples ([Winston, 75], [Mitchell, 82] et [Michalski, 83]), les concepts appris se trouvent à un seul niveau d'abstraction; ils ne composent pas une hiérarchie [Gennari et al., 89] et l'induction d'un arbre de décision [Quinlan, 86] où un nœud non-terminal correspond à un test sur un attribut et non à la description intentionnelle d'un concept (comme dans une hiérarchie de concepts). Winston et Mitchell adoptent un apprentissage supervisé et l'approche de Michalski et de Quinlan est non-incrémentale.

Considérons maintenant et brièvement les trois systèmes (EPAM, UNIMEM et COBWEB).

→ EPAM : la hiérarchie est organisée et construite selon un réseau de descrimination où un nœud non-terminal représente un test et un lien en sortie du nœud un résultat possible pour le test. A tout nœud non-terminal est associé un lien particulier étiquetté "OTHER". Un nœud terminal représente une "image" (les valeurs d'attributs, pas tous) des instances qui ont été triées jusqu'à ce nœud. La classification est effectuée comme suit : le test contenu dans le nœud courant (au départ la racine) est appliqué à la description de l'instance, si le résultat du test est celui d'un des liens associés au nœud, la description est "propagée" au nœud cible du lien, sinon le lien "OTHER" est considéré. Une fois la description de l'instance atteint un nœud terminal, elle est comparée à l'image associée : si cette dernière s'apparie à l'instance (tous les attributs-valeurs de l'image se trouvent dans l'instance), l'image est alors spécialisée en y ajoutant un des attributs spécifiques à l'instance, sinon une spécialisation du réseau de descrimination est effectuée ; l'instance est triée une seconde fois (à partir de la racine) afin de localiser le premier nœud-test vis-à-vis duquel l'instance et l'image diffèrent, si un tel nœud est localisé alors deux branches sont créées (une avec la valeur de l'instance et l'autre avec celle de l'image) sinon l'instance va atteindre de nouveau l'image en question et le processus de classification va créer un nœud-test qui prend la "position" du nœud-image, ce dernier ainsi que le nœud pour l'instance deviennent les fils du nœud-test.

→ UNIMEM : ce système appartient à la famille des systèmes basés sur le modèle de la *mémoire basée sur la généralisation* ([Schank, 82], [Kolodner, 84], [Riesbeck et Schank, 89]). La hiérarchie des concepts dans UNIMEM est une forme étendue du réseau de descrimination; le test sur la valeur d'un attribut est à présent spécifié sur le lien (au lieu d'avoir le nom de l'attribut dans le nœud-test et la valeur sur le lien) et un test correspond à un indexe. Un lien peut avoir plusieurs indexes (et non un seul) et à chaque indexe est associé un coefficient (predictiveness score) qui correspond au nombre de liens ayant cet indexe. Les nœuds contiennent des descriptions de concepts, chaque description est une liste d'attributs-valeurs et à chaque attribut est associé un coefficient (predictability score). A un concept dans la hiérarchie est associé ses instances et la description d'un concept (et aussi d'une instance) contient uniquement ce qui la caractérise (exploitant ainsi le principe d'héritage et de l'économie de stockage). La classification d'une instance est comme suit : l'instance est comparée au nœud courant (au début, la racine) impliquant une mise à jour des "coefficients de prédiction" (predictability score), si les deux descriptions sont "assez" similaires, alors l'instance continue sa descente, via les liens (et non un seul lien) dont les indexes sont vérifiés par l'instance, a des fils du nœud courant. Si aucun des fils du nœud courant ne peut être apparié à l'instance, cette dernière est alors comparée à toutes les instances du nœud courant, si aucune des instances n'est "assez" similaire à la nouvelle instance cette dernière est alors ajoutée comme un nouveau fils du nœud courant. Si une ancienne instance est "assez" similaire à la nouvelle instance alors un nœud G est créé qui contient les éléments en commun et qui prend la position de l'ancienne instance, cette dernière ainsi que le nœud pour la nouvelle instance deviennent les fils du nœud G. Les coefficients des indexes du nœud G sont incrémentés.
Lors de la comparaison de la nouvelle instance avec un nœud courant, si un attribut de l'instance se trouve également dans le nœud courant, le coefficient de l'attribut est incrémenté sinon il est décrémenté. Si le coefficient d'un attribut atteint un seuil minimal (suite à plusieurs décrémentations), l'attribut est enlevé de la description du nœud et si (suite à des enlevements de plusieurs attributs) la description d'un nœud contient "peu" d'attributs alors le nœud est éliminé ainsi que ses descendants. Enfin, si le coefficient attaché à un indexe est "très grand" (il se trouve sur plusieurs liens) alors l'indexe est enlevé des liens.

→ COBWEB : Le système UNIMEM (ainsi que son "frère" CYRUS [Kolodner, 83]) utilise donc des coefficients pour exploiter des notions "probabilistes" comme la fréquence d'occurrence d'un attribut. Cette utilisation ne se base pas toutefois sur un modèle probabiliste, comme dans COBWEB. Pour ce dernier, chaque nœud-concept de la hiérarchie est décrit par une liste d'attributs valeurs, un concept lui est associé une probabilité d'occurrence, à chaque concept est associé tous les attributs des instances du concept, à

chaque attribut est associé toutes les valeurs possibles et à chaque valeur est associée deux poids similaires aux coefficients utilisés dans UNIMEM. Une des différences fondamentales entre les deux systèmes est que COBWEB n'utilise pas d'indexes ; les liens sont de simple "is-a" sans indexes ou autre information supplémentaire. Concernant la classification : à un nœud courant, COBWEB considère chacun des fils et utilise une fonction d'évaluation lui permettant de déterminer le fils qui constitue la meilleure catégorie pour la nouvelle instance (les poids associés au fils choisi sont mis à jour). De façon récursive, les fils du fils choisi sont considérés afin de chercher *une* catégorie plus spécifique pour la nouvelle instance.
Lors de l'évaluation des fils d'un nœud courant et pour optimiser la construction de la hiérarchie, COBWEB considère la possibilité de fusionner (merge) les deux "meilleurs" fils pour la nouvelle instance. L'opération inverse (splitting) est aussi considérée : le fils choisi par la fonction d'évaluation peut être éliminé et ses fils seront attachés au nœud père.

Ces différentes approches sont concernées par la classification *d'instances* et la formation d'une hiérarchie de concepts suite à un flot d'instances. Elles ne considèrent pas la formation d'une base de connaissances qui est une hiérarchie (généralement un graphe) de concepts avec les instances associées aux concepts, mais aussi les schémas qui spécifient les utilisations possibles des concepts. Dans ce contexte, l'information à intégrer ne se limite pas à une instance mais elle peut correspondre à la définition d'un concept ou à la description d'un schéma. L'intégration ne se réduit donc pas à une classification dont le but est de trouver la(les) meilleur(s) catégorie(s) pour une nouvelle instance. Nous devons donc considérer une approche plus "conceptuelle", la prochaine section considère le cas du modèle de la mémoire dynamique de Schank.

Cette remarque sur le processus de formation d'une hiérarchie de concepts à partir d'instances reflètent une différence fondamentale dans la perception de l'apprentissage et la mémoire; en effet, Gennari et al. commencent leur article comme suit [Gennari et al., 89], *"Much of human learning can be viewed as a gradual process of* ***concept formation****. In this view, the agent observes a succession of objects or events from which he induces a hierarchy of concepts that summarize and organize his experience"*. L'apprentissage humain nous semble plus complexe, dûe en partie à la nature sociale de l'humain ; Gennari et al. minimisent cet aspect et leur "agent" ressemble à un "Tarzan" qui vit seul, apprenant uniquement de ce qu'il observe. Un "agent" plus "réaliste" (ou plus humain) vit en communiquant avec d'autres agents et vit dans une culture. De ces "autres sources d'information", il peut apprendre directement ce qu'est un concept (la définition d'un concept), on peut lui "raconter" une situation (particulière ou générale), il peut apprendre que deux termes signifient la même chose, etc. Ce qu'un agent apprend ne se limite donc pas à

des instances mais concerne plutôt des structures conceptuelles formulées à des niveaux d'abstraction variables.
Cette remarque sur "l'apprentissage social" nous semble plus "réaliste" aussi bien pour un enfant (qui apprend de ses parents et de son entourage) que pour un adulte (qui apprend à propos d'un domaine particulier, en lisant des "manuels" ou en apprenant directement d'agents, experts ou non, du domaine).

2.6.2 Mémoire dynamique basée sur la généralisation

Schank [Schank, 82, 86, 88, 91] a montré l'importance d'une *mémoire dynamique*, appelée aussi *mémoire basée sur la généralisation* (MBG) et a montré comment celle-ci permet de rendre compte de plusieurs processus cognitifs comme le rappel (reminding), la recherche d'information, l'apprentissage et la compréhension du langage. En s'inspirant du modèle de la mémoire dynamique, le groupe de Schank ([Lebowitz, 83, 86, 90], [Kolodner, 84], [Kolodner et Riesbeck, 86], [Riesbeck et Schank, 89], [Schank et Leake, 89]) a proposé des programmes qui implantent les principes d'une MBG et qui montrent son utilité dans différents domaines. Les systèmes à base de cas ont été influencés par les travaux du groupe ([Riesbeck et Schank, 89], [Kolodner, 93]).

Le modèle de la mémoire dynamique de Schank est basé d'une part sur un réseau de descrimination qui fournit la structure de base pour indexer les informations en mémoire et d'autre part, sur la généralisation et la spécialisation; deux opérations fondamentales qui ont été proposées par différents chercheurs en psychologie cognitive ([Vygotsky, 62], [Piaget, 68], [Rumelhart et al., 72], [Rumelhart et Norman, 78]). La généralisation fait ressortir ce qui est en commun entre deux descriptions et la spécialisation suscite généralement une subdivision des concepts ; des attributs communs à certains concepts et non à d'autres vont impliquer une subdivision de concept et éventuellement la création de concepts intermédiares [Rumelhart et al., 72].

L'algorithme de base utilisé par le groupe de Schank est illustré par le système UNIMEM présenté ci-dessus. Comme dans le cas des algorithmes de formation probabiliste d'une hiérarchie de concepts, la plupart des algorithmes développés par le groupe de Schank considèrent des descriptions d'instances (des cas) comme information à intégrer, le problème consiste donc à déterminer leurs catégories (ou concepts) et leur rapport avec d'autres instances. Les descriptions sont toutefois plus complexes : elles concernent des actions, des plans, des structures d'explication (explanation pattern) et même des structures d'histoires (story pattern [Schank, 91]).

Dans son modèle de la mémoire, Schank tient compte de la notion "de pertinence" : des "éléments" sont plus importants que d'autres et ils doivent donc influencer et guider l'intégration ainsi que la récupération d'une information de la mémoire. Pour cela, Schank utilise une forme de réseau de descrimination; il considère les éléments pertinents comme des indexes, associés aux liens et jouant le rôle de contraintes (l'information n'est "propagée" via le lien que si elle vérifie les contraintes associées au lien).

Dans chacun de ses ouvrages, Schank [Schank, 82, 86, 88, 91] illustre l'importance en tant qu'indexe d'une "notion" particulière, dans le cadre des situations présentées et traitées dans l'ouvrage; l'indexe peut être de type objet, action, but, plan, thème, cas d'échec, attente non satisfaite, structure d'explication, structure d'histoire, etc. Les indexes peuvent donc être complexes mais le principe de base est que pour un domaine particulier, des éléments sont considérés comme pertinents (importants ou en focus), ils sont fixés au départ et toute nouvelle structure est intégrée en mémoire selon ces éléments. La plupart des systèmes à base de cas suivent cette approche; des indexes sont préfixés au départ selon la nature de l'application [Kettler et al., 94].

Une grande attention a été consacrée à *l'utilisation* d'une telle mémoire, beaucoup plus qu'à la représentation des connaissances et à l'organisation de la mémoire (en particulier, en terme de définition, schéma, instance et autres structures conceptuelles mises en valeur en science cognitive). En effet, dans un modèle général de la mémoire dynamique, les informations à intégrer ne se limitent pas à des descriptions d'instances (des cas), elles peuvent correspondre à des définitions de concepts, à des schémas (qui peuvent être spécifiques et/ou génériques), à des descriptions d'instances et à des déclarations de synonymie.

Aussi, peu d'attention a été consacrée à une formulation détaillée d'un modèle de la mémoire dynamique qui soit indépendant d'un domaine ou d'un processus d'utilisation particulier.

Les remarques de Kettler et al. [Kettler et al., 94] sont appropriées à ce stade : parmi les problèmes associés à la préindexation ils citent que "indexing schemes are often domain- and task-specific and can thus limit the general utility of a case memory". Ils notent également que la préindexation reduit la flexibilité (hinders flexibility); un cas peut être retrouvé de façon efficace seulement à partir des attributs selon les quels il a été indexé, si un cas en mémoire partage avec le nouveau problème seulement des attributs non-indexés alors le cas ne sera pas recupéré (retrieved). Enfin, les auteurs notent que "psychological plausibility of indexing is questionable in light of psychological studies of human analogy retrieval"; cette dernière est rapide et hautement associative et plusieurs rappels sont générés.

Comme alternative, Kettler et al. utilisent, dans leur système pour la planification basée cas, une grande mémoire non-indexée (ce type de mémoire est à la base du "raisonnement basé mémoire" [Waltz, 90]). Elle correspond à un réseau sémantique "classique" comprenant des "connaissances conceptuelles" (description de concepts et d'instances, organisés selon des liens comme "is-a" et "part/whole") et des "connaissances épisodiques" qui correspondent aux

situations de planification traitées par le système, à chaque épisode (case) est associée une date, un temps, une place, un but, un plan et autres attributs. Devant un nouveau problème, le système, appelé Caper, cherche un(des) plan(s) de la mémoire selon certains éléments pertinents comme des buts et autres attributs de la situation courante, il fusionne les plans récupérés et considère les cas d'interaction qui peuvent résulter d'une telle fusion. La formation dynamique de la mémoire n'est toutefois pas adressée par Kettler et al. Comme le note ces derniers, elle fait partie des travaux futurs; "we are investigating the storage of adaptation episodes in memory" [Kettler et al., 94].

En résumé, ces approches (exceptée le système Caper qui est un système de planification et non de formation de concepts) sont concernées par la formation d'une hiérarchie de concepts à partir d'un flot d'instances (ou cas). Elles ne sont pas concernées par la formation d'une base de connaissances (mémoire) à partir d'un flot de structures conceptuelles comme des définitions, des schémas et des instances. Parmi les questions soulevées par cette problématique, citons : quelle est la représentation des connaissances à utiliser ?, quelle est l'organisation de la mémoire ? quel(s) est(sont) le(s) rapport(s) entre ces structures conceptuelles, comment s'effectue l'intégration en mémoire de ces structures ? et, comment de nouvelles structures sont induites ?
Des éléments incorporés dans les approches introduites dans cette section sont toutefois à retenir : formation incrémentale et non-supervisée de la mémoire (ce qui est plus "réaliste"), importance d'une mémoire basée sur la généralisation et importance de la notion d'éléments pertinents (comme illustrée par Schank) sans toutefois les réaliser avec une préindexation (selon un réseau de descrimination) ; la mémoire devrait être non-indexée (comme le suggère Kettler et al. [Kettler et al., 94] et aussi Waltz [Waltz, 90]).
Nous voulons développer, dans le cadre de la théorie des GC, un modèle de formation incrémentale d'une mémoire dynamique qui répond aux besoins ci-dessus (intégration de structures conceptuelles et non seulement d'instances) et qui tient compte des "éléments à retenir" précités.

2.7 Résumé

Nous avons considéré des travaux sur les réseaux sémantiques selon deux axes de recherche : opérations et langages logiques pour les réseaux sémantiques et formulation des connaissances procédurales et leur activation avec les réseaux sémantiques. Pour le premier axe nous avons considéré plus particulièrement le travail du groupe de Aït-Kaci et pour le second axe, le travail du groupe de Rumelhart et Norman (le groupe des années 70 !). Nous avons introduit brièvement des formalismes basés activation de graphes, en particulier les

réseaux de "frames", les graphes de flot de données, les réseaux de Pétri, les diagrammes de transition d'états et les objets actifs.

Puisque notre proposition d'un système multi-paradigme de représentation et manipulation des connaissances est effectuée dans le cadre de la théorie des GC, nous avons considéré plus en détail cette théorie et les systèmes qui l'ont adopté. Nous y avons identifié les limitations (ou plus précisement absences) suivantes :

→ L'absence d'une algèbre pour les GC composés avec liens de co-références, qui offre un grand nombre d'opérations en précisant les rapports entre ces opérations; en les organisant par exemple selon une hiérarchie d'opérations.

→ L'absence d'un "Prolog pour les GC" intégrant les extensions conceptuelle, contextuelle et orientée objet sans masquer le langage Prolog.

→ L'absence d'un mécanisme d'activation des GC. Contrairement à d'autres formalismes orientés graphe (comme les graphes de tâches, les graphes de flot de données, les réseaux de Pétri et les diagrammes de transition), les GC sont "statiques", aucun mécanisme d'activation des GC n'a été proposé. Un tel mécanisme serait pourtant très approprié car il permettrait non seulement de représenter des connaissances procédurales en GC (exercice commun à d'autres formes de réseaux sémantiques) mais de les simuler, au niveau du graphe. De plus, une telle extension dynamique des GC permettrait d'intégrer des modèles de traitements, comme le modèle procédural, le modèle fonctionnel et le modèle orienté objet.

→ L'absence d'un modèle de formation incrémentale d'une base de connaissance selon la théorie des GC. Cette dernière correspond à la hiérarchie des types avec les structures conceptuelles (définition, instance, canon et schéma). Malgré l'importance d'un tel modèle pour les bases de connaissances et surtout pour les processus cognitifs (compréhension du langage, planification et comportement, raisonnement et apprentissage), aucune étude n'a été envisagée dans cette direction.

Le système que nous voulons concevoir et développer tente de combler ces "absences" dans la communauté des GC. Le chapitre 3 présente un langage parallèle et multi-paradigme basé sur l'activation des GC, les chapitres 4 et 5 illustrent l'utilisation du langage selon différents modèles de programmation. Le chapitre 6 présente notre modèle de la mémoire dynamique (organisée comme une base de connaissance selon la théorie des GC) qui correspond aussi à la "mémoire à long terme" dans une application Synergy (le langage parallèle). Le chapitre 7 présente un "Prolog pour les GC" avec une extension conceptuelle, contextuelle et orientée objet et ce, sans masquer le langage Prolog. Enfin, l'annexe 1 présente une algèbre pour les GC composés avec liens de co-références, l'algèbre offre un grand nombre d'opérations organisées selon une hiérarchie d'opérations. L'algèbre est incorporée dans les deux langages du système, rendant son utilisation plus appropriée.

Chapitre 3

Synergy : un langage parallèle et multi-paradigme, basé sur l'activation des GC

Nous introduisons dans ce chapitre les éléments de base de Synergy, un langage graphique, parallèle et multi-paradigme basé sur l'activation des GC. Le langage intègre différents modèles de programmation, en l'occurrence le modèle procédural, le modèle fonctionnel et le modèle orienté-objet. D'autres modèles peuvent également être réalisés en Synergy.
Le caractère actif des GC résulte de l'hypothèse qu'une telle structure peut représenter un contexte de traitement; il résulte en particulier de l'interprétation d'un concept du GC comme une entité active ayant son propre cycle de vie et aussi, de la capacité de certaines relations à propager l'activation dans le graphe.

Après un aperçu sur le langage, nous présentons son environnement conceptuel, les "GC de Synergy", les mécanismes d'abstraction du langage, le mécanisme d'activation des GC, la boucle principale de l'interpréteur, la "syntaxe" et l'interface graphique du langage.

3.1 Aperçu sur le langage Synergy

Synergy est un langage basé sur l'activation des GC ; les concepts du GC peuvent être actifs et certaines relations peuvent propager l'activation dans le graphe, ainsi :

→ Un concept peut avoir un type primitif ou défini (par un GC) et le type peut correspondre à une entité "activable" (par exemple, une opération ou un processus) ou "statique" (par exemple, un entier ou un booléen). Un concept peut être traité comme une variable au sens fonctionnel (une seule assignation de valeur est possible), une variable au sens procédural (possibilité d'assignation multiples) ou comme un objet au sens orienté-objet (avec des attributs, des méthodes et possibilité d'héritage).

→ Une entité activable peut avoir des arguments en entrée et/ou en sortie, les arguments sont alors liés à l'entité par des relations de données (*in*, *out*) et l'activation de l'entité peut dépendre de la présence/absence des valeurs de ses arguments (nous avons

alors une *dépendance de données*). L'activation d'une entité E1 peut dépendre aussi d'une condition booléenne ou d'une entité E2 quelconque (pour ce dernier cas, l'activation de E2 est un prérequis ou une pré-condition pour l'activation de E1). La condition (ou l'entité E2) est alors liée à l'entité E1 par la relation "guard" (*grd*) et nous avons dans ce cas une *dépendance de contrôle*. Enfin, alors que l'activation est parallèle par défaut, on peut spécifier l'activation d'une séquence d'entités, reliées par la relation "next" (*next*), ce qui constitue une autre forme de dépendance de contrôle.

→ L'activation en Synergy intègre et combine l'activation par flot de données et/ou flots de demande (comme dans la programmation fonctionnelle), l'activation par flot de contrôle (comme dans la programmation impérative/procédurale) et l'activation par "cycle de vie" d'objets actifs (comme dans la programmation concurrente orientée objet). Synergy offre différents moyens pour contrôler ces flots et aussi pour influencer le cycle de vie des objets.

Synergy utilise cette dimension dynamique des GC pour produire une intégration de différents modèles de programmation (en l'occurrence le modèle procédural, le modèle fonctionnel et le modèle orienté-objet). Le caractère multi-paradigme du langage est renforcé par la possibilité que différents modèles de traitement, à part les trois modèles ci-dessus, peuvent être "réalisés" en Synergy. Citons par exemple les modèles orientés diagrammes de transition d'états, "frame", événements et agents. La réalisation en Synergy d'un "programme" écrit selon un modèle de traitement particulier consiste en général à reformuler le programme en rendant explicite la sémantique et l'interprétation sous-jacente. Cette sémantique, qui correspond au modèle de traitement adopté, peut être définie en Synergy; en termes de types de concepts que le programmeur peut utiliser comme des types "primitifs" (offerts par le langage). Les chapitres 4 et 5 illustrent cet aspect du langage.

3.2 L'environnement conceptuel de Synergy

Définition : *Une application Synergy*
Une application Synergy est composée d'une *mémoire à long-terme* (MLT) et d'une *mémoire de travail* (MT). La première représente la base qui contient la connaissance sur un domaine (ou sur des domaines reliés), la MT fournit un espace de travail où l'utilisateur peut spécifier ses requêtes. ♦

3.2.1 Memoire à long terme

Comme dans la programmation orientée objet, la mémoire à long terme est représentée par un graphe de généralisation. Quand une nouvelle application est créée, Synergy fournit son graphe de généralisation de base qui contient essentiellement les types primitifs (qui ne sont pas définis en Synergy) et les types prédéfinis (types définis en Synergy par le concepteur du langage et offerts aux utilisateurs comme des types "primitifs").
La Figure 3.1 montre le graphe de généralisation de base de Synergy. La spécification des types primitifs est fournie par la suite.

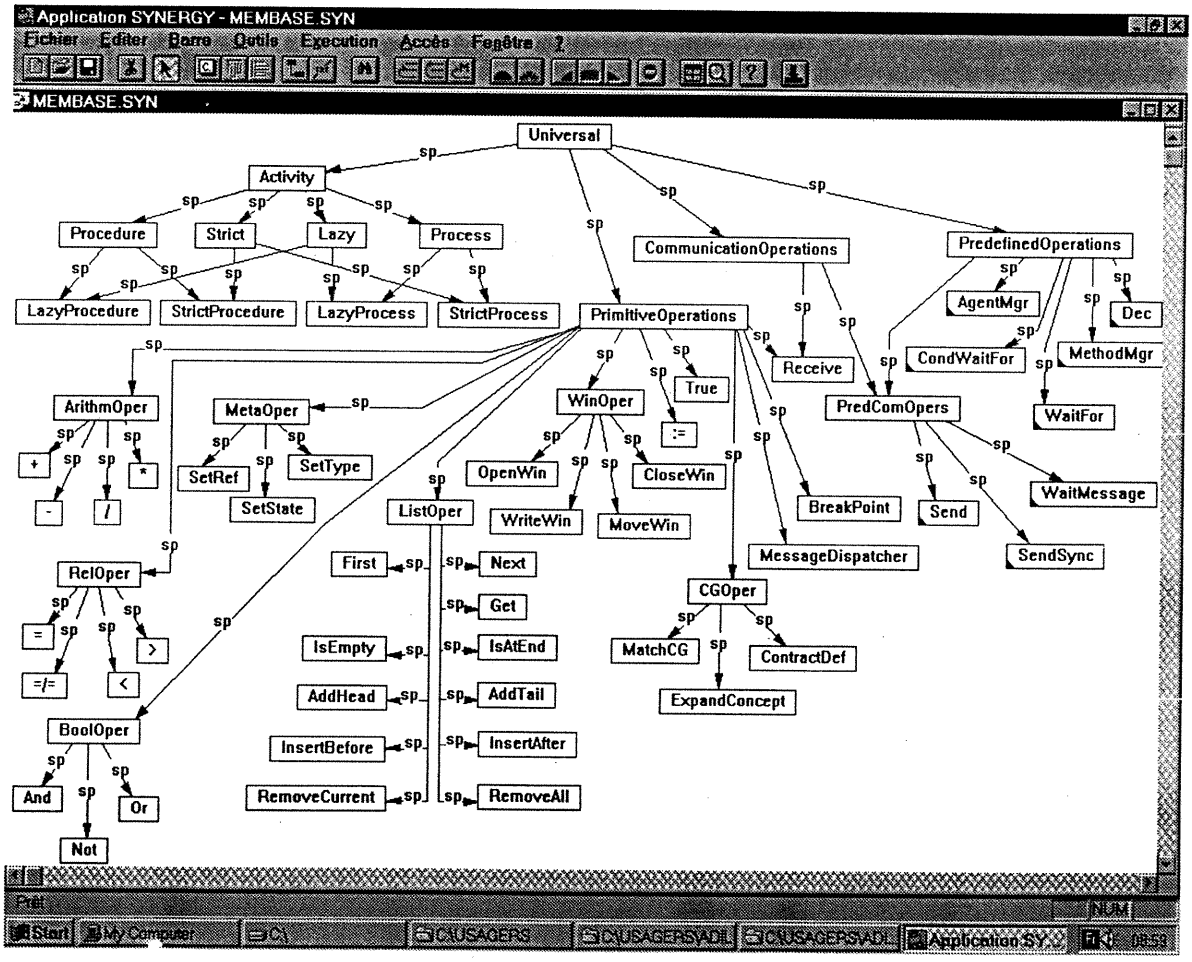


Figure 3.1: Graphe de généralisation de base de Synergy

L'utilisateur peut définir un nouveau type comme une spécialisation des types existants (qu'ils soient primitifs, prédéfinis ou définis). La Figure 3.2a donne un exemple de définition : le type IsAdult est une procédure avec une évaluation stricte. Le lien "sp" représente le lien de spécialisation.

L'utilisateur peut spécifier aussi des instances d'un type, en fournissant une description spécifique de l'instance (comme pour MyTable dans la Figure 3.2b) ou bien en spécifiant seulement que l'individu est une instance du type (comme pour Tble34). Pour ce dernier cas, Synergy va créer, au besoin, la description de l'instance par instanciation de la définition de son type. Enfin, l'utilisateur peut spécifier un schéma, décrivant une situation générale ou particulière. La Figure 3.2c donne un exemple de schéma. La définition d'un type, la description d'une instance ou d'un schéma sont introduits plus en détail dans la section 3.4.

Note: Certains exemples et figures sont extraits, sans traduction, du manuel de Synergy (rédigé en anglais) [Kabbaj, 96].

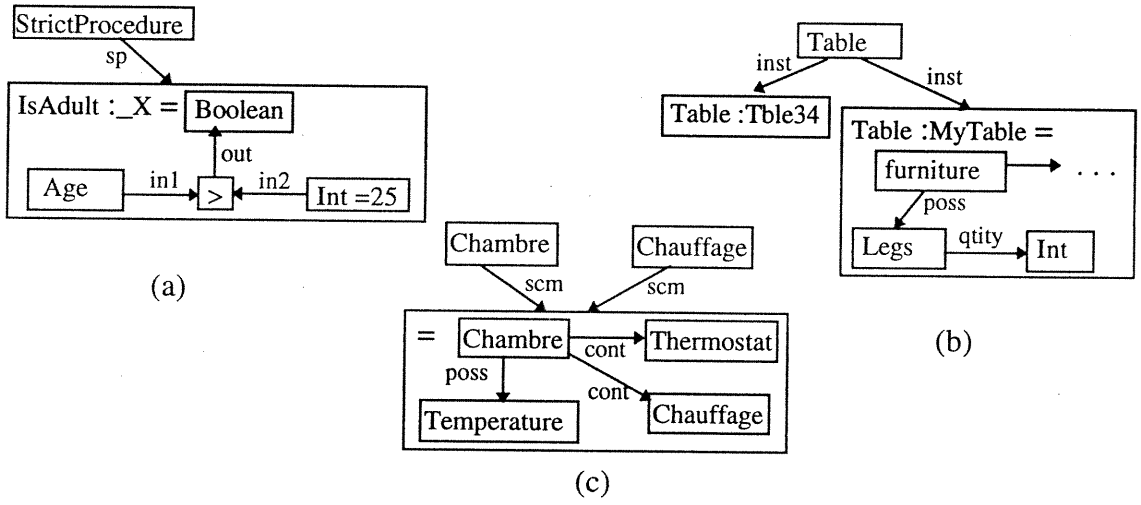


Figure 3.2: Définition d'un type, descriptions d'instances et d'un schéma

Les opérations primitives de Synergy

Les types de données primitifs de Synergy sont Number, String, Boolean, List, Window et CG. Parmi les types d'opérations primitives, Synergy offre l'affectation (:=), les opérations arithmétique, relationelle, booléenne, opérations d'entrée/sortie, opérations sur les listes, opérations sur les GC et des "méta-operations". La Figure 3.3 fournit une formulation graphique de la signature des opérations arithmétiques et booléennes.

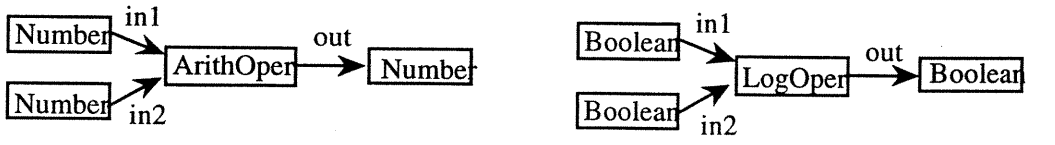


Figure 3.3: Signature des opérations arithmétiques et booléennes

Signatures des autres opérations (extrait du manuel de Synergy) :

- Not(**in** Boolean; **out** Boolean) ;

- RelOper(**in** AtomicData, AtomicData; **out** Boolean) ; *RelOper can be =, =/=, < or >, and AtomicData is a Number, a String or a Boolean.*

- True ; *operation with no effect. it is similar to "Continue" in Fortran !*

- IsBind(**in** Concept; **out** Boolean) ; return true if the concept has a value

IsFree(**in** Concept; **out** Boolean) ; return true if the concept has not a value

- *Time operations*

WaitFor(**in** Integer) ; *Wait will remain in activation for the given period, defined in Synergy-time*

CondWaitFor(**in** Integer, Boolean) ; *While the boolean is true, CondWaitFor will remain in activation for the given period. If the boolean becomes false, the operation will stop even if the waiting period is not terminated*

- *I/O and Window operations :*

OpenWindow(**in** String; **out** Window) ; *open a window with a title*

MoveWindow(**in** String, **out** Window) ; CloseWindow(**in** Window) ;

WriteWindow(**in** Window, AtomicData);

ReadWindow(**in** Window; **out** AtomicData) ; */** not implemented yet **/*

- *List operations :*

List operations in Synergy are similar to those provided by Visual C++

First(**out** List) ; Next(**out** List) ; IsEmpty(**in** List; **out** Boolean) ; IsAtEnd(**in** List; **out** Boolean) ;

InsOper(**in** Data; **out** List) ; *InsOper can be AddHead, AddTail, InsertBefore and InsertAfter ;*

Get(**in** List; **out** Data) ; RemoveOper(**out** List) ; *RemoveOper can be RemoveCurrent or RemoveAll.*

- *meta-operations :*

SetState(**in** State, Referent) ; *changes the state of the concept with Referent to State ;*

SetRef(**in** Referent, Referent) ; SetType(**in** Type, Referent) ;

GetState(**in** Referent; **out** State) ; */*** the three Get are not implemented yet ***/*

GetRef(**in** Referent; **out** Referent) ; GetType(**in** Referent; **out** Type) ;

- *Type hierarchy operations*

SuperTypes(**in** Type; **out** LTypes): *superTypes* gives all super types of a given type.

SubTypes(**in** Type; **out** LTypes) : *subTypes* gives all subtypes of a given type.

IsSuperType(**in** Type, Type; **out** Boolean) : *IsSuperType* checks if the first type is a super type of the second.

IsSubType(**in** Type, Type; **out** Boolean) : *IsSubType* checks if the first type is a subtype of the second.

NComSuperType(**in** Type, Type; **out** Type) : *NComSuperType* computes the nearst common super type of type of the two input types.

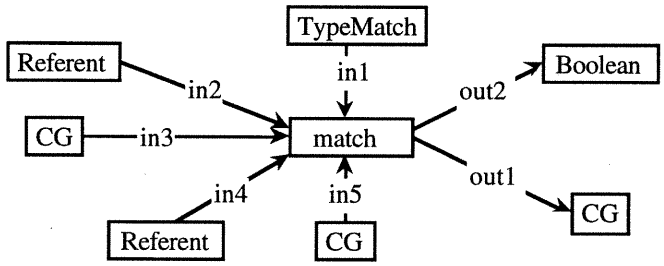
NComSubType(**in** Type, Type; **out** Type) : *NComSubType* computes the nearst common subtype of the two input types.

IsNComSuperType(**in** Type, Type, Type; **out** Boolean) : *IsNComSuperType* checks if the third type is the neart common super type of the first two input types.

IsNComSubType(**in** Type, Type, Type; **out** Boolean) : *IsNComSubType* checks if the third type is the neart common subtype of the first two input types.

Les opérations sur les GC sont définies comme des spécialisations de l'opération générique "match" :

match(**in** TypeMatch; **inout** E1: Referent, G1:CG, E2: Referent, G2: CG, G3: CG, BRes: Boolean)



Le paramètre TypeMatch est défini sur l'ensemble suivant : {match, project, subsume, unify, maximalJoin, funcMaximalJoin, specialize, generalize, funcGeneralize, partialContrat, cplteContract}.

Les paramètres E1 et E2 peuvent être soit des référents de concepts dans les deux graphes respectivement, soit une chaîne vide " " dans le cas où ils ne sont pas utilisés. S'ils sont spécifiés, E1 et E2 correspondent aux points d'entrée pour les graphes G1 et G2.
La définition de "match" et les opérations dérivées sont fournies dans l'annexe A. Il en est de même pour deux autres opérations sur les GC :

- contractDef(**in** T: Type, R: Referent, G:CG; **out** B:Boolean) : *contract a definition of a type T from G, focusing on the concept in G with referent R.*
- expandDef(**in** T: Type, R: Referent, G:CG; **out** B:Boolean) : *specialize G by the definition of the type T, focusing on the concept in G with referent R.*

3.2.2 Mémoire de travail

La mémoire de travail fournit un espace de travail où l'utilisateur peut spécifier ses requêtes, représentées par des GC et initier ensuite leur exécution (éventuellement en parallèle).

3.3 GC de Synergy

Les GC constituent la structure de base dans Synergy : la mémoire à long-terme, les requêtes dans la mémoire de travail, la définition d'un type et la description d'une instance ou d'un schéma sont toutes représentées en GC.

Définition : *GC de Synergy*
Un GC dans Synergy est composé de *concepts* connectés par des *relations dyadiques*. Un concept spécifie un *référent* avec son *type*, sa *valeur* (ou description) et son *état*. Si un champ du concept est non spécifié, une valeur par défaut est alors considérée. ♦

Remarque : Puisque les relations de rang supérieur à 2 sont rarement utilisées, qu'elles peuvent être reformulées en relations dyadiques et qu'elles rendent "l'implantation" plus lourde (pour un gain minime), nous considérons uniquement les relations dyadiques.

Cette section définit notre version des GC, son caractère dynamique est défini dans la section 3.5. La section 3.6 présente la "syntaxe" de Synergy et résume la description des éléments de base du langage.

Détails sur les composantes d'un concept

Définition : *Type d'un concept*
Le type d'un concept peut être primitif, prédéfini ou défini. Le type par défaut est "Universal". ♦

Définition : *Référent d'un concept*
Le référent d'un concept peut être un identificateur d'instance (ou d'objet), une variable (qui peut avoir un référent comme valeur) ou une *co-référence* , une référence à un autre concept. Le référent par défaut est la chaîne vide " ".
Un identificateur d'instance est une chaîne de caractères ne débutant pas par "_". Un identificateur de variable est toute chaîne de caractères qui commence par "_". ♦

Définition : *Co-référent*
Un *co-référent* est de la forme "Identificateur." ou " "Identificateur1.Identificateur2. ... .IdentificateurN", où "IdentificateurJ" est un identificateur d'instance ou de variable. ♦

La seconde forme de co-référence indique le chemin "à suivre", via des GC imbriqués, pour atteindre (ou déterminer) le concept référé.

Exemple :

Le concept [Boolean :Daniel.eyes.canOpen] est une co-référence à un concept ayant "canOpen" comme référent, qui est contenu dans la description d'un autre concept dont le référent est "eyes". Ce dernier concept est lui-même contenu dans la description d'un concept dont le référent est "Daniel" (Figure 3.4).

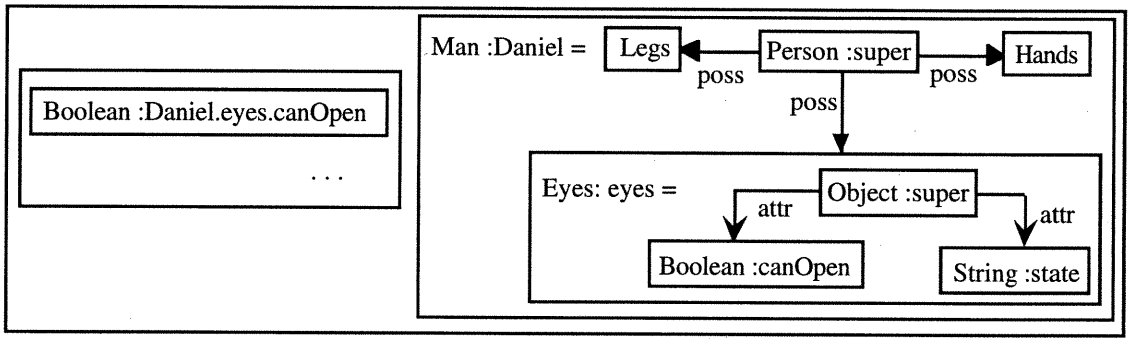


Figure 3.4: Co-référence à un concept

Notons que la valeur d'un concept avec co-référence est généralement celle du concept référé, le champ "Valeur" du premier concept contient en fait un "pointeur" au second concept.

Définition : *Valeur d'un concept*

La valeur d'un concept peut être un nombre, un booléen, une chaîne de caractères, une liste, une fenêtre ou un GC (ou une collection de GCs considérés comme un GC composé). ♦

Définition : *Cycle de vie et état d'un concept*

Tout concept possède un *cycle de vie* défini sur les *états* possibles du concept. ♦

La description du cycle de vie est présentée dans la prochaine section (nous devons introduire d'autres notions préalables au cycle de vie).

Remarques (sur les composantes d'un concept) :

→ Dans la formulation linéaire d'un concept, nous utilisons la notation suivante :

[Type :Referent =Valeur #Etat].

→ La définition d'un type et la description d'une instance ou d'un schéma sont formulées elles-mêmes comme des concepts (Figure 3.2). En plus, la mémoire à long-terme et la mémoire de travail sont considérées aussi comme des concepts avec des GC comme valeurs.

→ Pour avoir une formulation plus concise, nous utilisons dans ce chapitre des expressions comme "concept primitif" (le type du concept est primitif), "concept défini" (le type du concept est défini), "concept paresseux" (le type du concept est un sous-type du type LazyActivity), "concept co-référent" (un concept dont le référent est une co-référence), "concept actif" (un concept à l'état "en-activation"), etc., nous attribuons à un concept un qualificatif qui porte sur un de ses champs.

→ La co-référence dans la théorie des GC correspond en Synergy au co-référent avec la forme "Identificateur.". Ainsi, la co-référence entre [Integer : *_X] et [Integer : ?_X] est formulée dans Synergy comme suit [Integer : _X] et [Integer : _X.] . Dans la théorie des GC, le concept [Integer : *_X] est le nœud définition (defining node) pour la variable _X alors que [Integer : ?_X] est une référence au nœud définition de _X [Sowa, 93, 95].
La seconde forme de co-référence fournie par Synergy, n'a pas d'équivalent dans la théorie des GC. La Figure 3.5 illustre la nature "dynamique" de cette forme de co-référence : nous spécifions que le concept [Boolean :Daniel.eyes.canOpen] est une référence à un autre concept, qui peut être présent ou non au temps de spécification (dans la Figure 3.5, il ne l'est pas).

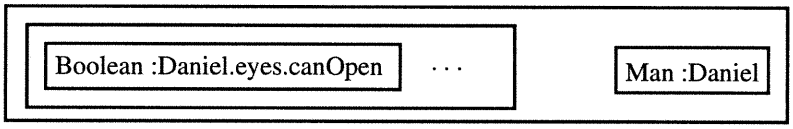


Figure 3.5: Co-référence à établir

Au moment de l'exécution, la co-référence va guider la localisation ou la création du concept référé : pour notre exemple, la co-référence [Boolean :Daniel.eyes.canOpen] spécifie qu'on doit localiser le concept qui définit le référent "Daniel", puis déterminer la description de "Daniel" (par instanciation si possible) ainsi que la description de "eyes" qui se trouve dans la description de Daniel et ensuite, localiser le concept pour "canOpen".
La résolution de co-référence transforme ainsi le GC de la Figure 3.5 en un autre GC, celui de la Figure 3.4.

→ La co-référence entre le concept référé C1 et le concept-référent C2 constitue une relation procédurale "implicite et virtuelle" : C1 --coref--> C2. La relation est implicite car elle n'est pas présente dans le graphe, elle est indirectement spécifiée par la co-référence et établie lors de la résolution de la co-référence. Nous considérons la co-référence comme une relation car elle peut affecter l'activation d'un GC, comme toute autre relation procédurale (définie par la suite). Nous détaillons ce point dans la prochaine section.

→ Dans la théorie des GC, le concept est composé du type et du référent qui peut désigner un individu particulier comme "Marc2" ; [Man : Marc2]. L'individu peut être décrit par la structure conceptuelle *individual* : **individual** Man (Marc2) **is** CGQuiDecritMarc2 . Pour le concept [Integer : 3], le référent "3" désigne un individu (la valeur 3 !) qui n'est pas décrit par individual (c'est une primitive). En pourrait supposer que toute opération qui s'interésserait à la description du référent "3" tiendrait ce dernier pour la valeur 3. La situation est plus "complexe" si on veut représenter un concept dont le référent est NbreVoiture, le type est Integer et la "valeur/description" est 3.
Dans notre formulation, la description de l'individu Marc2 est spécifiée directement dans le concept : [Man : Marc2 =CGQuiDecritMarc2].
La composition d'un concept en : type, référent, valeur et état permet une formulation simple de la notion d'objet en programmations procédurale et orientée objet (l'état est important surtout en programmation concurrente orientée objet). Dans les deux formes de programmation, un objet possède un type, un référent (qui désigne et permet d'identifier et de référer à l'objet) et une valeur (ou une description). Par exemple, la déclaration "Int NbreVoiture := 3" sera formulée dans Synergy comme un concept : [Int :NbreVoiture =3], NbreVoiture désigne un objet de type Int et de valeur 3. La valeur de NbreVoiture peut changer par la suite.

Détails sur les relations

Synergy fournit un ensemble fini de *relations procédurales.* L'ensemble est "spécial" dans la mesure où chacune de ces relations possède une sémantique procédurale et peut donc avoir un effet sur l'activation d'un GC (comme le montre la prochaine section).
Outre cet ensemble, le programmeur peut utiliser n'importe quel identificateur comme nom de relation, la relation sera toutefois sans effet procédural sur l'activation des concepts du(des) GC concerné(s).

Définition : *Relations procédurales*
L'ensemble des relations procédurales est subdivisé en : 1) relations de donnée (**in/out**) qui connectent les concepts-arguments au concept-activité (concept qui représente une fonction ou une procédure par exemple), 2) relations de contrôle (*guard* "**grd**" et *succession* "**next**"), 3) relations concernant la mémoire (*specialisationDe* "**sp**", *instanceDe* "**inst**", *schemaPour* "**scm**" et *synonymeDe* "**synm**") et 4) la relation implicite de co-référence "**coref**" qui existe "implicitement" entre un concept référé C1 et un concept-référent C2 (la co-référence peut avoir un effet sur l'activation d'un GC, comme toute autre relation procédurale). ♦

Définition : *Les relations in et out*

Les relations de donnée sont **in** et **out**, elles spécifient les arguments en entrée/sortie d'un concept-activité. Le lien C1 -in-> C2 indique que le concept C1 est un argument en entrée du concept C2 et le lien C3 -out-> C4 indique que le concept C4 est un argument en sortie de C3. ♦

Définition : *La relation grd*

La relation "guard" **grd** entre un concept C1 et un concept C2 ; C1 -grd-> C2, indique une dépendance de contrôle de C2 envers C1. C1 est un "guard-condition" pour C2 : la valeur de C2 ne peut être déterminée et activée que si la valeur de C1 est déterminée, activée et qu'après son activation la valeur de C1 soit différente de "false". ♦

Par défaut, nous adoptons l'*interprétation procédurale* des relations de donnée et de la relation "grd" : dans C1 -in-> C2, l'accès de C2 à la valeur de C1 n'implique pas une consommation de la valeur de C1. Aussi, dans C3 -out-> C4, C3 peut assigner une nouvelle valeur à C4 même s'il en possède déjà une. De même, dans Ca -grd-> Cb, la valeur de Ca n'est pas consommée suite à l'activation de Cb.

L'*interprétation fonctionnelle* est toutefois adoptée si le programmeur spécifie l'attribut optionnel *"fonctionnelle"*, représenté par "**f**".

Définition : *L'attribut fonctionnel f*

→ Le lien -grd,f-> dans C1 -grd,f-> C2 indique que la valeur de C1 sera consommée juste avant l'activation du concept C2. Il en est de même pour C1 -in,f-> C2 si le type de C2 est primitif.

→ Le lien -out,f-> dans C3 -out,f-> C4 indique que C3 ne peut assigner une nouvelle valeur à C4 que si ce dernier n'en a pas, dans le cas contraire, C3 devra attendre que C4 devienne sans valeur. ♦

Remarques :

→ Comme le note Treleaven et al. [Treleaven et al., 82], les deux interprétations, procédurale et fonctionnelle sont nécessaires si une grande variété d'applications est à considérer. C'est effectivement un des objectifs du langage Synergy.

→ L'attribut optionnel "f" modifie la sémantique de base des relations "in", "out" et "grd". Deux autres attributs optionnels sont introduits dans la prochaine section.

Définition : *La relation next*

La relation **next** entre un concept C1 et un concept C2 ; C1 -next-> C2 spécifie qu'après l'activation de C1, C2 sera déclenché. ♦

Notons la différence entre les relations grd et next : dans C1 -next-> C2 , C1 ne constitue pas une condition pour C2, contrairement à "grd".

Les relations de la mémoire, **sp** *(spécialisation),* **inst** *(instanciation),* **scm** *(schémaPour)* et **synm** *(synonymeDe)* interviennent dans la description de la mémoire à long-terme. Cette dernière est un GC composé de concepts (qui représentent des définitions de types, des synonymes de types, des descriptions d'instances et de schémas) reliés par les relations de la mémoire.

Définition : *Les relations de la mémoire* ***sp, inst, scm et synm***

sp : DecType -sp-> DecType, Schema -sp-> Schema , ou Schema -sp-> DecType.
DecType représente la définition d'un type ou la déclaration d'un type primitif.
C1 -sp-> C2 est lue comme suit : " La description de C2 est une spécialisation de la description de C1".

inst : DecType -inst-> Instance. "Instance est une instance du type déclaré par DecType".

scm : DecType -scm-> Schema, ou Instance -scm-> Schema.
C1 -scm-> C2 est lue comme suit : "C2 est un schéma pour C1".

synm : DecType -synm->[= LTypes]. "Les types dans la liste LTypes sont des synonymes du type déclaré par DecType". ♦

Remarques :

- Les synonymes représentent des *alias* pour le type ; un synonyme peut être utilisé à la place du type.
- Pour la relation "coref", nous avons déjà noté que sa formulation se réduit au co-référent d'un concept et qu'elle n'est qu'implicite.

3.4 Mécanismes d'abstraction dans Synergy

L'abstraction est présente dans Synergy via la définition de types, la description d'instances et de schémas, l'héritage et le mécanisme de co-référence. Enfin, Synergy fournit au programmeur la possibilité d'une formulation abstraite d'un concept-fonction (un concept qui représente une fonction).
Cette section présente ces différents aspects de l'abstraction en Synergy.

3.4.1 Descriptions d'une instance et d'un schéma

Définition : *Description d'une instance*
La description d'une instance peut se faire selon deux formes possibles : [Type :NomInstance] ou [Type :NomInstance =DescrInst] , où "NomInstance" est un identificateur d'instance et "DescrInst" un GC ou une collection de GCs. Une instance peut être décrite dans la MLT (auquel cas le programmeur devrait la relier, avec le lien "inst", au nœud qui définit son type) ou dans un autre contexte. ♦

La forme [Type :NomInstance] spécifie seulement qu'un individu est une instance d'un type alors que la seconde fournit de plus la description de l'individu. Si une instance est spécifiée par la première forme et si son type est défini, alors la description (ou valeur) de l'instance est déterminée par une instanciation de la définition du type ; la description serait une copie du corps de la définition du type. Si un GC g contient le concept [Chat :Marc] alors cette spécification de Marc est *locale* à g; dans un autre graphe (un autre contexte) on peut avoir le concept [Homme :Marc]. La résolution de la référence "Marc." dépendra de la position, dans la "hiérarchie des contextes", du contexte où se trouve cette référence. La notion de *hiérarchie des contextes* ainsi que la résolution de la co-référence sont considérées par la suite.
Notons que la résolution de références selon la hiérarchie des contextes est commune à plusieurs langages de programmation (notamment, Pascal).

Définition : *Description d'un schéma*
La description d'un schéma a la forme générale suivante : [:IdentSchm =DescrSchema]. Le référent IdentSchm est optionnel, mais il est utile pour référer au schéma et à son contenu. ♦

Exemple : Le schéma suivant est indéxé dans la MLT sous quatre types.

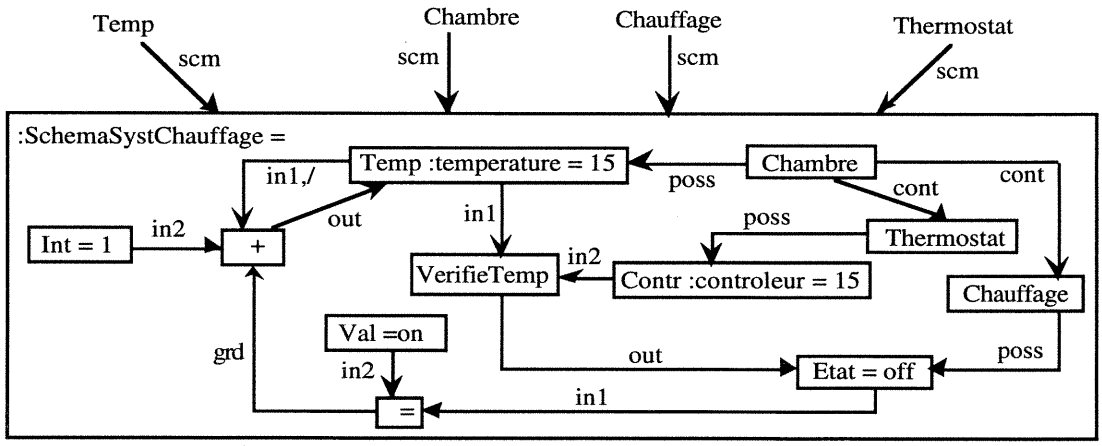


Figure 3.6 : Le schéma pour le système de chauffage d'une chambre

La notion de "droit d'accès à l'information", distinguant ce qui est "public" (accessible aux autres contextes) de ce qui est "privé" au contexte, est utilisée dans la formulation des types de données abstraits ("abstract data types"). En Synergy, la notion "d'information publique" peut être réalisée en assignant des référents à certains concepts du contexte en question (par exemple les référents "temperature" et "controleur" dans le schéma ci-dessus). On peut ensuite y accéder par référence (bien sûr, les référents ainsi que le référent du contexte doivent être connus par les autres contextes), par exemple : SchemaSystChauffage.temperature .
Les autres concepts qui n'ont pas de référent (par exemple [Int = 1] qui représente le taux d'augmentation de la température) ne peuvent être référés et constituent donc une "information privée" au contexte en question.

3.4.2 Définition d'un type de concept

Un nouveau type T est défini comme une spécialisation des types existants (qu'ils soient primitifs, prédéfinis ou définis).

Définition : *Définition d'un type de concept*
Un type de concept est défini suit : [NouvType :ParametreFormel =CorpsDef] ; où "ParametreFormel" est une variable et "CorpsDef" un GC ou une collection de GCs.
La définition doit être ajoutée à la mémoire à long-terme et reliée, avec le lien "sp", aux super-types du nouveau type. ♦

Remarque : Un type défini peut avoir plusieurs super-types directs mais il peut avoir au plus un *concept-super* (un concept avec le mot clé "super" comme référent) dans le corps de sa définition. Comme le montre la prochaine section, l'héritage dépend de la présence du concept-super. Dans la définition actuelle de Synergy, nous considérons uniquement *l'héritage simple* (un concept peut avoir un seul concept-super). Le but immédiat étant d'expérimenter la notion d'héritage dans le cadre de Synergy. Une fois ce but atteint, l'intégration de *l'héritage multiple* sera effectuée en utilisant le même principe qu'avec l'héritage simple. En particulier, une des options que nous envisagons pour cette intégration est la suivante : un concept peut avoir plusieurs concept-supers, avec des référents "superN" où N représente un entier qui indiquera la priorité du concept-super par rapport aux autres, cette information sera utilisée par le mécanisme d'héritage pour fixer sa stratégie de recherche des super-concepts.

Les deux prochaines définitions concernent les concepts "paramétrés" ; concepts qui représentent des fonctions, procédures ou autres activités avec arguments/paramètres.

Définition : *Paramètres*
Les paramètres d'un type défini (s'il en a) sont des co-références aux arguments correspondants. Les paramètres sont représentés dans le corps de la définition par des concepts avec des référents préfixés par **in/** ou **out/** : [Type :in/ParamId] ou [Type :out/ParamId]. Si le *rang* d'un paramètre doit être spécifié, il faut utiliser la forme **inN/** ou **outN/** où N représente un entier. ♦

Note : Si l'argument correspondant à un paramètre est lui-même une co-référence, alors le paramètre réfère directement au concept référé par l'argument.

Suite au déclenchement d'un concept défini avec paramètres, on pourrait avoir une *transmission des états* des arguments aux états des paramètres correspondants.

Définition : *Transmission des états*
Pour un concept C déclenché et défini avec paramètres, si un argument en entrée de C est mis à l'état "déclenchement ; ?" alors l'état du paramètre correspondant est mis aussi à l'état "?". De même, si un argument en sortie de C est à l'état "attend-valeur ; @" quand le concept C s'apprête à commencer son activation, alors l'état du paramètre correspondant est mis à l'état "?". ♦

Le graphe de généralisation de base de Synergy contient quatre types primitifs qui correspondent à quatre forme d'activités : ProcedureActivity, ProcessActivity, StrictActivity et LazyActivity. ProcedureActivity et ProcessActivity concernent la "durée de vie" d'une activité alors que StrictActivity et LazyActivity concernent le mode d'évaluation de ses arguments (si elle en a). Comme le montrent les prochaines définitions, un nouveau type peut être déclaré comme une spécialisation d'un (ou plusieurs) de ces types. Par exemple, si un nouveau type doit être interprété comme une procédure paresseuse, le programmeur doit alors ajouter sa définition en MLT comme une spécialisation de [ProcedureActivity] et de [LazyActivity], en reliant la définition aux deux concepts par des liens -sp->.

Définition : *Le type primitif ProcedureActivity*
Un concept dont le type est un sous-type de **ProcedureActivity** est qualifié de "concept-procédure". La valeur d'un concept-procédure est détruite après son activation. ♦

Ainsi, une ré-activation d'un même concept-procédure implique l'activation d'une nouvelle valeur. Cette interprétation procédurale d'une activité correspond à l'interprétation d'une procédure (ou fonction) dans la programmation procédurale, où un appel à une procédure

implique la création d'un "dossier d'activation" et la fin d'exécution de la procédure, la destruction du dossier [Pratt, 84].
La création d'un dossier d'activation suite à un appel de la procédure correspond en Synergy à la détermination de la valeur d'un concept-procédure (éventuellement la création de la valeur) suite au déclenchement du concept, la destruction du dossier correspond à la destruction de la valeur du concept.

Définition : *Le type primitif ProcessActivity*
Un concept dont le type est un sous-type de **ProcessActivity** est qualifié de "concept-processus". A la différence de celle d'un "concept-procédure", la valeur d'un concept-processus n'est pas détruite après son activation. ♦

Ainsi, la même valeur d'un concept-processus peut être exécutée (activée) plusieurs fois suite à plusieurs activations du même concept, une nouvelle exécution ferait donc suite à la précédente.

Note: Un (sous)type d'activité qui n'est pas un sous-type de ProcedureActivity ou de ProcessActivity est considéré par défaut comme un sous-type de ProcessActivity.

Les types StrictActivity et LazyActivity définis ci-dessous, sont considérés uniquement si l'activité est paramétrée.

Définition : *Le type primitif StrictActivity*
Un concept dont le type est un sous-type de **StrictActivity** est qualifié de "concept-strict". Sa valeur ne peut être activée (ou exécutée) que si tous les arguments en entrée du concept ont des valeurs. ♦

Définition : *Le type primitif LazyActivity*
Un concept dont le type est un sous-type de **LazyActivity** est qualifié de "concept-paresseux". Pour activer sa valeur, un concept-paresseux ne vérifie pas que ses arguments en entrée aient des valeurs. En d'autres termes, sa valeur peut être activée même si ses arguments en entrée n'ont pas de valeurs. ♦

Note : Un (sous)type d'activité qui n'est pas un sous-type de StrictActivity ou de LazyActivity est considéré par défaut comme un sous-type de StrictActivity.

Les distinctions entre *Procedure* et *Process,* d'une part, et entre *Strict* et *Lazy,* d'autre part, sont respectivement importantes dans les programmations procédurale et fonctionnelle.

3.4.3 Héritage dans Synergy

L'héritage est un mécanisme d'abstraction très important, mis en valeur par la programmation orientée classe/objet. Une classe est composée d'une partie déclarative (les attributs de la classe) et d'une partie procédurale (l'ensemble des méthodes de la classe), une classe peut avoir plusieurs instances (appelées aussi objets) et une ou plusieurs super-classes. Une instance "hérite" en général de sa classe (aussi bien des attributs que des méthodes) et des super-classes de sa classe.

Définition : *Classe*
Une classe peut être représentée en Synergy par un type de concept et son contenu (les attributs et les méthodes) par un GC (ou une collection de GC). Une classe peut avoir une *super-classe*, représentée dans le corps de la classe par un concept avec le mot clé **super** comme référent : [TypeSuperClasse : super], ce dernier est appelé "concept-super". ♦

Les attributs d'une classe sont représentés par des concepts et les méthodes par des sous-graphes du GC décrivant la classe. Les prochaines définitions indiquent comment l'*héritage par délégation* est "naturellement" réalisé en Synergy, pour les attributs aussi bien que pour les méthodes.

Définition : *Héritage des attributs*
L'héritage des attributs est relié à la résolution de co-référence et à la présence ou non du concept-super dans le contexte courant. Pour résoudre une co-référence "r1.r2...rN", la procédure de résolution cherche dans le contexte courant un concept avec "r1" comme référent. S'il n'y est pas, elle regarde si le contexte courant contient un concept-super. Si oui, la recherche du concept avec "r1" comme référent se poursuit dans la description du concept-super (si ce dernier n'a pas de description, elle est tout d'abord determinée). ♦

Exemple (extrait du manuel de Synergy) :
Le type Nurse hérite du type Agent puisque Nurse possède dans sa définition un concept-super dont le type est Agent (Figure 3.7). Si nous avons le concept [Nurse :Magy] présent dans un contexte s1 et nous demandons la description de Magy, Synergy va la créer par instanciation:

[Nurse :Magy = [:position]<-has-[Agent :super]<-agnt-[Assessment :assessment]-obj->[Patient]]

Pour avoir par exemple la valeur du concept [Queue :Magy.messageQueue], la co-référence Magy.messageQueue doit être résolue. Puisqu'il n'y a pas de concept dans la description de Magy avec comme référent messageQueue, et puisque la description contient un concept-super

[Agent : super], la recherche se poursuit alors dans la description de ce dernier (après sa création par instanciation).

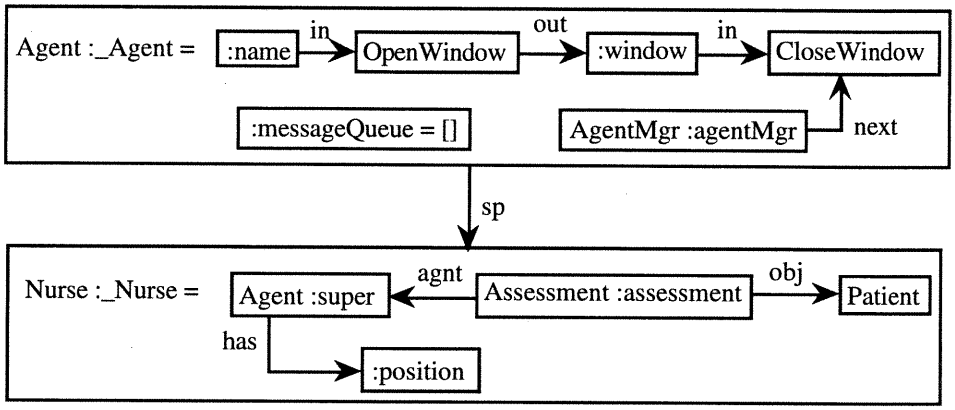


Figure 3.7 : Définitions et héritage

Comme le montre la Figure 3.8, la description du concept-super contient un concept avec le référent messageQueue. Le concept référé est ainsi localisé et la co-référence entre les concepts [Queue :Magy.messageQueue] et [:messageQueue = []] est établie ; la valeur du premier concept est un "pointeur" sur le second.

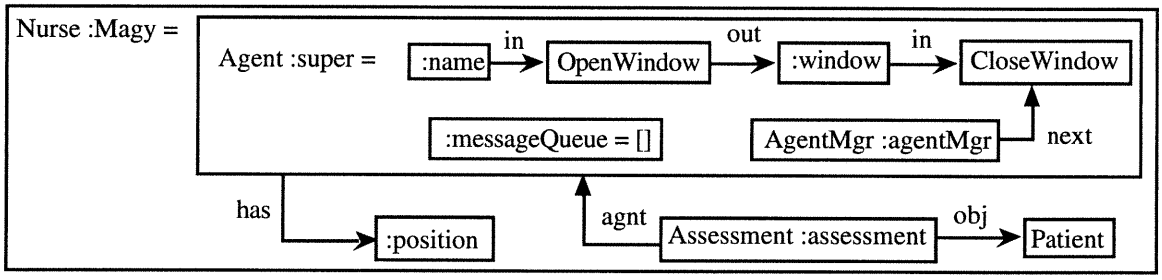


Figure 3.8 : Description de Magy

La réalisation de l'héritage des attributs en Synergy est donc basée essentiellement sur la transformation suivante d'une co-référence CoRef :

Si le contexte courant contient un concept avec "super" comme référent Alors

{ CoRef := "super." + CoRef ; /** *cet ajout permettra de "rentrer" dans la description du concept-super* **/

Essayer de nouveau la résolution de la co-référence CoRef }

Ainsi pour notre exemple, la co-référence "messageQueue" est devenu "super.messageQueue". La section suivante complète la discussion sur l'héritage des attributs en Synergy.

Définition : *Héritage des méthodes*
Quand une instance d'une classe reçoit un message, ses méthodes sont considérées, si celles-ci ne peuvent pas répondre au message, ce dernier est envoyé (délégué) au concept-super (si l'instance en a un). ♦

Le chapitre 5 (et en particulier la section 5.2) fournit plus de détails sur l'héritage des méthodes.

3.4.4 Co-référence et résolution de co-référence

Comme mécanisme d'abstraction, la co-référence permet de référer à un concept décrit ailleurs. La formulation d'un contexte est ainsi plus concise. La réalisation de ce mécanisme d'abstraction se base sur la procédure de résolution de la co-référence, initiée par la demande de valeur d'un concept avec co-référence. La résolution consiste à rechercher dans la *hiérarchie des contextes* le concept référé, afin d'avoir sa valeur.

Définition : *Hiérarchie des contextes*
La hiérarchie des contextes est composée des contextes actifs durant une session d'exécution. Elle est formée et continuellement mise à jour par l'interpréteur de Synergy. La racine de la hiérarchie des contextes est la Mémoire à Long Terme (MLT), avec la Mémoire de Travail (MT) comme descendant direct. Chaque concept actif de la MT qui a un GC comme valeur, constitue un descendant direct de la MT et ainsi de suite. ♦

La définition suivante présente les différents cas de résolution d'une co-référence.

Définition : *Résolution de co-référence*
→ La *résolution* de la co-référence [T : r.] qui se trouve dans le contexte s (un nœud dans la hiérarchie des contextes) est effectuée par une recherche ascendante qui commence avec le contexte s, se poursuit avec le "père" de s et ainsi de suite jusqu'à ce qu'un concept avec référent "r" soit localisé dans le contexte courant ou que la racine de la hiérarchie des contextes soit atteinte.

→ La résolution de la co-référence [T : r1.r2. ... .rN] commence avec la résolution de "r1." comme indiqué ci-dessus. Une fois le concept avec le référent "r1" localisé, et après avoir déterminé la description du concept, si celui-ci n'en possédait pas encore une, la recherche descendante est amorcée avec la description du concept comme racine : la résolution tente de localiser dans la description courante un concept avec "r2" comme référent, ensuite elle tente de localiser un concept avec "r3" comme référent dans la description du concept ayant "r2" comme référent et ainsi de suite pour chaque élément "rj" de la co-référence.

Pour tenir compte de l'héritage et autres aspects, le traitement général ci-dessus est augmenté par les considérations suivantes :

→ Rappelons que la MLT est considérée comme un concept, sa description étant le graphe de généralisation. Le programmeur peut fournir un référent particulier pour ce concept (par exemple [LTM :IcuAppl =..<GrapheGeneralisation>..]); s'il ne le fournit pas, Synergy assigne au concept le mot clé **memory** comme référent par défaut.
Le programmeur peut référer directement à un objet défini dans la MLT en spécifiant le référent de cette dernière comme premier élément de la référence : "IdMem.r1. ..." ou "memory.r1. ...".
Cette référence directe à la mémoire est résolue comme suit : avant de commencer la résolution d'une co-référence, la procédure de résolution teste si le premier élément de la co-référence est le référent de la MLT. Si oui, la recherche commence directement dans la mémoire (qui correspond à la racine de la hiérarchie des contextes) afin de localiser le concept avec "r1" comme référent, la recherche dans la hiérarchie des contextes est ainsi évitée.

→ L'héritage est intégré à la résolution de co-référence comme suit : quand un concept avec le référent "rj+1" ne peut être localisé dans la description du concept avec le référent "rj", la procédure de résolution vérifie si la description de "rj" contient un concept-super. Si oui, la référence "rj+2. ..." est transformée en "super.rj+2. ..." et la procédure de résolution poursuit son traitement avec cette nouvelle co-référence.

→ Tout concept, excepté un concept-super, possède une identité, un *self*. Le concept-super n'a pas une "identité" car il constitue une "partie" (une extension ou un "annexe") du contexte courant. Le programmeur peut utiliser le mot clé **self** pour référer à un concept ayant un *self*.
La résolution d'une co-référence contenant *self*, "self.r1. ...", est réalisé comme suit : si le contexte où se trouve la co-référence est celui d'un concept-super alors la résolution ignore ce contexte et passe directement au contexte père. Si ce dernier est lui-même un concept-super, il est ignoré également et ainsi de suite jusqu'à ce qu'on arrive à un contexte sj ne correspondant pas à un concept-super. Si la référence ne peut pas être résolue dans le contexte sj, la procédure de résolution continue alors la recherche d'un contexte ayant un self plus "englobant".

→ Le contexte qui correspond au corps d'une définition peut être référé par son paramètre formel (le référent de la définition). Cette possibilité est souhaitable si la recherche d'un

concept doit commencer à partir du contexte de base de la définition et non du contexte courant (qui peut être imbriqué dans le contexte de la définition).
Notons que la résolution d'une co-référence contenant le paramètre formel est effectuée dans le contexte d'une instance et non dans celui de la définition (celle-ci n'est qu'un modèle pour créer des instances).
Par exemple, si on a la définition [Nurse :_Nrse = ... [Integer :_Nrse.NbreHours] ...] et si on a dans un contexte les concepts [Nurse :Janet] et [Nurse :Cathi] et si on demande la description des instances Janet et Cathi, on aura alors suite à l'instanciation :

[Nurse :Janet = ... [Integer :_Nrse.NbreHours] ...] et [Nurse :Cathi = ... [Integer :_Nrse.NbreHours] ...]

La résolution de "_Nrse.NbreHours" est effectuée comme si _Nrse correspondait à Janet pour le premier cas et à Cathi pour le second cas. ♦

La Figure 3.9 illustre la majorité des cas introduits ci-dessus. Considérons en particulier ceux désignés dans la figure par (a), (b) et (c) respectivement (pour chacun nous avons spécifié un lien qui pointe sur le concept-référent et un autre sur le concept référé) :

→ Le cas (a) : Considérons le concept [Tr :mr.] qui est dans la description de [T :rfa =...]. La résolution commence par chercher un concept avec le référent "mr" dans le contexte courant [T :rfa =...]. Supposons qu'il n'existe pas un concept de référent "mr" dans la description de "rfa" [T :rfa =...] ni dans son héritage, la recherche continue alors avec le contexte "père" : [Type: _X = ...]. La description de ce concept ne contient pas un concept avec le référent "mr" mais elle contient un concept-super [T2: super =...], la recherche se poursuit donc dans [T2: super =...]. Comme le montre la Figure 3.9, c'est dans le concept-super de [T2: super =...] que le concept [Tr :mr] se trouve.

→ Le cas (b) : Considérons maintenant la résolution de [Tb :self.titi] qui est dans la description de [T5 :super =...]. Dans sa recherche ascendante, la résolution "saute" les contextes [T5: super =...], [T4: super =...] et [T2: super =...] puisqu'ils n'ont pas de *self*. Elle arrive alors au contexte [Type: _X =...] et cherche le concept [Tb : titi]. [Type: _X =...] ne contient pas un tel concept et il en est de même pour son concept-super. La recherche se poursuit donc avec le contexte père [TypeA: _A =...] qui a aussi un *self* et qui contient un concept avec "titi" comme référent. Le concept référé par [Tb :self.titi] est ainsi localisé.

→ Le cas (c) : La résolution de [Tc : self.tata] qui se trouve dans la description de [T5 :super =...] illustre un autre point. Comme pour la co-référence précédente, la résolution "saute" les contextes [T5: super =...], [T4: super =...] et [T2: super =...] et cherche dans le contexte de [Type: _X =...] un concept avec "tata" comme référent. [Type: _X =...] ne contient pas un tel concept mais son concept-super oui. Notons que le contexte [T4: super =...] contient aussi un concept avec "tata" comme référent, le concept [Tc : tata] dans [T2: super =...] a toutefois la priorité.

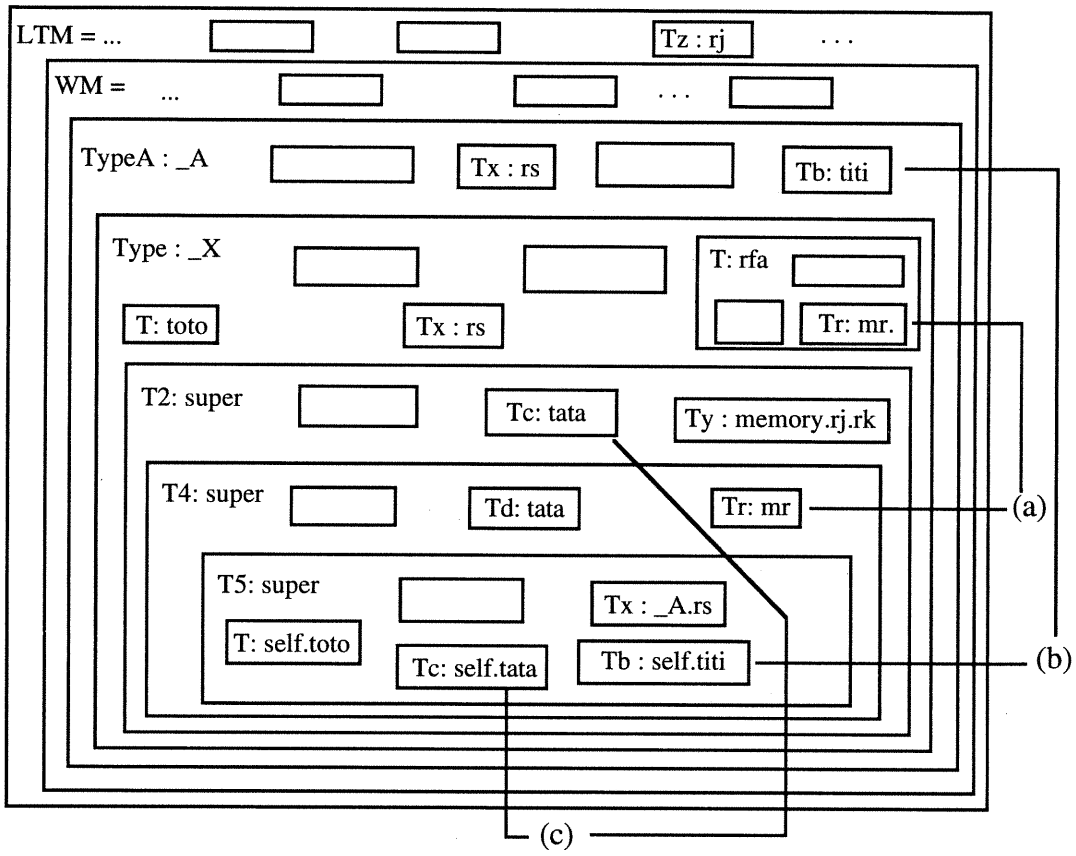


Figure 3.9 : Exemples de résolution de co-référence

3.4.5 Formulation abstraite d'une fonction

Avoir une formulation plus concise est un effet de l'abstraction, c'est le cas de la formulation d'une fonction en Synergy. Une fonction est une opération qui retourne un seul résultat. Synergy offre au programmeur la possibilité de spécifier le résultat séparement de la fonction (un concept pour le résultat et un concept pour la fonction) ou de n'utiliser qu'un seul concept pour les deux (Figure 3.10). Notons que cette différence de formulation n'est pas associée à la signature de l'opération mais à son utilisation : la Figure 3.10 montre deux utilisations (appels) différentes de la même fonction *.

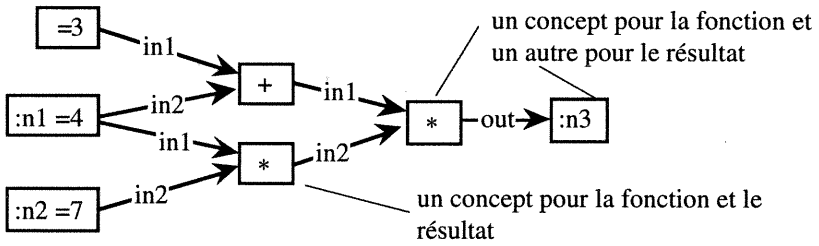


Figure 3.10 : Exemple de formulation abstraite d'une fonction

Remarque (technique) : Si un concept représente aussi bien la fonction que son argument résultat et si le type du concept est défini, alors son activation nécessite un traitement particulier. En effet, un tel concept peut avoir deux valeurs s'il est actif : son corps de fonction et la valeur calculée (l'ancienne valeur ou la nouvelle). Synergy résout ce "problème" en mettant le corps de la fonction dans la liste des contextes actifs et la valeur calculée dans le champ valeur du concept.
Un problème identique survient avec les états : en tant que fonction active, le concept peut être à l'état "en-activation ; !@", et en tant que résultat il peut être à l'état "déclenchement ; ?" (en raison d'une affectation). Pour éviter ce problème, Synergy garde le concept à l'état "!@" et l'ajoute dans la liste LForward. Cette dernière enregistre des concepts qui doivent initier, au prochain cycle de l'interpréteur, une propagation en avant, comme le ferait un concept à l'état "?".

3.5 Activation des GC et interpréteur de Synergy

Définition : *Activation d'un GC*
L'activation d'un GC correspond à une interprétation parallèle des cycles de vie de ses concepts.
Elle commence par l'activation parallèle de certains concepts du graphe et se propage ensuite en parallèle à travers le graphe, via les relations procédurales. Les attributs optionnels des relations procédurales permettent de contrôler la propagation. ♦

Si le type d'un concept actif est une opération primitive, alors celle-ci est activée et si la valeur d'un concept est un GC alors ce dernier est activé à son tour. On a donc en général, une activation parallèle de plusieurs GC et de plusieurs opérations primitives. La section 3.5.3, qui fournit une description générale de l'interpréteur Synergy, présente plus de détails sur cet aspect. L'environnement graphique de Synergy, décrit dans la section 3.7, aide le programmeur à naviguer dans (et à explorer) cet espace dynamique, composé des GC actifs.

En général, un concept ne peut être activé que si ses préconditions sont satisfaites, pré-conditions formulées à l'aide de certaines relations procédurales. Aussi, après l'activation d'un concept, ses post-conditions, formulées également à l'aide de certaines relations procédurales, doivent être satisfaites. Le *cycle de vie* d'un concept montre cette "activation contextuelle" d'un concept.
Nous allons introduire tout d'abord les règles de propagation, définies sur les relations procédurales, qui sont prises en compte dans la formulation du cycle de vie d'un concept. Ce dernier sera introduit par la suite.

3.5.1 Règles de propagation de l'activité dans un GC

Comme le souligne la définition précédente, l'activation d'un GC est initiée généralement par une demande simultanée à des concepts du graphe de "déterminer et activer ta valeur". Cette demande correspond à mettre le concept à l'état de déclenchement "?". Les relations procédurales attachées aux concepts déclenchés vont ensuite *propager* la demande à travers le graphe, si certaines conditions sont vérifiées. La propagation peut se faire "en avant" (du concept source au concept cible de la relation) ou "en arrière" (du concept cible au concept source de la relation).

Définition : *La propagation en avant*
Si un GC actif contient la branche C1 -Lien-> C2, où Lien ∈ {"in", "out", "grd", "next", "coref"} avec éventuellement des attributs autres que "/" (défini ci-dessous), et si C1 a "répondu" à la demande "déterminer et activer ta valeur" (et si, pour la relation "grd", la valeur de C1 après son activation est différente de "false"), alors le lien Lien va propager la demande en avant vers C2.
Si C1 est la source de plusieures Lien alors la propagation via ces liens s'effectue en parallèle. ♦

Note : le concept *répond* à la demande s'il a déterminé et activé sa valeur et si celle-ci a terminé son exécution.

Nous sommes à présent en mesure de définir l'attribut optionnel "coupe-propagation-avant", noté "/", qu'une relation "in", "out", "grd" ou "coref" pourrait avoir. Cet attribut permet d'inhiber la propagation en avant.

Définition : *L'attribut* coupe-propagation-avant "/"
La présence de l'attribut "coupe-propagation-avant" dans une relation in (in,/), out (out,/), grd (grd,/) ou coref (l'attribut "/" est ajouté en fait au co-référent : co-referent,/) annule (inhibe) la propagation en avant par cette relation. ♦

Notons que pour le cas du out : C1 -out,/-> C2, C1 pourrait affecter une valeur à C2 sans que ce dernier soit mis à l'état de déclenchement (le "/" n'annule pas l'affectation mais uniquement la propagation du déclenchement).

Définition : *La propagation en arrière*

Si un GC actif contient la branche C1 -Lien-> C2, où Lien ∈ {"in", "out", "grd", "coref"} avec éventuellement des attributs autres que "\" (défini ci-dessous), et si C2 a reçu la demande "déterminer et activer ta valeur", (la suite dépend de la relation) :

→ et si Lien = "in" et le type de C2 est une activité "stricte" (tous les arguments en entrée de C2 doivent avoir une valeur avant que C2 commence son activation) et si C1 n'a pas de valeur et qu'il est à l'état "repos", alors le lien va propager la demande en arrière vers C1.

→ et si Lien = "out" ou "coref", et C2 n'a pas de valeur et qu'elle ne peut être déterminée par instanciation (si le type de C2 n'est pas défini) alors la relation va propager la demande en arrière vers C1.

→ et si Lien = "grd" et C1 est à l'état "repos" et qu'il n'a pas de valeur, alors la relation va propager la demande en arrière vers C1.

Si C2 est la cible de plusieurs relations de types "in", "out" ou "grd", alors la propagation via ces liens s'effectuera en parallèle. ♦

Nous sommes à présent en mesure de définir l'attribut optionnel "coupe-propagation-arrière", noté "\", qu'une relation "in", "out", "grd" ou "coref" pourrait avoir. Cet attribut permet d'inhiber la propagation en arrière.

Définition : *L'attribut* coupe-propagation-arrière "\"

La présence de l'attribut "coupe-propagation-arrière" dans une relation in (in,\), out (out,\), grd (grd,\) ou coref (l'attribut "\" est ajouté au co-référent : co-referent,\) annule la propagation en arrière via cette relation. ♦

Remarque : Les attributs "coupe-propagation-avant" et "coupe-propagation-arrière" ont un rôle similaire au "cut" Prolog ; le "cut" permet un contrôle sur le retour-arrière alors qu'ici, les deux attributs permettent respectivement un contrôle sur les propagations en avant et en arrière.

Remarque sur l'activation d'un lien de co-référence

Il faut déterminer tout d'abord si l'activation d'un concept-référence *devrait* influencer celle du concept référé et/ou inversement. Cela dépend en fait de l'interprétation associée au lien de co- référence :

→ Si on considère que le lien décrit une "dépendance mutuelle" entre le concept référé et le concept-référence, alors la demande de valeur, formulée dans le contexte du concept référé ou dans celui du concept-référence, est automatiquement propagée aux autres contextes

(y compris le contexte des autres concept-références). Aussi, quand la valeur du concept référé est déterminée, les concept-références qui sont à l'état "attend-valeur ; @" vont (et doivent) changer leur état.

→ Si on considère que le lien de co-référence ne décrit pas une dépendance mutuelle, alors l'activation du concept référé n'influencera pas celle d'un concept-référence et inversement, et la demande de valeur dans un contexte (du concept référé ou d'un concept-référence) ne sera pas propagée aux autres contextes.
De même, un concept-référence à l'état "attend-valeur ; @" va attendre une valeur *de son contexte seulement.* Ainsi, si une valeur est déterminée dans un contexte (d'un concept-référence par exemple), alors les autres concepts-références qui sont en attente de valeur ne "percevront" pas la présence de la valeur.

Les deux interprétations sont considérées en Synergy : la co-référence spécifie par défaut une dépendance mutuelle entre le concept référent et le concept référé, le programmeur peut toutefois l'annuler partiellement ou en totalité, selon qu'il spécifie l'attribut "coupe-propagation-avant ; /", l'attribut "coupe-propagation-arrière ; \" ou les deux en même temps.

3.5.2 Cycle de vie d'un concept

Définition : *cycle de vie d'un concept*

→ Le cycle de vie d'un concept est un diagramme de transition d'états (Figure 3.11) où les états correspondent aux états possibles du concept et les transitions aux conditions/actions à vérifier/effectuer pour qu'un concept évolue d'un état à un autre. Les conditions/actions portent sur les champs et/ou le voisinage du concept, en particulier les relations procédurales qui lui sont connectées. Plus précisement :

→ Un concept C à l'état "déclenchement ; ?" va déterminer et activer (évaluer ou exécuter) sa valeur. Pour la détermination de la valeur de C, quatre cas sont possibles : a) le concept C possède déjà une valeur, b) le type de C est un primitif (la valeur de C correspond alors à un appel de primitif), c) la valeur de C peut être obtenue par instanciation de la définition du type de C, ou d) initier le calcul de la valeur de C par une propagation arrière, via tout lien "out" et/ou "coref" qui n'a pas l'attribut "\". Pour ce dernier cas, le concept C devient à l'état "attend-valeur ; @". Le concept C peut rester indéfiniment dans cet état, mais s'il reçoit une valeur, il deviendra de nouveau à l'état "déclenchement" (cette fois-ci pour indiquer qu'il a une valeur).

→ Quand un concept C est à l'état "déclenchement ; ?" et a une description, ses pré-conditions (s'il en a) sont considérées et C devient à l'état "vérifier préconditions ; @prc". Les pré-conditions sont formulées à l'aide des liens "in" et "grd". Ces liens peuvent initier des propagations en arrière à partir de C. Tant que toutes les pré-conditions ne sont pas

satisfaites, le concept C reste à l'état "attend-préconditions ; @prc@". C peut rester indéfiniment dans cet état.

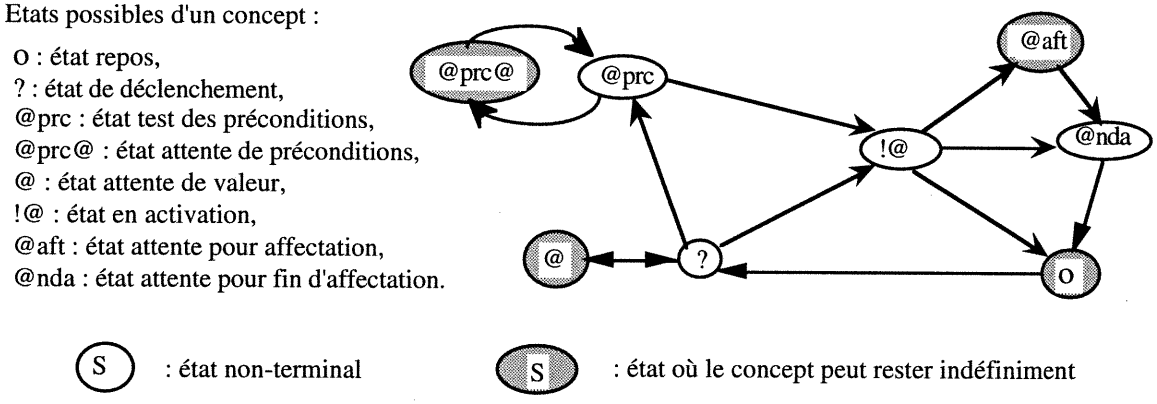


Figure 3.11 : Le cycle de vie d'un concept

→ Une fois les pré-conditions du concept C (s'il en a) satisfaites, sa valeur est activée et le concept devient à l'état "en-activation ; !@". Si la valeur est un appel primitif (le type du concept représente une opération primitive) alors son activation correspond à l'exécution de la primitive, si la valeur est un GC alors ce dernier est activé, si la valeur est un autre type de donnée alors son activation est "nulle" (activation instantanée et sans effet).

→ Une fois l'activation de la valeur du concept C terminée, les post-conditions de C (s'il en a) sont considérées. L'affectation des valeurs calculées par C (dans le cas où le type de C est une opération primitive) est un exemple de post-condition : si un concept C est relié à un concept C1 avec le lien "out,f" (C -out,f-> C1) et si C1 a déjà une valeur et que C a produit une valeur pour C1, alors C devra attendre que C1 soit sans valeur. Le concept C sera alors à l'état "attente-pour-affectation ; @aft" et il peut rester indéfiniment dans cet état (par exemple, tant que la valeur de C1 n'est pas consommée).

Même si les valeurs calculées par C peuvent être affectées aux arguments correspondants, C peut encore attendre en raison des "conflits d'affectation" qui peuvent survenir quand plusieurs concepts s'apprêtent, en même temps, à affecter des valeurs à un même concept.

Après l'activation du concept C (et l'affectation des résultats si nécessaire), la propagation en avant à partir de C: C -Lien-> C1, où Lien peut être "in", "out", "guard", "next" ou "coref", peut constituer une autre post-condition.

→ Une fois les post-conditions du concept C traitées, C retourne à l'état repos ("o"). ♦

3.5.3 Aperçu sur l'interpréteur de Synergy

En général, l'exécution d'un programme Synergy correspond à une activation parallèle de plusieurs GC et opérations primitives. L'activation d'un GC correspond elle aussi à une activation parallèle de plusieurs concepts du graphe. Ce parallélisme inhérent à Synergy est "réalisé" par la boucle principale de l'interpréteur. Ce dernier considère en effet à chaque cycle, tous les GC et les primitives actives et, pour chaque GC, considère tous les concepts qui nécessitent un traitement (de sa part).
L'activation parallèle d'un GC est rendue possible grâce, essentiellement, à l'association d'un état à un concept; chaque concept est intérprété selon son cycle de vie, l'activation (ou interprétation) d'un GC est ainsi décentralisée. En effet, l'interpréteur de Synergy simule l'exécution parallèle des cycles de vie des concepts actifs (c'est comme si à chaque concept est associé un processeur qui l'exécute, selon son cycle de vie).

Nous précisons dans cette section le rapport entre le parallélisme "conceptuel" (en l'occurrence le cas de Synergy) et le parallélisme "physique" (un multi-processeur) et nous décrivons la boucle principale de l'interpréteur de Synergy. L'annexe B complète la description algorithmique en fournissant l'interprétation des états d'un concept. L'annexe C fournit la description algorithmique de la procédure de résolution d'une co-référence.

L'algorithme (plus précisement, le haut niveau) de l'interpréteur, extrait du manuel de Synergy, est décrit ci-dessous. Voici quelques commentaires :

→ La première partie de l'interpréteur concerne l'initialisation de certaines structures globales. En particulier, la liste des contextes actifs (LActiveCtxts) contient au départ les deux contextes de base : ceux de la mémoire à long terme (MLT) et de la mémoire de travail (MT). La structure d'un contexte (l'enregistrement Context) est décrite ci-dessous. Pour chaque contexte on spécifie son contexte père. Ainsi, pour le contexte de la MT, son père est le contexte de la MLT.
La liste des contextes actifs sert essentiellement à informer l'interpréteur des contextes actifs et à permettre la résolution de la co-référence. C'est ce second rôle qui justifie la présence, dans cette liste, d'un contexte pour la MLT. Pour le contexte de la MT, sa présence est justifiée par les deux rôles (rappelons que l'activation commence avec la MT).

→ La boucle principale est conditionnée par la MT dont le contenu est un GC ou une collection de GC (while IsActiveCG(WMemory.Value)...). Suite à l'activation de certains concepts dans la MT, d'autres GC (valeurs de concepts actifs) deviennent actifs et ainsi de suite. Tous les GC actifs sont interprétés en parallèle.
Le corps de la boucle principale reflète la composition du cycle de vie d'un concept :

Pour *tout* GC actif, on interprète *tout* concept du graphe dont l'état est "déclenchement ; ?" et ensuite *tout* concept dont l'état est "verifie-preconditions ; @prc". Notons que suite à l'interprétation d'un concept à l'état "?", ce dernier peut devenir à l'état "@prc". Si c'est le cas, il sera interprété une seconde fois selon son nouveau état.
Après cette première phase, un concept peut être à l'état "@", "@prc@", "o" ou il peut être prêt pour l'activation, auquel cas la valeur du concept est activée (qu'il s'agisse d'une primitive ou d'un GC) et le concept est mis à l'état "en-activation ; !@".
Ainsi, la première partie de la boucle de l'interpréteur reflète la "partie gauche" du cycle de vie d'un concept, si on prend l'état "!@" comme centre.

Deux cas particuliers sont traités dans cette première partie de la boucle :

♦ co-références entre concepts appartenant à différents GC actifs : rappelons que l'interprétation d'un concept peut influencer celle d'autres concepts, via les co-références. Un traitement particulier est adopté pour préserver le parallélisme (dans l'interprétation des concepts) vis-à-vis de cette influence : à chaque cycle on détermine en premier et pour chaque GC actif, les concepts à l'état "?". Ces derniers sont enregistrés dans la liste Lc? associée au GC (toutes les listes Lc? sont groupées dans la liste Lcs?).
Ainsi, même si l'état d'un concept C a changé de "?" à un autre état, à cause de l'interprétation d'un concept co-référent, le concept C est interprété selon son ancien état. On simule ainsi le fait que les concepts des GC actifs sont interprétés en même temps.

♦ un concept-fonction actif qui a reçu une valeur : rappelons que dans ce cas, le concept-fonction n'est pas mis à l'état "?", comme il devrait l'être suite à une affectation, mais il est enregistré dans la liste LForward afin d'être interprété comme s'il était à l'état "?". A cet effet, on prend chaque élément de la liste et on initie une propagation en avant.

Considérons maintenant la seconde partie de la boucle qui reflète la partie "droite" du cycle de vie d'un concept :
→ Les deux procédures TestTerminationPrimitives et TestTerminationCGs, définies dans l'annexe B, testent la terminaison de tout concept actif, qu'il soit primitif ou GC. Si le concept a terminé son exécution, on considère alors ses post-conditions (par exemple l'affectation des résultats pour les primitives).
Les autres étapes sont : considérer les concepts à l'état "attente-pour-affectation ; @aft" (vérifier si un concept n'a plus à attendre et peut donc commencer l'affectation), effectuer ensuite les affectations (le rôle de la procédure AffectationManagment) et traiter tout concept à l'état "attente-fin-affectation ; @nda" pour lequel toutes les valeurs en sortie ont été affectées.

→ La troisième et dernière partie de la boucle principale termine l'itération courante et prépare la prochaine. Ainsi, pour chaque concept actif primitif, on décrémente de 1 son

"temps-d'exécution Synergy" (une unité de temps Synergy correspond à une itération de la boucle principale). En effet, pour Synergy, le temps d'exécution de la valeur d'un concept (qu'elle soit une primitive ou un GC) est égal au nombre d'itérations de la boucle principale, du début de l'activation du concept à sa terminaison. Pour un concept primitif, le temps d'exécution "physique" est celui du "cpu", ce temps est toutefois converti en unité de temps Synergy comme suit : 1 temps-Synergy = Cste * 1 temps-cpu, avec Cste un coefficient fixé au départ.

Remarques :

→ Dans l'implantation actuelle de Synergy, nous adoptons une forme plus simple de conversion : toute opération primitive prend 2 temps-Synergy. Aussi, une fois une primitive activée, on suppose qu'il n'y a aucune interaction entre elle et les autres parties du "programme". Synergy permet et gère uniquement les interactions entre les concepts définis; s'il doit y avoir une interaction entre certaines opérations, il faudrait alors les définir en Synergy.

→ "Parallélisme conceptuel" vs "parallélisme physique" : Synergy est un langage parallèle (par défaut), le programmeur conçoit ainsi ses programmes Synergy. Par ailleurs, l'environnement graphique montre l'activation parallèle des GC et l'interpréteur est défini essentiellement pour gérer le parallélisme de Synergy. Maintenant, le code de l'interpréteur (en l'occurrence du C++) est interprété par une machine séquentielle. L'interpréteur constitue le pont entre le parallélisme du langage et la nature séquentielle de la machine.

→ Dans une implantation "physique" du parallélisme de Synergy, l'exécution parallèle des "primitives" ne serait pas simulée et une "primitive" pourrait correspondre à un processus qui s'exécute en parallèle (physiquement), avec les autres parties du programme Synergy et avec la possibilité de synchronisation et de communication via des canaux, des messages, un espace commun, ou autre moyen de communication.

Structures de base (extrait du manuel de Synergy)

```
Concept = {  Type        : String  ;
             Ref         : String ;
             ParamCase : String ; /**** Optional ****/
             Val         : Object ;
             State       : {o, ?, @, @prc, @prc@, !@, @aft, @nda}
             att/, att\ : Boolean ;   /**** Optional ****/
             IncomeRelations : ListOf Relations ;
             OutcomeRelations : ListOf Relations ;
          ... Other technical and graphical informations ...  } ;
```

♦ *Ref* est une chaîne qui représente un identificateur d'instance, un identificateur de variable ou une co-référence. Si c'est une variable avec une valeur, alors la forme de Ref sera celle-ci : IdentVar=Referent.

♦ *Val* est un objet qui peut être un nombre, un booléen, une chaîne de caractères, une fenêtre, un GC ou une liste d'objets. Si le concept est une co-référence, on met dans son champ Val le pointeur au concept référé, une fois trouvé.

♦ *ParamCase* est un champ optionnel, utilisé uniquement pour un concept qui est un paramètre. La valeur du champ sera alors "**inX/**" ou "**outX/**".

Les méthodes définies sur l'objet Concept sont : Type, Referent, Value, State, attr/ et attr\.
Une méthode peut être utilisée pour lire la valeur d'un champ ou pour y affecter une nouvelle valeur. Un appel à une méthode se fait comme suit : C.Méthode où C est un concept.
Deux autres méthodes sont utilisées :

♦ C.GetVal(Ctxt: Context) : "GetVal" résout la co-référence (s'il y a lieu) et retourne la valeur "effective" d'un concept (ou nil s'il n'en a pas), contrairement à la méthode Value qui effectue un simple accès au champ Val du concept. GetVal correspond à la procédure de résolution de la co-référence, elle est fournie dans l'annexe C.

♦ C.SetVal(V: Object, Ctxt: Context) : Cette méthode détruit l'ancienne valeur et affecte la nouvelle. Si le concept est une co-référence, la valeur est affectée au concept référé.

♦ C.SetState(Ctxt: Context, S: State, AffectType: String) : elle est définie ci-dessous.

```
Relation = {  Type      : String ;  /*** the name of the relation ***/
              Range     : Integer ;  /*** 0 if not specified ***/
              ConsAttr : {"f"} ; /*** optional ***/
              CutFrwd   : {"/"} ; /*** optional ***/
              CutBkwrd  : {"\"} ; /*** optional ***/
              Source, Target : Concept ;
              ... Other technical and graphical informations ...  } ;

CG = {  Concepts : ListOf Concept ;
        Relations : ListOf Relations ;
        ... Other technical and graphical informations ... } ;

Context = {   FormalParam : String ;   /*** optional ***/
              ActiveCG :  CG ;
              PreviousContext : Context ;
              LConceptsWithArgs :  ListOf <Concept, LRangeArgsIn> ;
                    /*** an element of this list records the list of ranges of the income arguments of a
                         concept in the CG that have the state "?".
                         The reason for this recording is introduced later ***/
          } ;

LActiveCtxts, LNewActiveCtxts : ListOf Context ;

LPrimitivesInExec : ListOf  { Ctxt : Context ;
                              C : Concept ;
                              LValues :  ListOf object ;
                              D : Integer   } ;

LAffectation = ListOf { Ctxt : Context ;
                        C : Concept ;
                        V : Object ;
                        A : Concept } ;
```

```
LForward : ListOf Concept ;

LReceives = ListOf  {C : Concept ;
                     Queue : List of Object ;
                     M : Object ;
                     ResBool : Boolean } ;

LRemoves = ListOf  {Queue : List of Object ;
                    Elem : Object }
```

Note : Les procédures qui encodent l'interprétation des états d'un concept (comme State?(C, Ctxt), State@prc(C, Ctxt), etc.) sont définies dans l'annexe 3.

```
procedure SynergyInterpreter  is

    /*** LTMemory  and  WMemory are the two concepts that represent, respectively, the LTM and WM
            memories ***/
  LActiveCtxts := ( <" ", LTMemory.Value, nil, nil>,
                    <" ", WMemory.Value, ReferenceToTheFirstElemOfLActiveCtxts, nil> ) ;
  LNewActiveCtxts := nil ;
  LPrimitivesInExec := nil ;
  LAffectation := nil ;
  Lc? := nil ;
  Lcs? := nil ;
  LForward := nil ;
  LReceives := nil ;

  while IsActiveCG(WMemory.Value)  do :

            /*** Part 1 ****/
    for any element Ctxt in LActiveCtxts.Cdr  do :  /* Cdr returns the list without its first element */
      Insert in Lc? all the concepts Ci of Ctxt.ActiveCG with Ci.State = '?'  Or effState?(Ci, Ctxt) ;
      InsertAtEnd(Lc?, Lcs?) ;
    endFor  ;
    for any element C in LForward  do :
      {Remove(C, LForward) ;
       Ctxt := CtxtRecFor(ContextDef(C)) ;  /** ContextDef(C) returns the graph that contains C and
                                              CtxtRecFor(CG) returns a Ctxt that : Ctxt.ActiveCG = CG **/
       PropagateForward(C, Ctxt) }
    endFor  ;

    for any element Ctxt in LActiveCtxts.Cdr And its associated element Lc? in Lcs?  do :
      for any  element C in Lc?  do :  State?(C, Ctxt)  endFor;
      for any  concept C in  Ctxt.ActiveCG  with C.State = '@prc' do : State@prc(C, Ctxt)  endFor;
    endFor  ;

            /*** Part 2 ****/
    TestTerminationPrimitives ;
    TestTerminationCGs ;
    ExecReceives ;  /** manages the execution of "Receive" operations ***/

    for any element Ctxt in LActiveCtxts.Cdr  do :
      /*** we consider in the next "for" the special case of an active co-reference concept that its refeered
                                                                  concept is no more active ***/
```

```
            for any concept C in Ctxt.ActiveCG with : C.State = '!@' And IsCoReference(C.Referent, _)
                                                    And C.Value.State <> '!@' do :
                PostConditions(C, Ctxt) ;
            endFor;

            for any concept C in Ctxt.ActiveCG with C.State = '@aft' do : State@aft(C, Ctxt) endFor;
        endFor;

        AffectationManagment ;

                    /*** Part 3 ****/
        for any element <Ctxt1, C, LComputedValues, LArgsOut, D> in LPrimitivesInExec do :
            Replace( <Ctxt1, C, LComputedValues, LArgsOut, D> ,
                        <Ctxt1, C, LComputedValues, LArgsOut, D - 1> , LPrimitivesInExec) ;
                                                /*** 1 corresponds to one Synergy time-unit ***/
        endFor;

        LActiveCtxts := Append(LActiveCtxts, LNewActiveCtxts) ;
        Free(LNewActiveCtxts) ;
        Free(Lcs?) ;
    endWhile ;

end SynergyInterpreter.
```

Note : Le manuel de Synergy comprend une description plus complète et plus détaillée de l'algorithme.

3.6 "Syntaxe" de Synergy

Synergy est un langage graphique avec les GCs comme unique structure de base : un GC dans Synergy n'est pas décrit de façon linéaire et textuelle, selon une notation et donc une syntaxe particulière, mais plutôt de façon graphique avec des rectangles qui représentent les concepts et des flèches qui représentent les relations entre concepts.
La "syntaxe" ci-dessous (Figure 3.12) n'est donc pas celle d'une notation textuelle mais résume plutôt les éléments de base du langage, en particulier la composition d'un concept, d'une relation et de la mémoire à long terme. Cette dernière est un GC dont les concepts représentent des définitions, instances, schémas et synonymes des types. Ces structures conceptuelles sont définies dans la prochaine section.

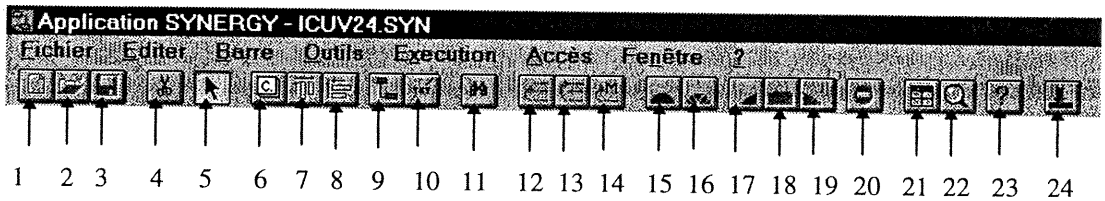
Nous utilisons la notation EBNF. [X] signifie que l'élément X est optionnel, X | Y signifie X ou Y, {X} signifie qu'on peut avoir une ou plusieurs occurrences de X, [X]* signifie qu'on peut avoir zéro ou plusieures occurrences de X, les () sont utilisées pour grouper des éléments et /** ..**/ pour les commentaires.

```
GC = { Concept -Relation-> Concept }.

Concept  = <Type, Referent, Valeur, Etat> . /** forme adoptée pour
                                  désigner la composition d'un concept **/
Type = ChaineDeCaractères .

Referent = IdentObj | ( Coreference [",/"] [",\"] ) .

Coreference = ( IdentObj "." ) | ( IdentObj {"." IdentObj} ) .

IdentObj = IdentInst | IdentVar .

IdentInst = ChaineDeCaractères qui ne commence pas par "_".

IdentVar = ChaineDeCaractères qui commence avec "_".

Valeur = Nombre | Booléen | Chaine | Fenêtre | GC | ListeDeValeurs.

Etat = "o" | "?" | "@prc" | "@prc@" | "@" | "!@" | "@aft" | "@nda" .


Relation = LienProcAttr | "next" | LienMem | AutreLien .

LienProcAttr = ( "in" | "out" | "grd" ) [",f"] [",/"] [",\"] .

LienMem = "sp" | "inst" | "scm" | "synm" .

AutreLien = ChaineDeCaractères .


MT = <_, _, GC, _> .  /*** Mémoire de travail est un concept.. **/

MLT = <_, RefMem, GC-Generalisation, _> . /** Mémoire à long terme **/

RefMem = IdentInst | "memory" .

GC-Generalisation = un GC composé de branches de la forme :

   Schema -sp-> Schema ,  Schema -sp-> DecType,
   DecType -scm-> Schema,  Instance -scm-> Schema,
   DecType -inst-> Instance, DecType -synm-> Synonymes.

DecType = Definition | Declaration .

Definition = <Type, IdentVar, GC, _> .

Declaration = <Type, _, _, _> .  /** le cas d'un type primitif **/

Instance = <Type, IdentInst, GC, _> .

Schema = <_, Ref, GC, _> .  /** le Ref est optionnel **/

Synonymes = <_, _, ListeDeTypes, _> .
```

Figure 3.12 : "Syntaxe" de Synergy

3.7 Environnement graphique de Synergy

L'environnement graphique de Synergy est un éditeur de GC qui permet de gérer des fichiers de GC, des GC (création, destruction, modification), l'activation/animation des GC, des contextes et des fenêtres. L'éditeur permet aussi de décrire et exécuter la(les) requête(s) et d'augmenter la mémoire à long terme avec la définition de nouveaux types et la description d'instances et de schémas. Cette section présente une vision globale de l'environnement graphique. La figure 3.13 montre les fonctionnalités de base de l'interface. La partie graphique concernant la formation et l'exploration de la mémoire à long terme est reportée au chapitre 6. Le manuel de Synergy [Kabbaj, 96] fournit une description plus complète et plus détaillée de l'environnement.



Légende :

1 : ouvrir un nouveau fichier 2 : ouvrir un fichier existant 3 : enregistrer l'application

4 : éliminer la partie sélectionnée 5 : mode commande 6 : icône pour concept

7 : aligner les concepts selon le haut 8 : aligner les concepts selon la gauche 9 : icône pour la relation

10 : déplacer le texte sur la relation 11 : zoom (primaire)

12 : mettre en focus la fenêtre du contexte père 13 : mettre en focus la fenêtre de la mémoire de travail

14 : mettre en focus la fenêtre de la mémoire à long terme

15 : activation directe 16 : activation pas à pas non contrôlée

17 : initier l'activation pas à pas avec contrôle du programmeur 18 : initier un autre cycle (un autre pas)

19 : compléter l'activation sans le pas à pas 20 : arrêt de l'activation

21 : organiser les fenêtres en mosaïque

22 : activer le "browser" pour localiser un concept dans la MLT 23 : aide (non disponible pour l'instant)

24 : activer le processus d'intégration de la connaissance en MLT

Figure 3.13 : Les fonctionnalités de base de l'interface

L'édition des fichiers L'icône 1 fournit un nouveau fichier "graphique" avec la hiérarchie de base comme contenu initial. Le programmeur peut alors ajouter ses propres types et/ou instances et schémas.

A tout moment, le programmeur peut sauvegarder son application (icône 3). Bien sûr, il peut aussi ouvrir une application existante (icône 2).

L'édition d'un concept Un concept est représenté par un rectangle dont la dimension dépend des champs du concept qui seront visibles. Pour créer un concept, le programmeur choisit l'icône 6, déplace le curseur à l'emplacement désiré et clique ensuite sur le bouton gauche de la souris, il obtient alors une fenêtre d'information lui permettant de spécifier le type, le référent, la valeur et l'état du concept (Figure 3.14). Le programmeur peut ignorer un champ particulier, la valeur par défaut est alors considérée. Si la valeur du concept est un GC, le programmeur doit cocher la case appropriée et décrire le GC par la suite.

Une fois le "Ok" sélectionné, le programmeur peut continuer avec un autre concept, retourner au mode commande (icône 5) ou, si la valeur du concept est un GC, garder le curseur sur le concept et "double-cliquer" sur le bouton droit de la souris, il obtiendra alors une fenêtre dans laquelle il pourra rentrer ou mettre-à-jour un GC (ou une collection de GC).

Le programmeur peut modifier le contenu, déplacer ou éliminer un concept. Le déplacement d'un concept implique le déplacement des relations qui lui sont connectées. Il en est de même pour l'élimination d'un concept.

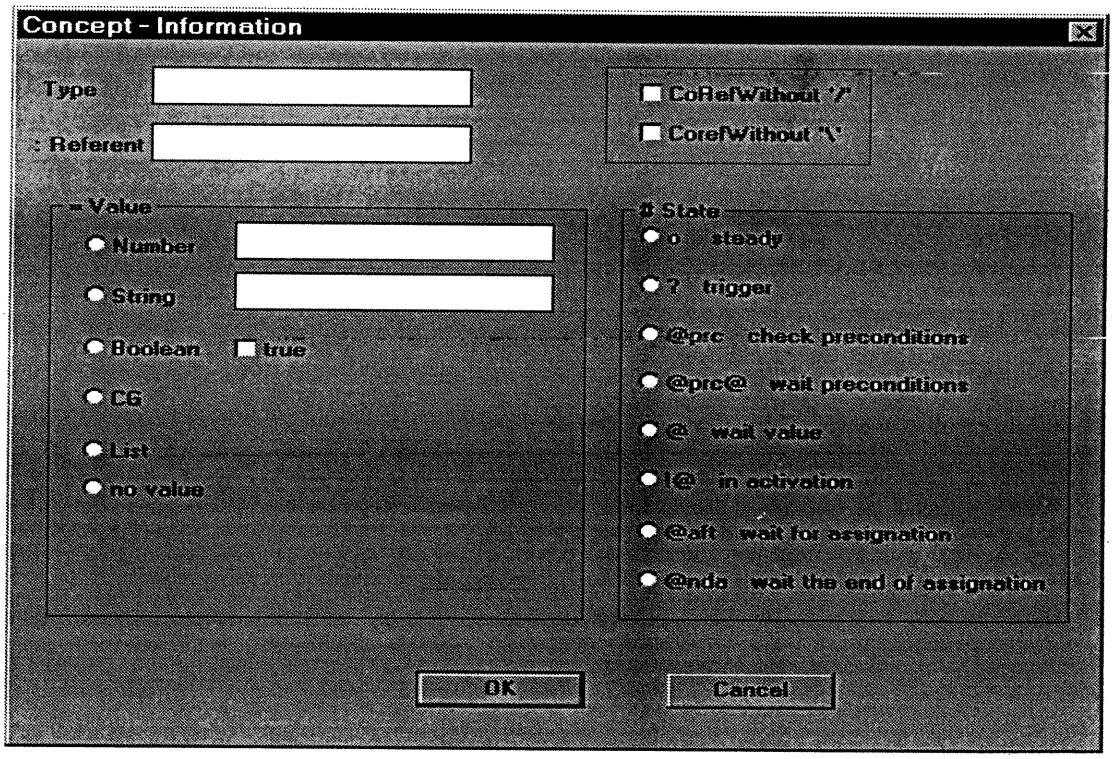


Figure 3.14 : La fenêtre-information pour le concept

Les champs “valeur” et “état” d'un concept ont une représentation graphique :
– Si la description du concept est un GC, alors un petit triangle rouge est dessiné au coin gauche-bas du rectangle décrivant le concept. Si la description est une valeur, le triangle est bleu. L'absence du triangle correspond à l'absence de description.
– Lorsqu'un concept est à l'état "?", le rectangle est vert. S'il est à l'état "@prc" ou "@prc@" (les deux représentent la phase *pré*condition), alors un triangle avec une surface noire est placé à *gauche* du rectangle. S'il est à l'état "@" (qui indique aussi une forme de précondition), alors un triangle avec une surface blanche est placé à gauche du rectangle. S'il est à l'état "en-activation ; !@", alors le rectangle est rouge.
En tant que *post*-conditions, l'état "@aft" est représenté par un triangle avec une surface noire placé à *droite* du rectangle et l'état "@nda" est représenté par un triangle avec une surface blanche placé à droite du rectangle.

L'édition d'une relation En choisissant l'icône 9, le programmeur peut relier par une relation deux concepts existants, en cliquant sur le premier et ensuite sur le deuxième. Si le programmeur veut tracer un lien composé de plusieurs segments, il suffit de cliquer aux endroits désirés pour fixer les segments au fur et à mesure. En cliquant sur le concept cible de la relation, le système le reconnaît et donc identifie la fin du lien.
Une fois le lien tracé, une fenêtre-information est ouverte permettant au programmeur de décrire la relation (Figure 3.15). Le nom de la relation est ensuite placé sur le lien. Le programmeur peut déplacer le nom à l'endroit désiré sur le lien en choisissant l'icône 10. Pour consulter ou modifier les informations relatives à une relation, il faut se positionner sur la relation et cliquer sur le bouton gauche de la souris.

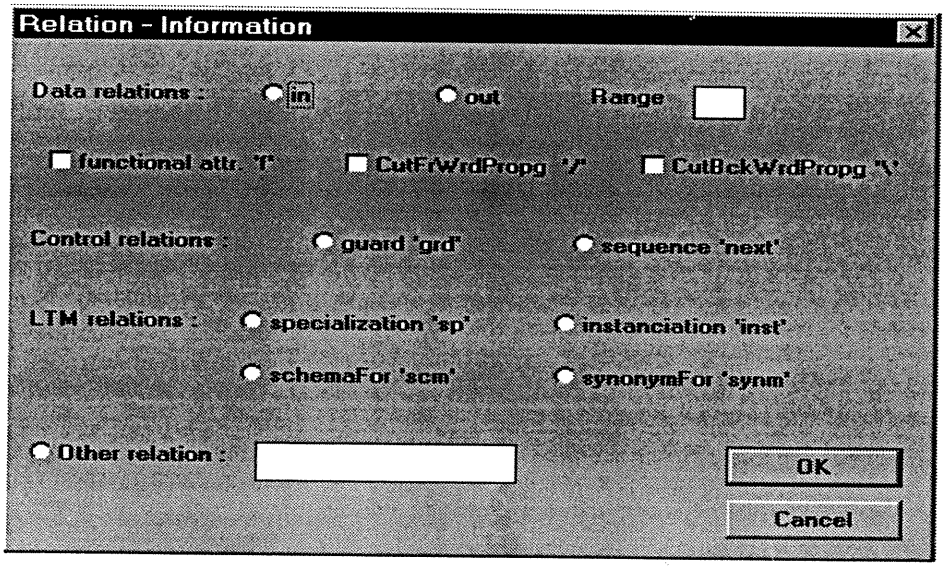


Figure 3.15 : La fenêtre-information pour une relation

L'édition d'un GC Plusieurs fenêtres peuvent être ouvertes en même temps et le programmeur peut travailler sur les différents graphes qui s'y trouvent. Trois icônes (12 à 14) permettent le changement de focus et la traversée de la hiérarchie des contextes actifs : ils permettent au programmeur de se placer au niveau du contexte père, de la mémoire de travail ou de la mémoire à long terme.
Une partie d'un graphe, comprenant des concepts et des relations, peut être sélectionnée et ensuite déplacée.

Activation de l'interpréteur Trois modes d'activation sont offerts au programmeur (icônes 15 à 19) : activation directe (icône 15), activation pas à pas sans le contrôle du programmeur (icône 16) et activation pas à pas avec le contrôle du programmeur (icônes 17 à 19). Cette dernière est très utile pour la correction (debug) ; à chaque cycle (un pas) de l'interpréteur, le contrôle est transmis au programmeur qui peut consulter les valeurs des concepts et explorer l'espace dynamique composé des GC actifs. Il peut ouvrir plusieurs fenêtres afin de suivre l'exécution parallèle de différents GC, il peut fermer d'autres fenêtres, etc. Le programmeur peut ainsi suivre, de près et au besoin, l'exécution de son "programme". Enfin, il peut toujours changer ce mode, afin de terminer rapidement l'exécution (icône 19) ou arrêter l'exécution (icône 20).

3.8 Résumé

Nous avons présenté dans ce chapitre les éléments de base de Synergy, un langage graphique, parallèle et multi-paradigme basé sur l'activation des GC. Une application Synergy est composée d'une mémoire à long-terme (MLT) et d'une mémoire de travail (MT). La première représente une base de connaissances décrite avec des GC et la seconde fournit un espace de travail où l'utilisateur peut spécifier ses requêtes, formulées aussi avec des GC. Synergy se base sur un modèle "simple" et uniforme pour représenter *et* activer les connaissances procédurales : *tout est formulé en GC et tout traitement est basé sur l'activation des GC*. Un concept peut avoir un type primitif ou défini et le type peut correspondre à une entité "activable" (par exemple, une opération ou un processus) ou "statique" (par exemple, un entier ou un booléen). L'activation d'un concept peut dépendre de certaines conditions, formulées à l'aide de relations procédurales qui permettent par ailleurs de propager l'activation dans le graphe.
Nous avons montré par ailleurs, les différents mécanismes d'abstraction offerts par le langage ainsi que le mécanisme d'activation parallèle d'un GC. le parallélisme est réalisé grâce (en partie) aux notions d'état et de cycle de vie d'un concept.

Les deux prochains chapitres montrent comment, avec la dimension dynamique des GC, on peut "programmer" plusieurs types de connaissances.

Chapitre 4

Programmations fonctionnelle, procédurale, par accès et par événement en Synergy

Nous présentons dans ce chapitre des exemples de "programmes" Synergy qui illustrent différents modèles de programmation, en l'occurrence les modèles fonctionnel, procédural, par accès et par événement. Pour le premier, nous illustrons la formulation d'une expression, avec son évaluation par chaînage avant et/ou arrière, d'une fonction récursive et d'une fonction paresseuse. Le second modèle est illustré en formulant en Synergy les structures de contrôle de base (la séquence, l'alternative et l'itération). La programmation par accès est ensuite illustrée en formulant les notions de "frame", "d'attachement procédural" et d'activation par accès. Un exemple spécifique concrétise cette formulation.
Enfin, la programmation par événement en Synergy est illustrée par deux exemples.

4.1 Programmation fonctionnelle en Synergy

L'expression constitue le concept de base en programmation fonctionnelle. La figure 4.1 montre la formulation d'une expression en graphe de dépendance fonctionnelle (GDF) et en Synergy.
Un GDF est un graphe dont les nœuds représentent des fonctions (primitives et/ou définies) et les liens les dépendances entre les fonctions (un argument en sortie d'une fonction est généralement un argument en entrée d'une autre fonction). Une fonction est déclenchée par la présence de ses arguments en entrée, la fonction consomme ces derniers, s'exécute et produit ensuite des données en sortie qui déclenchent d'autres fonctions et ainsi de suite, produisant un flot de données.
Dans la formulation Synergy, nous spécifions explicitement que les liens de données sont fonctionnels (avec l'attribut "f") afin de respecter la sémantique du modèle fonctionnel et d'annuler l'interprétation procédurale de ces liens, considérée par défaut. Toutefois et comme le montre l'exemple 1 ci-dessous (Figure 4.2), l'interprétation procédurale des relations de données peut également être utilisée dans la formulation d'une expression.

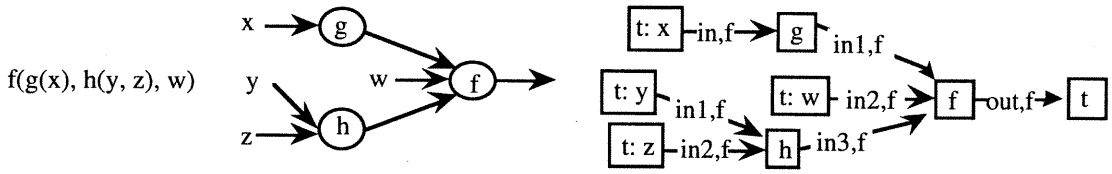


Figure 4.1: Formulation d'une expression en GDF et en Synergy

Comme on peut le noter, le parallélisme implicite dans la formulation de l'expression est rendu explicite dans les deux graphes : les fonctions g et h peuvent s'exécuter en parallèle, la fonction f doit attendre toutefois leur exécution (on suppose que les fonctions sont strictes) car deux de ses arguments en entrée sont des arguments en sortie de g et h.

Dans les GDF comme dans Synergy, l'activation peut être réalisée par chaînage avant et/ou arrière; la première est dirigée par la disponibilité des données et la seconde par la demande de "calcul" des données. Dans notre exemple ci-dessus, la disponibilité de x, y et z déclenche g et h et ensuite la disponibilité de w et des résultats intermédiaires des deux fonctions déclenche la fonction f. Par ailleurs, la demande de la valeur de la fonction f déclenche cette dernière, puis les fonctions qui fournissent à f ses données en entrée et ainsi de suite.
Cette "propagation stricte" est nuancée par la possibilité d'une évaluation paresseuse de certaines fonctions : si une fonction f est paresseuse et qu'elle est déclenchée par une demande, elle ne propage pas la demande à ses arguments en entrée ; elle commence directement son exécution et les valeurs des arguments seront demandées au besoin.

Remarque : Les graphes de dépendance fonctionnelle, appelés aussi graphes de flot de données, sont une formulation orientée graphe du modèle fonctionnel [Davis et Keller, 82].

Exemple : *Résolution d'une équation du second degré*
Considérons comme exemple, les solutions x1 et x2 pour une équation du second degré $Ax^2 + Bx + C$: $x1 = (-b + \sqrt{(B*B - 4*A*C)})/2*A$, et $x2 = (-b - \sqrt{(B*B - 4*A*C)})/2*A$.
La définition du type Eq2D (Figure 4.2) est un GC qui intègre ces deux équations.
La figure 4.3 décrit deux appels à Eq2D, un avec déclenchement en avant et l'autre avec un déclenchement en arrière.

→ Considérons le premier cas (Figure 4.3a) : le concept [Eq2D] est déclenché car ses trois arguments en entrée sont à l'état "?" (il aurait suffit qu'un seul le soit pour qu'il y ait déclenchement). Puisque tous ses arguments en entrée sont disponibles et qu'aucune condition supplémentaire n'est requise, le concept [Eq2D] est activé immédiatement, ses paramètres vont référer aux arguments correspondants et ses trois paramètres en entrée ([Int:in1/_A], [Int:in2/_B] et [Int:in3/_C]) auront l'état "?". L'activation du corps de [Eq2D] va donc

commencer à partir de ces trois concepts (Figure 4.2). Ces derniers vont alors susciter, par une propagation en avant via les liens "in", le déclenchement et ensuite l'activation parallèle des concepts de multiplication (*) qui les ont en entrée. Ces multiplications vont produire des résultats qui déclencheront et activeront d'autres concepts-fontions et ainsi de suite jusqu'à ce qu'on obtienne les valeurs des paramètres x1 et x2.

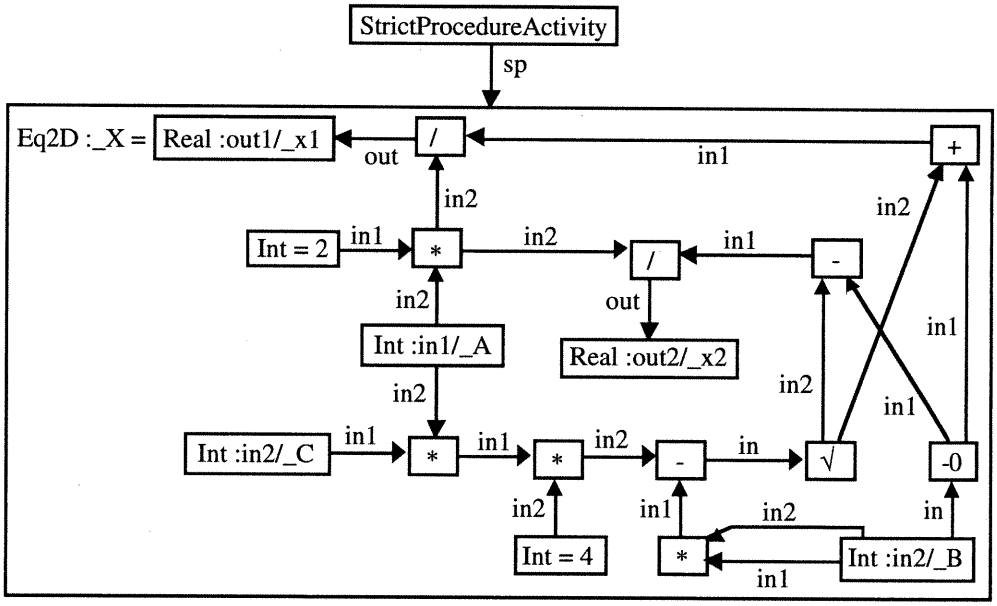
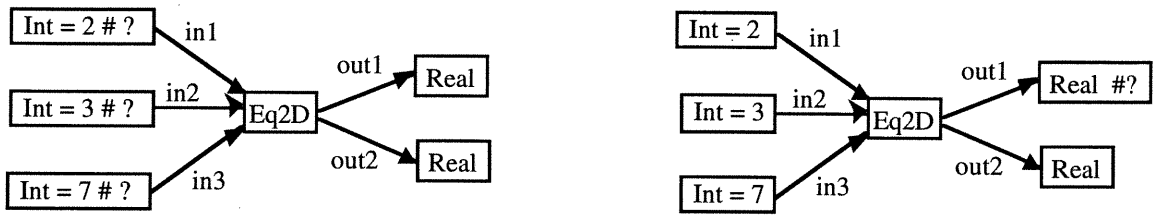


Figure 4.2 : Définition de Eq2D

→ Considérons le second cas (Figure 4.3b) : le concept [Eq2D] est déclenché car l'un des arguments en sortie (x2) est à l'état "?". Le concept [/] propage cette demande en arrière à ses deux arguments en entrée [*] et [-] (puisqu'ils n'ont pas de valeur), le concept [*] peut être activé mais le concept [-] ne le peut pas (Figure 4.2), il propage donc lui aussi la demande à ses deux arguments en entrée et ainsi de suite. Pendant que la propagation se réalisera en parallèle, [*] aura terminé son activation et [/] attendra alors son second argument. La propagation arrière de la demande génère des attentes et, au fur et à mesure que les concepts-fonctions s'exécutent, les attentes se réduisent.

→ Si dans la Figure 4.3b, le premier argument en entrée de [Eq2D] était à l'état "?", alors on aurait une activation "mixte" du corps de [Eq2D] ; des parties du graphe activé seraient animées par chaînage-avant alors que d'autres le seraient par chaînage-arrière, les deux se feront en parallèle.



(a) déclenchement par avant (b) déclenchement par arrière

Figure 4.3 : Activations du concept [Eq2D]

Exemple : *définition d'une fonction récursive*

Cet exemple montre la définition récursive du type Factorielle pour un entier non nul :

Factorielle(1) = 1

Factorielle(N) = N * Factorielle(N - 1) , pour N <> 1.

Dans la formulation en Synergy (Figure 4.4) et en raison du parallélisme inhérent au langage, on contrôle les propagations avant et/ou arrière pour avoir un comportement "correct" et conforme à la définition de l'opération. Par exemple, en donnant une valeur à N, le calcul de "N - 1", de Factorielle(N - 1) et de "N * Factorielle(N - 1)" ne doit se faire qu'après avoir vérifié que N <> 1.

Ces contraintes sont réalisées en inhibant, avec l'attribut "/", la propagation avant du concept [Int: in/_N] vers les concepts [-] et [*] (Figure 4.4).

De même, si le résultat de "Factoriel" est demandé, initiant ainsi une propagation arrière qui déclencherait [*] (Figure 4.4), ce dernier ne doit être déclenché que si la condition N <> 1 est vérifiée. On doit donc inhiber la propagation arrière, via le lien "out" en utilisant l'attribut "\". Ainsi, c'est uniquement le lien "grd" qui peut déclencher le concept [*] (Figure 4.4).

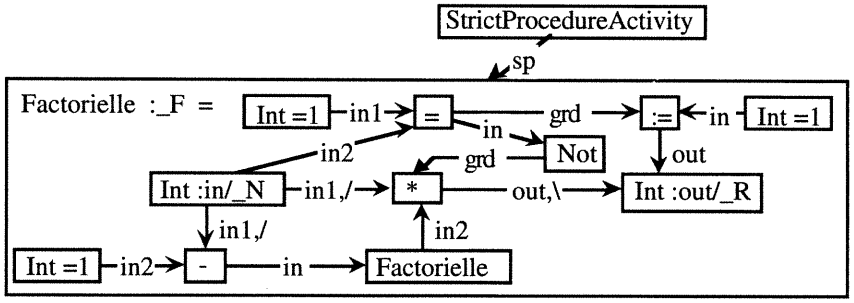
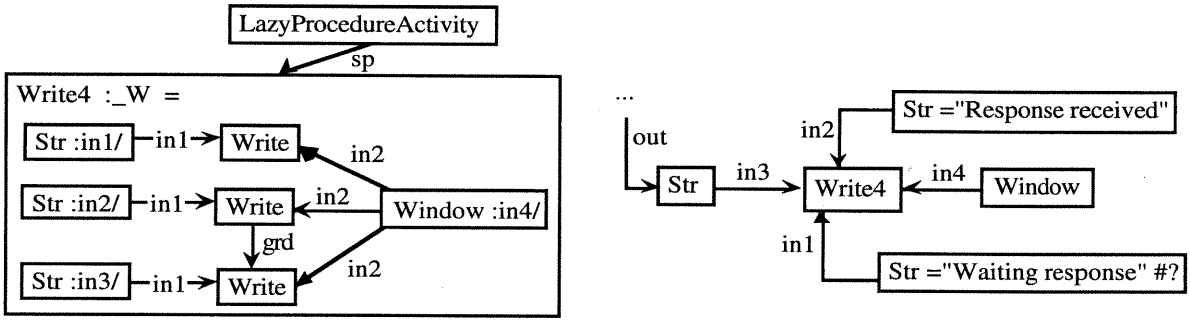


Figure 4.4 : Définition récursive de Factorielle

Exemple : *définition d'une fonction paresseuse*

Considérons la fonction paresseuse "Write4", définie dans la Figure 4.5a et son activation avec l'appel de la figure 4.5b : l'argument "in1" de Write4 déclenche ce dernier qui est

directement activé; Write4 ne demande pas les valeurs de ses autres arguments en entrée. Write4 écrit le message "Waiting response" et termine son exécution. Par la suite, lorsque son argument "in3" reçoit une valeur, Write4 est déclenché et activé de nouveau. Son traitement consiste cette fois-ci à écrire la valeur de son argument "in3", une fois le message "Response received" écrit. La fonction Write4 termine ensuite sa seconde exécution.



(a) (b)

Figure 4.5 : Définition paresseuse d'une fonction

4.2 Programmation impérative (procédurale)

Considérons la formulation en Synergy de quelques exemples d'instructions de contrôle (la séquence, l'alternative et l'itération). D'autres exemples sont fournis dans les prochaines sections.

Exemple : *Les cas de la séquence et de l'alternative*

```
/*** On suppose que les variables A, Ind, Prod et Som ont déjà des valeurs ***/
   if A = 0 then
          {Ind := Ind - 1 ;
           Prod := Prod * Ind }
   else
          {Ind := Ind + 1 ;
           Som := Som + Ind }
   endIf.
```

Nous avons deux blocs d'instructions, un est contrôlé par la condition A = 0 et l'autre par sa négation. Le GC ci-dessous rend explicite cette distinction (Figure 4.6). Notons par ailleurs que l'exécution du code procédural ci-dessus est dirigée par le flot de contrôle et non par le flot de données. La formulation en Synergy (Figure 4.6) est conforme à cette "sémantique". En effet, l'exécution du graphe commence avec le concept test [= #?] et, selon que (A = 0) ou (Not A = 0), l'un des deux "guards" déclenchera la première action du bloc associé. Supposons que A = 0, le concept [-] est donc déclenché, activé et il en est de même ensuite pour [*]. Ces deux opérations effectuent une *mise à jour* d'une donnée ("Ind" pour [-] et

"Prod" pour [*]) ; elles ont un argument en entrée qui est aussi un argument en sortie. Pour annuler une activation cyclique de l'opération (et autre effets de bord) suite à la mise à jour, le "out" possède l'attribut "/", la nouvelle valeur est donc affectée au concept en sortie sans déclencher ce dernier.

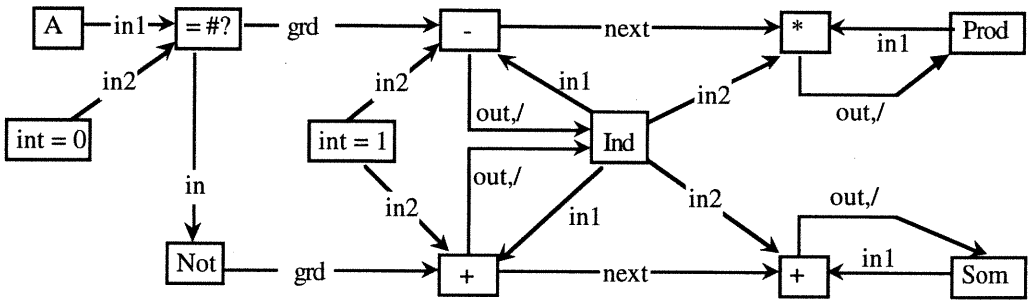


Figure 4.6 : Formulation en Synergy d'une alternative

Exemple : *itération*

*/*** Lire une série d'entiers et retourner le plus grand. La terminaison de la série est indiquée par 0 ***/*

{MaxCour := 0 ;
read NouvVal ;
While NouvVal <> 0 **do**
{**if** NouvVal > MaxCour **then** MaxCour := NouvVal **endIf** ;
read NouvVal }
endWhile.

Comme pour l'exemple précédent, c'est le flot de contrôle qui est mis en valeur dans la formulation en Synergy (Figure 4.7). L'activation du GC commence avec l'opération [Read], la nouvelle valeur pour "NouvVal" déclenche le test [<>] mais non le test [>] et l'opération [:=] (Figure 4.7). Le test [<>] est donc déclenché par le flot de données. Si le test [<>] réussit, l'opération [>] est déclenchée et activée et si ce second test réussit alors l'affectation est effectuée, suivie de l'action [True] (qui est une action "neutre"). Cette dernière est activée également si le test [>] échoue. L'action [True] constitue le point de rencontre des deux branches du "if". Après l'exécution de [True], l'opération [Read] est déclenchée et activée, produisant une nouvelle valeur pour "NouvVal", ce qui déclenche de nouveau le test [<>].

Note : Dans le graphe de la Figure 4.7, nous avons utilisé l'opération primitive "True" pour montrer la configuration générale de la séquence "Act1, if-then-else, Act3". Si nous considérons toutefois les particularités de l'exemple, la configuration Figure 4.8a qui est présente dans le GC de la Figure 4.7 peut être remplacée par celle présentée dans Figure 4.8b.

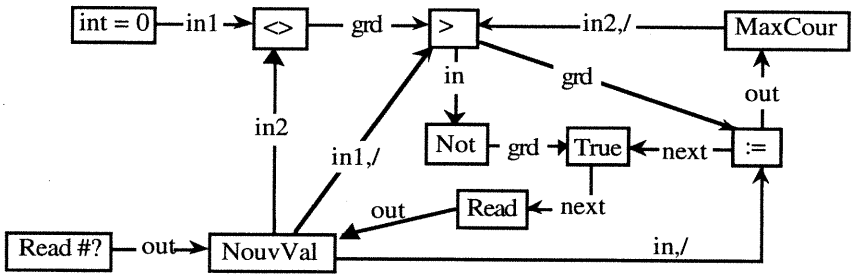


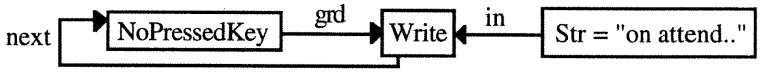
Figure 4.7 : Formulation en Synergy d'une itération

(a) (b)

Figure 4.8 : (b) est une formulation plus concise que (a)

Dans certains cas, la condition d'arrêt d'une itération n'est pas affectée par le corps de l'itération. Par exemple :

```
while No PressedKey do
   writeln("on attend...") ;
end ;
```



4.3 Programmation par accès

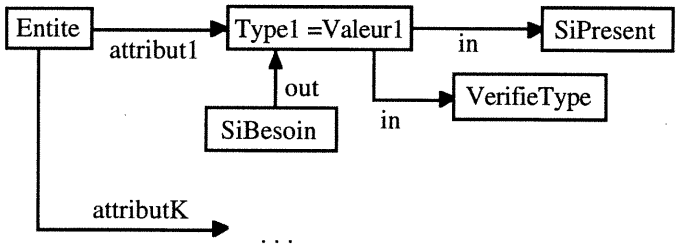
La programmation par accès a été introduite avec la notion de "frame" et ensuite avec les langages orientés frames [Masini et al., 90]. Un cadre ("frame") est composé généralement d'attributs ("slots"), chacun peut avoir plusieurs facettes ("facets"), une facette peut être "statique" (comme le type ou la valeur par défaut) ou "dynamique" (comme si-present, si-besoin, verifieType) ; la valeur d'une facette dynamique correspond alors à un traitement :

```
(entite
     (attribut1
              (facette11  valeur11)
              (facette12  valeur12)
              ... )
     (attribut2
              (facette21  valeur21)
              (facette22  valeur22)
              ... )
     ...)
```

Parmi les facettes dynamiques, citons les trois types suivants : 1) *si-présent* (si la valeur de l'attribut vient d'être calculée alors exécuter le traitement associé à "si-présent"), 2) *si-besoin* (si la valeur de l'attribut est demandée alors exécuter le traitement associé à "si-besoin") et 3) *verifieType* (exécuter le traitement si l'attribut reçoit une nouvelle valeur).

A noter que l'exécution d'une facette dynamique d'un attribut peut changer ou demander la valeur d'un autre attribut du cadre courant ou d'un autre cadre, constituant ainsi un graphe de dépendance sous-jacent aux cadres "actifs".

En Synergy, un cadre (frame) peut être représenté par un GC et les facettes dynamiques par des concepts actifs :

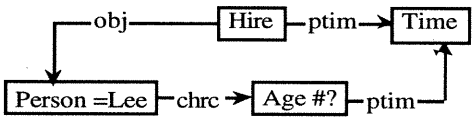


Les facettes si-present et verifieType sont déclenchées par chaînage avant : la valeur d'un attribut est un argument en entrée à si-present et à verifieType, une mise à jour de la valeur va donc les déclencher. La facette si-besoin est déclenchée par chaînage arrière : la valeur d'un attribut est un argument en sortie et une demande de valeur mettrait si-besoin à l'état "?".
Pour illustrer la programmation par accès (ou orientée-cadre) en Synergy, nous allons considérer l'exemple proposé par Sowa concernant un système de question/réponse pour une base de données [Sowa, 84], en l'interprétant comme une application à base de cadres.

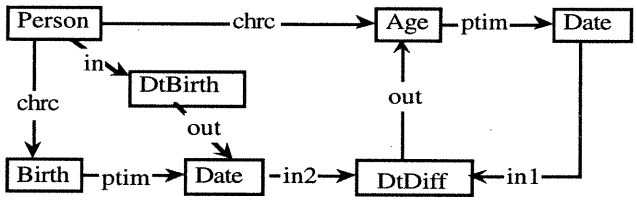
Exemple : *système de question/réponse basé sur une programmation par accès*
La sémantique de la base de données est décrite par un ensemble de schémas (qui sont similaires aux cadres), un schéma contient en général des informations structurales et procédurales (en particulier les dépendances fonctionnelles entre les attributs d'une relation). Sowa représente le second type d'information par des acteurs (fonctions) qui peuvent être déclenchés par avant (disponibilité des arguments en entrée) ou par arrière (demande de la valeur d'un argument en sortie). Les acteurs dans les schémas jouent le rôle de facettes dynamiques dans les cadres. En Synergy, les acteurs sont aussi des concepts (une fonction est un concept, au même titre que le concept "table" ou "personne").
Le système de question/réponse peut être décrit comme suit :

{wg := qg ; /*** *qg correspond au graphe de la question et wg au graphe de travail* ***/
tantQue il y a dans wg une valeur à déterminer qui ne peut l'être par les acteurs présents dans wg **faire**
sg := choisirSchema(wg) ; /*** *chercher un schéma qui puisse répondre (partiellement ou en totalité) à cette demande* ***/
specialise(wg, sg) ; /*** *joindre à wg le schéma sg* ***/
interpréter les acteurs de wg ; /*** *considérer les acteurs à déclencher et à activer* ***/
finTantQue }

Le système effectue donc une programmation par accès et exploite une base de schémas (de cadres) pour composer de façon dynamique le graphe qui permettra le calcul de la réponse. L'interprétation des acteurs ainsi que la dynamique sous-jacente à la programmation par accès sont réalisées directement par Synergy. Considérons cela à l'aide de l'exemple suivant. Soit la question suivante : "What was Lee's age when hired ?". Elle peut être traduite en Synergy comme indiqué dans la Figure 4.9a (l'exemple est extrait de [Sowa, pp. 314-317, 84], nous l'avons reformulé en Synergy tout en conservant sa formulation en anglais). Le système va ensuite établir une jointure de la question au schéma présenté dans la Figure 4.9b. Le résultat est présenté dans la Figure 4.9c.



(a) Question



(b) Schéma

(c)

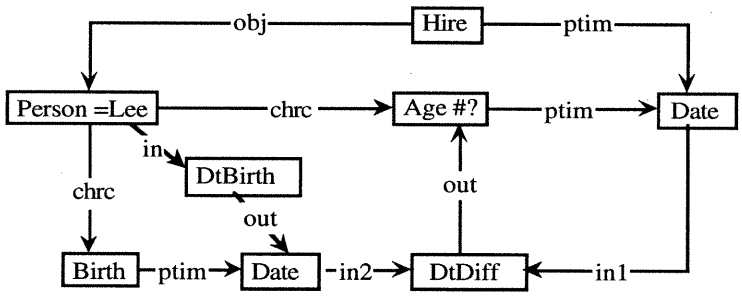


Figure 4.9 : Résultat après la jointure

Dans le graphe ci-dessus, le concept [DtDiff] correspond à un acteur et joue le rôle de la facette "si-besoin" pour Age. En effet, la valeur de Age est demandée et cela déclenche [DtDiff]. Ce dernier propage la demande à ses deux arguments en entrée. Le concept [DtBirth] est alors déclenché ; il joue le rôle de la facette "si-besoin" pour "Date" de "Birth". Après son exécution, "Date" aura une valeur, mais la valeur de la "Date" de "Hire" ne pouvant être déterminée à partir du graphe courant, il faudra donc chercher un schéma qui fasse avancer le traitement. Ce dernier va se poursuivre selon l'algorithme décrit ci-dessus.

Notons que le graphe de la figure 4.9c peut être utilisé autrement : si, au lieu du concept [Person =Lee], nous avons [Person =Jo #?] et, au lieu du concept [Age #?], nous avons [Age], alors le traitement consistera à déterminer ce qui peut être inféré à propos de "Jo" selon le graphe ci-dessus. Le concept [DtBirth] va jouer dans ce cas le rôle de la facette "si-present" pour [Person =Jo #?] et déclenchera un chaînage-avant dans le graphe de travail. En effet, l'exécution

de [DtBirth] va produire une valeur pour "Date" de "Birth", qui déclenchera [DtDiff], considéré alors comme la facette "si-présent" pour "Date".
Ainsi, selon le contexte d'exécution, un acteur (un concept actif en Synergy) peut jouer le rôle de facette "si-besoin" ou "si-present".

4.4 Programmation par événement

La programmation par événement est basée sur le principe qu'un traitement est déclenché suite à un événement. Les événements peuvent survenir de l'extérieur et/ou résulter de l'exécution d'un traitement. Ce type de programmation est très utilisé en programmation d'interfaces, en multi-média et en programmation temps-réel où les événements extérieurs sont généralement saisis par des capteurs.

Les deux exemples suivants illustrent l'utilisation de Synergy pour une programmation par événement. Le troisième exemple illustre la formulation en Synergy d'un diagramme de transition d'états, utilisé souvent dans des applications orientées événements.

Exemple : *système de chauffage d'une chambre*

→ *Enoncé :*

L'exemple concerne une version simplifiée du système de chauffage d'une chambre. Cette dernière possède une température avec une valeur initiale 15, renferme un thermostat et un chauffage. Le thermostat possède un contrôleur initialisé aussi à 15 et le chauffage possède un état initialisé à "off".
Considérons maintenant le mécanisme de chauffage : si la température de la chambre est inférieure à la valeur du contrôleur, l'état du chauffage est mis à "on", ce qui déclenchera une augmentation continue de la température. Cette augmentation est interrompue dès que la température est égale à la valeur du contrôleur, le chauffage se mettra alors à l'état "off".

→ *Formulation de l'exemple en Synergy :*

La situation est décrite comme un schéma indexé sous les quatre types : Temp, Chambre, Chauffage et Thermostat (Figure 4.10). Le schéma comprend une partie descriptive (description de la chambre) et une partie "procédurale" qui correspond au mécanisme de chauffage d'une chambre.
Le traitement est par événement : un changement de température (c'est l'événement) déclenche le concept-fonction [VerifieTemp] qui décide si le chauffage doit être à "on" ou à "off". Si le résultat de [VerifieTemp] est égal à "on", le concept-test [=] produit "true" et le lien "grd" va donc déclencher l'augmentation de la température. Le changement de cette dernière déclenche de nouveau [VerifieTemp] réalisant ainsi le cycle qui permettra une augmentation

continue de la température jusqu'à ce que la température soit égale à la valeur du contrôleur, le concept [VerifieTemp] retourne alors la valeur "false" et le test [=] produira "false", dans ce cas le lien "grd" ne déclenchera pas l'incrémentation de la température.
Notons que la mise à jour de la température ne doit pas déclencher le concept [+]. Cette interdiction de propagation est réalisée par l'attribut "/" qui est ajouté au "in1" de [+].

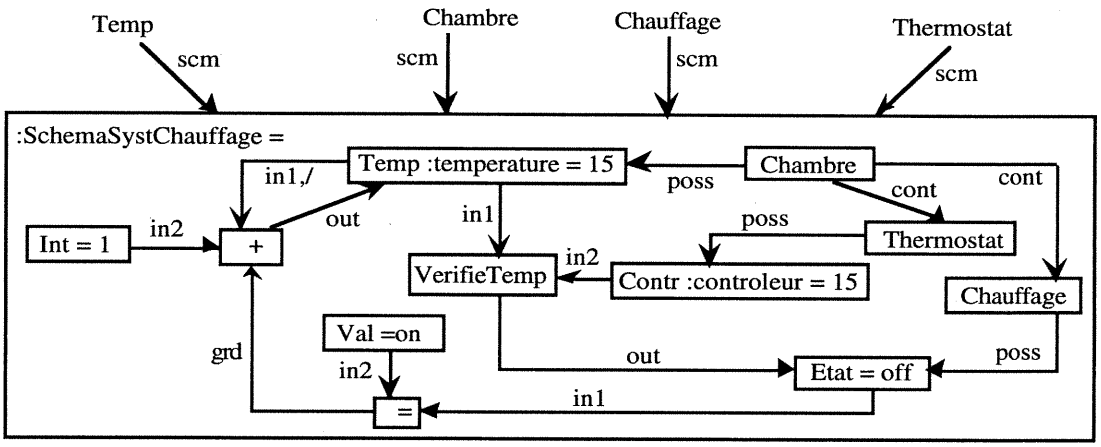


Figure 4.10 : Le schéma pour le système de chauffage d'une chambre

Définition du type VerifieTemp (Figure 4.11) : si la température est inférieure à la valeur du contrôleur alors affecter "on" à l'état, sinon affecter "off".

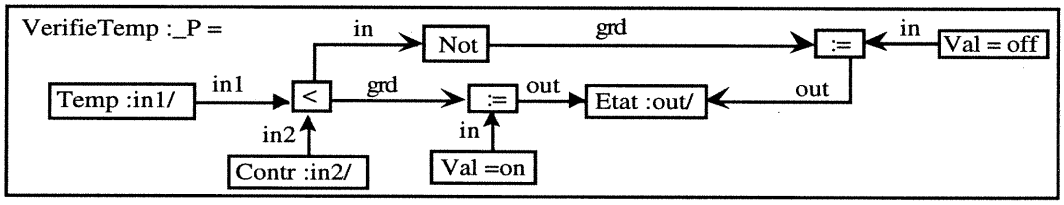


Figure 4.11 : Définition du type VerifieTemp

Une activation du schéma "SchemaSystChauffage" peut résulter de l'événement suivant : "la température de la chambre 34 est descendue à 10". Cette phrase peut être reformulée en Synergy comme suit : [Chambre :ch34]-poss->[Temp]<-out-[:=]<-in-[Val = 10 #?].
Pour analyser en profondeur cette "proposition" et considérer les conséquences possibles sur l'environnement modélisé, un système de compréhension et d'inférence cherchera un schéma de la mémoire, le schéma devrait décrire un comportement suscité par un changement de température d'une chambre. Le schéma "SchemaSystChauffage" satisfait la condition. Une fois trouvé, le système effectuera une jointure du schéma et de la proposition, le résultat sera alors considéré comme un "modèle exécutable" qui simulerait le changement de

l'environnement (la chambre dans notre exemple) suite à l'événement "la température de la chambre #34 est descendu à 10".

En résumé, l'exemple illustre, brièvement, l'avantage de représenter en GC les connaissances procédurales : les schémas peuvent contenir des connaissances procédurales et, les opérations sur les GC peuvent construire une tâche (ou un scénario) par composition "dynamique" des schémas, des tâches déjà construites et/ou d'autres structures. Synergy permet de plus, l'interprétation et la simulation de la(les) tâche(s) ainsi construite(s).

Exemple : *Base de connaissances à temps-réel*

→ *Enoncé :*

L'exemple concerne la gestion d'événements selon des contraintes temporelles, dans le cadre d'une application à temps-réel (adaptée de [Lanusse et Terrier, 92]). Cette gestion des événements est formulée comme un "graphe temporel" (figure 4.12) : "Le processus en cours doit attendre durant 2 minutes pour l'événement "augmentation du flot" et ensuite ou bien après que 3 heures se soient écoulées, il doit attendre au maximum 2 minutes pour l'événement "interruption de l'infusion" ou bien, après que 7 heures se soient écoulées, il doit attendre au maximum 5 minutes pour l'événement "arrêt de la pompe". .... ".

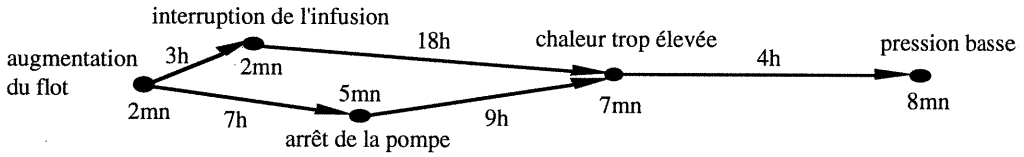


Figure 4.12 : Exemple de graphe temporel

Gérer un tel graphe temporel consiste à simuler le processus d'activation selon le flot d'événements et les contraintes temporelles. Traverser le graphe produit une liste comportant les événements qui sont survenus durant le processus d'activation. La liste des événements correspond à l'historique du processus et son contenu dépend du chemin qui mène du nœud racine au nœud final. Si un événement ne survient pas dans la période prévue, le chemin qui mène à l'événement suivant est abandonné.

Pour implanter un tel gestionnaire de graphe temporel, Lanusse et Terrier [Lanusse et Terrier, 92] ont utilisé un langage de la famille des langages d'acteurs [Agha et Hewitt, 87]. Une implantation "style" acteurs peut être réalisée également en Synergy, nous considérons toutefois ici une implantation orientée événement.

→ *Formulation de la solution en Synergy*

Le GC ci-dessous (Figure 4.13) est une reformulation du graphe temporel initial (figure 4.12) rendant explicite certaines informations comme le "gestionnaire", représenté par le

concept [C] qui réalise le processus d'attente et de reconnaissance d'événement selon les contraintes temporelles. Le gestionnaire est défini comme une fonction qui retourne une liste d'événements.

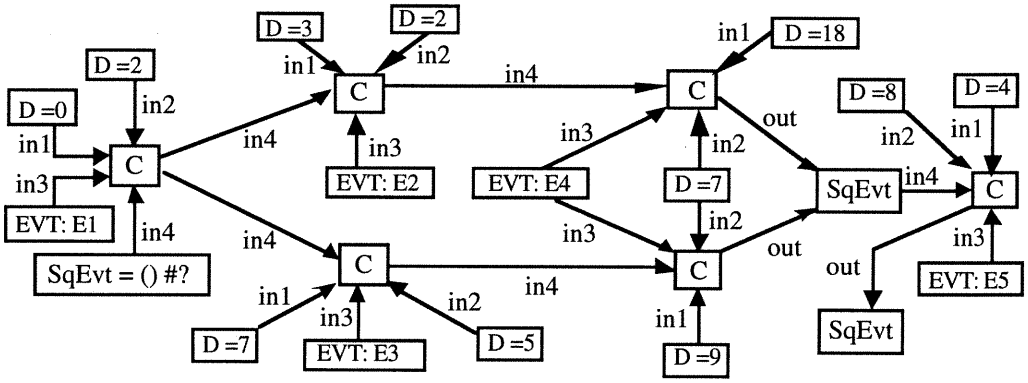


Figure 4.13 : Formulation en GC du graphe temporel

L'activation du graphe commence avec le gestionnaire [C] de l'événement E1, le concept [C] est déclenché à cause de son quatrième argument qui est initialisé à l'état "?". Le concept [C] peut rester actif durant toute la période spécifiée par son second argument (ici 2 minutes). Si l'événement E1 (augmentation du flot) ne survient pas durant cette période, alors [C] termine son activation sans produire de résultat et l'activation de tout le graphe est terminée.
Si toutefois, l'événement E1 survient durant la période spécifiée, [C] termine son exécution, produit en sortie la liste (E1) et déclenche ensuite les deux concepts [C] qui lui sont liés, l'un va attendre durant 3 heures et l'autre pour 7 heures. Supposons qu'après 3 heures et 1 minutes l'événement E2 survienne, le concept [C] associé va terminer son exécution avec comme résultat (E1, E2) et il va déclencher le concept [C] pour lequel il est une entrée. Supposons maintenant qu'après 18 heures et 7 minutes, l'événement E4 (chaleur très élevée) ne soit pas survenu, son gestionnaire [C] va alors terminer l'exécution sans produire de résultat, mettant ainsi fin à cette propagation en avant du calcul de la liste des événements.

→ *Définition du gestionnaire C* (Figure 4.14)

Le gestionnaire réalise les deux contraintes temporelles qui portent sur la perception d'un événement : ce n'est qu'après une durée D1, qu'il peut s'attendre à l'événement en question, ce dernier peut survenir uniquement durant la période D2.
Dans la définition de C, le concept [HorlogeReg] s'occupe des deux attentes et le concept [Capteur] s'occupe de la perception de l'événement.
Considérons maintenant la synchronisation entre les deux concepts : le concept [Capteur] est déclenché par une nouvelle valeur de la liste des événements et puisqu'il est strict, il ne peut être activé immédiatement étant donné que son argument en entrée [Bool:B1] n'a pas de valeur.

Le concept [Capteur] mettra donc cet argument à l'état de demande et il se mettra lui à l'état d'attente.

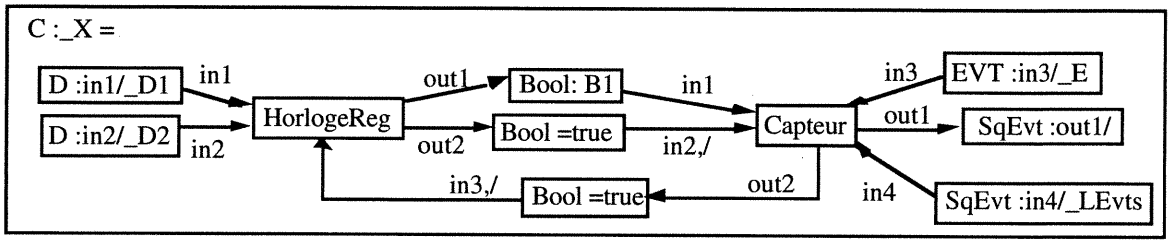


Figure 4.14 : Définition du gestionnaire C

Le concept [Bool:B1] déclenche le concept [HorlogeReg] qui peut commencer son exécution puisque toutes ses conditions sont vérifiées. Après une période de temps D1 (la première attente), [HorlogeReg] affectera la valeur "true" au concept booléen [Bool:B1] et [HorlogeReg] *continuera ensuite son exécution pendant que* [Capteur] commencera la sienne. Le concept [Bool:B1] joue le rôle d'un canal de synchronisation entre les deux concepts [HorlogeReg] et [Capteur].

Ces deux derniers s'exécutent ensuite en parallèle avec d'autres possibilités de synchronisation, via les deux autres booléens :

→ L'exécution de [Capteur] implique la "lecture/perception" d'un événement. Si la seconde période est écoulée, [Capteur] devra terminer son exécution même s'il n'a pas encore "lu" l'événement attendu. Ceci est réalisé à l'aide du booléen [Bool =true] dans [HorlogeReg]-out2->[Bool =true]-in2,/-> [Capteur], que [HorlogeReg] rend à "false" une fois la seconde attente terminée, signalant ainsi à [Capteur] la fin de la seconde période.

→ Considérons maintenant comment [Capteur] contrôle [HorlogeReg] : si [Capteur] capte l'événement attendu alors [HorlogeReg] devra terminer son exécution, même si la période d'attente maximale n'est pas encore terminée. Un tel contrôle est réalisé par le booléen [Bool =true] dans [HorlogeReg]<-in3,/-[Bool =true]<-out2-[Capteur], que [Capteur] rend à "false" une fois il a capté l'événement attendu (la seconde attente effectuée par [HorlogeReg] est une attente itérative conditionnée par la période et aussi par le booléen précédent).

Note: si [HorlogeReg] termine son exécution avant celle de [Capteur], alors ce dernier sera interrompu (par [HorlogeReg]) et il affectera une valeur au booléen en sortie [Bool =true] qui est en entrée de [HorlogeReg]. Ce changement du booléen ne doit pas redéclencher [HorlogeReg], d'où la nécessité d'utiliser l'attribut "/" : "in3,/".

De même, si [Capteur] termine son exécution et signale cela à [HorlogeReg] (dans le cas où il est toujours actif), via le booléen [Bool =true] dans [HorlogeReg]<-in3,/-[Bool =true]<-out2-[Capteur], alors [HorlogeReg] va interrompre la seconde attente et affecter "true" à son second booléen en

sortie [HorlogeReg]-out2->[Bool =true]-in2,/->[Capteur]. Ce changement du booléen ne doit pas redéclencher [Capteur], d'où la nécessité d'utiliser l'attribut "/" : "in2,/".

→ *Définition de* HorlogeReg (figure 4.15):

A l'exécution, seul [WaitFor] est activé impliquant un délai d'une durée D1. Après cela, la valeur "true" est affectée au premier paramètre en sortie de HorlogeReg (elle est affectée plus précisément à l'argument correspondant au paramètre : [Bool :B1] dans la Figure 4.14). Cette affectation permettra l'activation de [Capteur] qui attend justement cette valeur.

Dans le corps de HorlogeReg et suite à l'affectation, le concept [CondWaitFor] est délenché et activé. Son activation est conditionnée toutefois par la période D2 et le booléen [Bool:in3/_B3]. Après l'exécution du concept [CondWaitFor], la valeur "false" est affectée au booléen [Bool:out2/_B2].

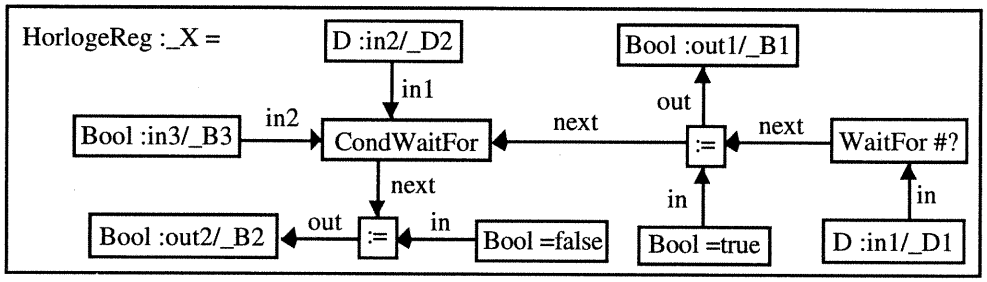


Figure 4.15 : Définition de HorlogeReg

→ *Définition de* Capteur (figure 4.16):

A l'exécution, [ReadEvt] attend que [HorlogeReg] lui envoie un signal via le booléen [Bool :in1/_B1], lui indiquant que la première attente est terminée; [ReadEvt] entammera alors une "lecture" itérative d'événements. La lecture est conditionnée par le test sur l'événement attendu et aussi par le booléen [Bool :in2/_B2] qui représente le second contrôle de [HorlogeReg] sur [Capteur]. Si un événement "lu" correspond à l'événement attendu, alors il est ajouté à la liste des événements et le booléen [Bool :out2/_B3] est mis à "false" indiquant à [HorlogeReg] (s'il est toujours en exécution) que l'événement est lu et donc qu'il doit interrompre sa seconde attente.

Exemple : *Diagramme de transition d'états*

Les diagrammes de transition d'états sont très utilisés en informatique (par exemple, dans la théorie des automates et l'analyse des langages de programmation, dans le génie logiciel et dans le traitement du langage naturel). Ils sont utilisés également comme formalisme pour la programmation par événements (un événement peut déclencher une transition d'un état à un autre).

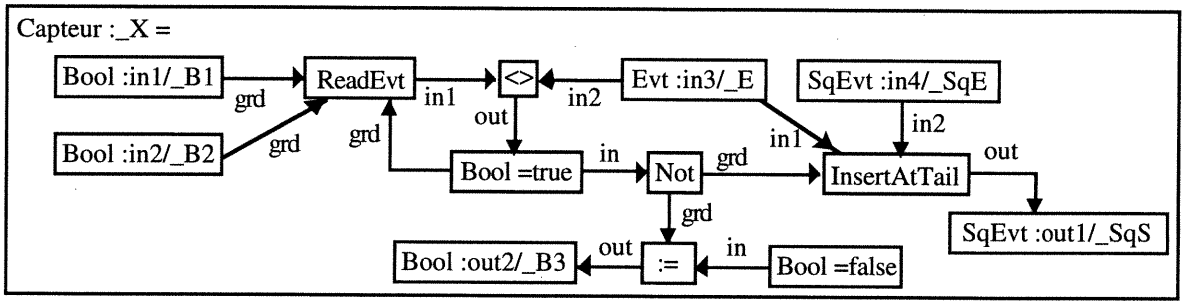


Figure 4.16 : définition de Capteur

Considérons le diagramme de transitions suivant (Figure 4.17) : il reconnaît un identificateur qui commence avec une lettre suivie de lettres et/ou chiffres, séparées éventuellement par "-". L'activation du diagramme commence avec l'état initial "Id", si le caractère courant est une lettre (ce qui correspond à la partie condition de la transition) alors on écrit "is lt" (la partie action de la transition) et on avance à l'état "e1". Deux transitions sont possibles à partir de "e1" : si le caractère est une lettre ou un chiffre on écrit "is lt-ch" et on reste à l'état "e1", si le caractère est "-" on écrit "is -" et on avance à l'état "e2".

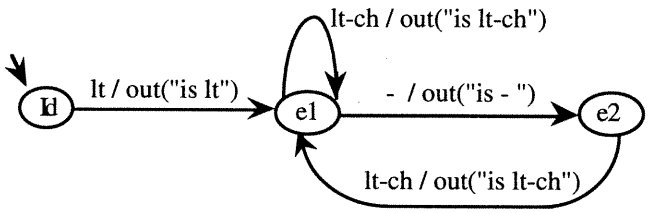
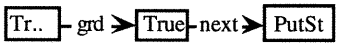
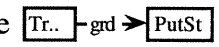


Figure 4.17 : Exemple de diagramme de transition

La formulation du diagramme de transition en Synergy pourrait être comme suit (Figure 4.18) : nous avons un concept qui représente l'état courant et le concept [PutSt] qui le met à jour. L'activation du GC commence avec l'affectation de l'état initial "Id" à l'état courant, la transition [TrLt] est ensuite déclenchée : si sa partie condition est vérifiée alors la partie action est effectuée et la transition retourne "true", sinon elle retourne "false". Ainsi, si la transition est effectuée, le concept [True] est alors déclenché suivi de [PutSt] qui mettra à jour l'état courant, sinon le concept [True] n'est pas déclenché et il en sera de même pour le concept [PutSt]. Nous utilisons  au lieu de  car si les transitions sont directement liées à [PutSt] via -grd->, alors [PutSt] ne pourraitt être activé que si toutes les transitions "en entrée" étaient effectuées, ce qui ne correspondrait pas à la sémantique du diagramme de la Figure 4.17. Le lien -next-> n'impose pas une telle contrainte ; il n'est pas une condition pour son concept cible.

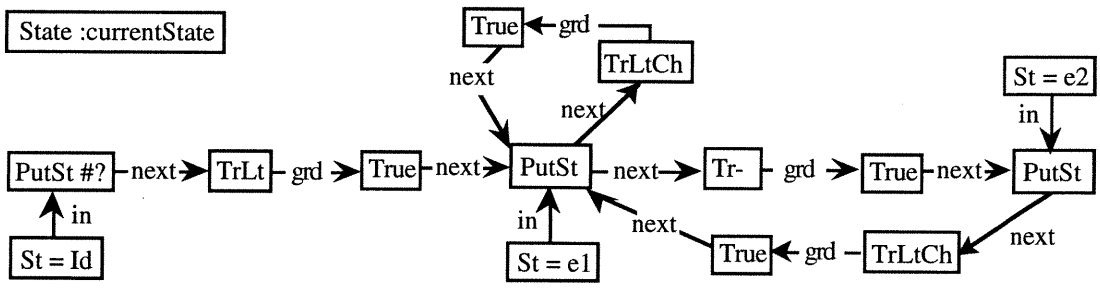


Figure 4.18 : Formulation en Synergy du diagramme de la Figure 4.17

Pour les transactions, considérons à titre d'exemple la définition de la transition TrLtCh (Figure 4.19) : si la condition est vérifiée (le caractère courant est conforme à la catégorie spécifiée), l'action associée est effectuée. Le résultat retourné par la transaction est celui de la condition. Notons que si la condition est vérifiée, la transition ne doit retourner "true" qu'après la terminaison de l'action (ici [WriteC]).

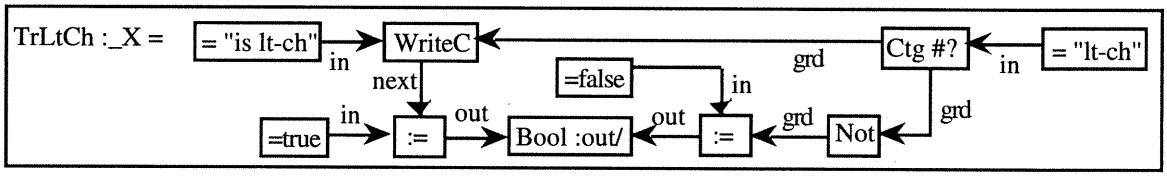


Figure 4.19 : Définition d'une transition

4.5 Résumé

Nous avons illustré, à l'aide de plusieurs exemples, l'utilisation de Synergy pour réaliser une programmation fonctionnelle, une programmation procédurale, une programmation par accès et une programmation par événement. Pour chacun des quatre types de programmation, nous avons identifié des éléments de bases et nous avons montré leur formulation en Synergy. Evidemment, le but du chapitre n'est pas de fournir une étude complète et détaillée du rapport entre Synergy et chacun des quatre types de programmation, mais plutôt de montrer les possibilités de Synergy vis à vis chacun d'eux et inversement, de montrer comment ces derniers peuvent être formulés en Synergy.
Le prochain chapitre complète celui-ci en présentant l'utilisation de Synergy pour réaliser d'autres modèles de programmation.

Chapitre 5

Programmations de processus concurrents, d'objets actifs et d'agents en Synergy

Nous présentons dans ce chapitre des exemples de "programmes" Synergy illustrant les modèles de programmation concurrente par processus, par objets actifs et par agents.
Le premier modèle est illustré par un exemple de processus concurrents avec communication synchrone par canaux. Pour la programmation concurrente par objets actifs, nous décrivons notre "implantation" de ce modèle en Synergy, en précisant la structure d'un objet actif et les types primitifs et/ou définis qui encapsulent ce modèle dans notre langage. Deux exemples sont présentés, un dans l'annexe D et le second dans ce chapitre.
Pour la programmation concurrente par agents, nous distinguons entre agent "réactif" et agent "cognitif", deux exemples illustrent la formulation des deux types d'agent en Synergy. Les systèmes multi-agents sont aussi considérés.

5.1 Processus concurrents avec communication synchrone par canaux

Différentes formes de programmation de processus concurrents ont été proposées en littérature [Andrews, 91]. Nous considérons dans cette section le modèle impératif, basé généralement sur une exécution parallèle de processus séquentiels avec des primitives pour la communication et la synchronisation. Les processus peuvent accéder/manipuler des objets communs (qui sont des entités statiques) et peuvent communiquer, de façon synchrone ou asynchrone, par un moniteur, un canal ou par envoi de messages. Différents modèles (comme CSP de Hoare [Hoare, 85] et CCS de Milner [Milner, 80, 89]) et langages (comme Occam, Ada, Emerald) ont été proposés pour ce type de programmation [Andrews, 91].
Une formulation orientée graphe est souvent utilisée pour ce type de modèles (par exemple les graphes de tâches [Gelenbe, 89], les graphes de communication [Aggarwal et Lee, 87] ou une extension des réseaux de pétri [Balbo, 92]).

Parmi les autres modèles de programmation de processus concurrents, citons le modèle de traitement parallèle basé sur l'exploitation concurrente d'un espace commun de données

([Banâtre et Le Métayer, 91], [Ciancarini, 94]) et le modèle UNITY [Banâtre et Le Métayer, 91] qui basé sur une généralisation et une intégration de la programmation impérative et des systèmes de transition d'états. Dans la section 5.2 nous montrons comment la notion d'espace(s) commun(s) est réalisée en Synergy.

Dans la programmation de processus concurrents avec communication synchrone par canaux, un processus P1 qui envoit un message M via un canal C1 (l'envoi est noté souvent C1 ! M) attend (si nécessaire) que le canal C1 soit libre (ne contienne pas de valeur), met ensuite M dans le canal et attend qu'il soit reçu par un autre processus (il attend que le canal soit libre de nouveau). Un processus P2 qui exécute une réception de message via le canal C1 (l'attente de réception est notée souvent C1 ? X) attend que C1 contienne une valeur V, cette dernière est ensuite enlevée du canal et affectée à X.
La communication par canal permet d'établir un *rendez-vous* entre deux processus, le "lieu de rencontre" étant le canal, celui qui arrive le premier attend l'autre et une fois la rencontre effectuée et l'information échangée, les deux processus poursuivent leur exécution, en parallèle.

Exemple : *Processus concurrents avec communication synchrone par canaux*
Pour illustrer la formulation en Synergy de ce type de programmation, considérons l'exemple classique de deux processus P1 et P2 qui demandent au processus Imp (imprimante) d'imprimer un certain nombre de données (Figure 5.1) : P1 communique à Imp le nombre de données via le canal CNbr1 et les données via le canal CDn1 (les données sont transmises une à une). Le même protocole est établi entre P2 et Imp.

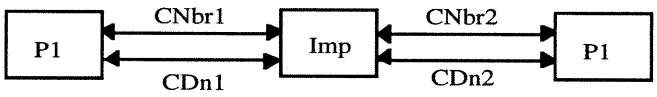


Figure 5.1 : Communication par canaux entre processus concurrents

Le code "procédural" est présenté ci-dessus, suivi de sa formulation en Synergy.

Le processus **Imp** : son corps correspond à une boucle infinie qui attend qu'un des deux canaux (CNbr1 ou CNbr2) contienne une valeur. Si par exemple le canal CNbr1 contient une valeur, alors Imp va exécuter le traitement associé (la boucle "for i=x-1 ..") et les autres cas du "select" ne seront pas considérés même si entre temps la condition d'un des cas est vérifiée (par exemple, le canal CNbr2 qui reçoit entre temps une valeur). L'exécution du traitement associé à un cas bloque les autres cas.

Notons que si les deux canaux contiennent en même temps des valeurs alors le "select" va choisir l'un des deux cas, de façon indéterminée.
Le traitement associé au canal CNbr1 (et aussi au canal CNbr2) est une boucle qui attend à chaque cycle une donnée, via le canal CDn1 (canal CDn2 pour le second cas); une fois la donnée reçue elle est imprimée.

```
Imp
Select  Loop
        CNbr1 ? x :
            for i = x-1 downto 0
                CDn1 ? Don
                ..print Don ..
            endFor
        CNbr2 ? x :
            for i = x-1 downto 0
                CDn2 ? Don
                ..print Don ..
            endFor
endSelectLoop.
```

Le processus **P1** : il calcule les données et envoit ensuite un message à l'imprimante, via CNbr1, l'informant du nombre de données à imprimer. Le processus P1 attend que Imp reçoive (et accepte) le message. Les deux processus s'engagent ensuite dans la réalisation d'une tâche commune (impression de données) : P1 envoit un flot de données qui sont reçues et imprimées par Imp.

```
P1
While true
    ...Calculer les données ArrDon[1..N] ..
    CNbr1 ! N
    for i = N-1 downto 0
        CDn1 ! ArrDon[i]
    endFor
endWhile
```

Le code pour P2 est similaire à celui de P1, avec respectivement les canaux CNbr2 et CDn2, au lieu de CNbr1 et CDn1.

Formulation en Synergy

Considérons tout d'abord la formulation du cas général du *selectLoop* (nous supposons que la condition d'un cas, appelée souvent "guard", ne peut être qu'un ordre de réception) :

```
selectLoop
        Chan1 ? X1 :  Trait1
        ChanN ? XN :  TraitN
endSelectLoop
```

La sémantique implicite de cette instruction est rendue explicite dans la formulation en Synergy (Figure 5.2) :

- l'activation indéfinie du code est assurée par l'initialisation de l'état du concept [SelLoop] à "!@", l'interpréteur de Synergy conclut indéfiniment que le contexte de la Figure 2 est actif puisqu'au moins un de ses concepts est actif.
- l'instruction "selectLoop" évalue les conditions de ses cas, elle déclenche donc des attentes de réception sur les canaux associés. Dès qu'un des canaux contient une information, "selectLoop" s'engage avec le cas associé et "ignore" les autres. Après l'exécution du traitement du cas, "selectLoop" reconsidère de nouveau ses cas.

Dans la formulation en Synergy, au lieu d'évaluer les cas et susciter des attentes, on réalise un *traitement par évenenement* : quand un canal reçoit une valeur, l'ordre de réception sur ce canal (qui se trouve dans un contexte actif) "réagit" à cet évenement et se déclenche. On ne force donc pas leur évaluation.

Comme le montre le graphe de la Figure 5.2, l'argument d'une réception (l'opération ?) est une référence au canal, c'est cela qui procure à cette opération son caractère "réactif".

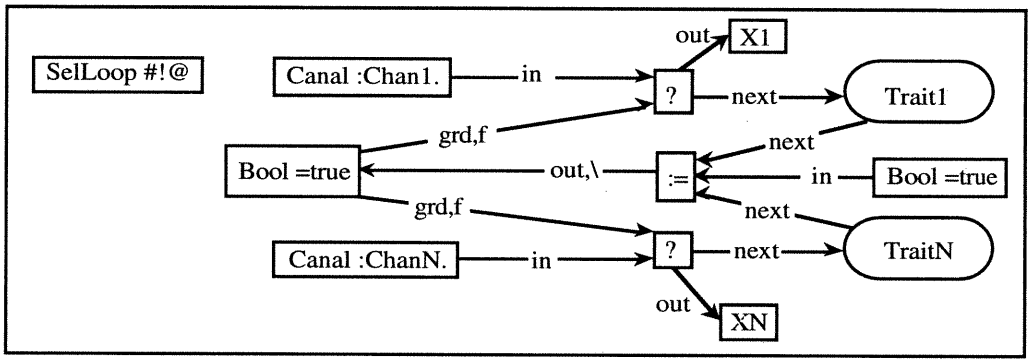


Figure 5.2 : Formulation en Synergy de la forme générale du *SelectLoop*

Dans le graphe ci-dessus, le concept booléen [Bool =true] assure d'une part l'exclusion des cas et, d'autre part, la séquence du traitement. L'exclusion est assurée par le fait que la première instruction de réception déclenchée consomme la valeur du booléen, les autres instructions de réception ayant été déclenchées entre temps sont ainsi bloquées car le booléen est sans valeur. Celle-ci est déterminée uniquement après l'exécution de la partie traitement du cas "actif". Notons l'utilisation du "couper-propagation-arrière ; \" : une fois la valeur du booléen [Bool=true] consommée, si elle est demandée de nouveau, il ne faut pas propager la demande via le lien "out" au concept [:=] ; ce dernier doit être déclenché uniquement suite à la terminaison du traitement du cas actif. L'attribut "\" permet d'inhiber la propagation arrière qui n'est pas souhaitée dans ce cas.

Considérons maintenant l'utilisation particulière du "SelectLoop" dans la définition du processus Imp.

Définition du processus **Imp** (Figure 5.3) :

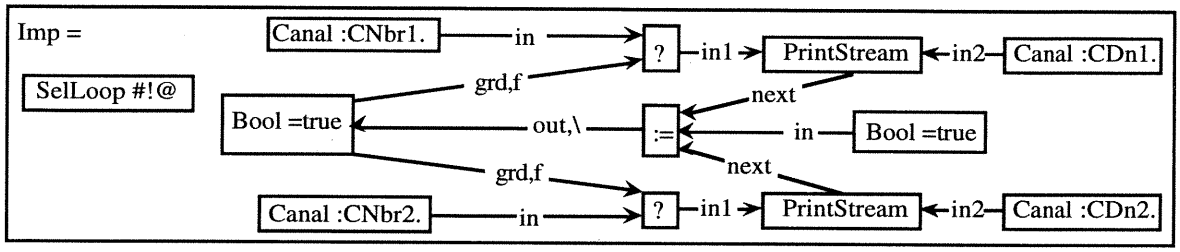


Figure 5.3 : Définition du processus Imp en Synergy

Définition de la procédure "auxiliaire" PrintStream(x, c) qui correspond au code suivant :

```
for i = x-1 downto 0
    c ? Don
    ..print Don ..
endFor
```

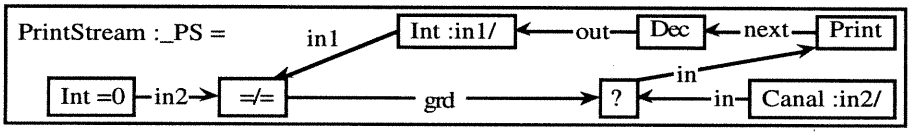


Figure 5.4 : Définition de l'opération PrintStream

Définition du processus **P1** (Figure 5.5) : comme pour le code procédural de Imp, une fois activée, P1 va le demeurer. Le traitement commence avec la partie "Calculer" qui retourne le vecteur des données et sa taille, le concept [!] est ensuite déclenché. Il est suivi du test [=/=]; s'il réussit, alors l'autre concept [!] est exécuté, suivi ensuite de [Dec] qui décrémente de 1 la valeur de i. Le changement de la valeur de ce dernier déclenchera de nouveau le test [=/=].

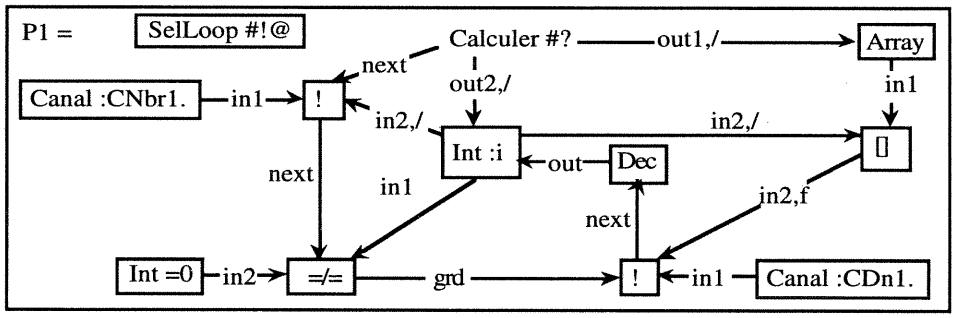


Figure 5.5 : Définition du processus P1 en Synergy

Voici à présent le "GC principal", avec les processus et les canaux qui sont accédés par référence (Figure 5.6). Les processus sont mis à l'état "?" pour initier leur exécution.

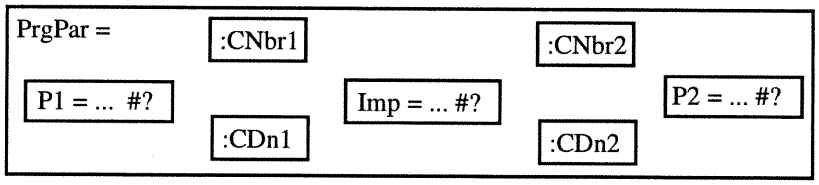
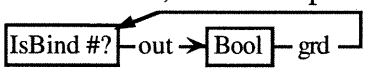


Figure 5.6 : L'exemple en Synergy des trois processus communiquant par canaux

Considérons maintenant la définition des deux "primitives" de communication ? et ! :

→ Rappelons la sémantique de l'opération Canal ! Message : un processus P qui envoit un message Message via un canal Canal attend (si nécessaire) que le canal soit libre (ne contienne pas de valeur), dépose ensuite Message dans le canal et attend ensuite que Message soit reçu par un autre processus (en d'autres termes, il attend que le canal soit libre de nouveau).

Le GC suivant décrit fidèlement cette sémantique (Figure 5.7) : une fois l'opération "!" activée, elle doit le rester tant que son traitement n'est pas terminé et cela, même si elle est dans un état d'attente.

Dans la formulation en Synergy, c'est le concept [:actif #!@] qui maintient l'opération "!" en activité (Figure 5.7). Le concept [:actif #!@] sera "neutralisé" par l'opération prédéfinie [Desactif] qui change l'état d'un concept à "repos ; o". L'opération [Desactif] n'est déclenchée que si le traitement correspondant à l'opération "!" est terminé. L'attente, avant et après l'affectation de la valeur au canal, est représentée par le cycle : 

qui est conditionné par le résultat de [IsBind] : si le résultat est différent de "false", alors [IsBind] est déclenché de nouveau, sinon le cycle est bloqué. Le concept [Not] produit par contre "true", ce qui déclenche l'affectation du message au canal (Figure 5.7). Le traitement rentre ensuite dans une nouvelle phase d'attente : il attend que la valeur qui vient d'être affectée au canal soit consommée, c'est le rôle du second cycle.

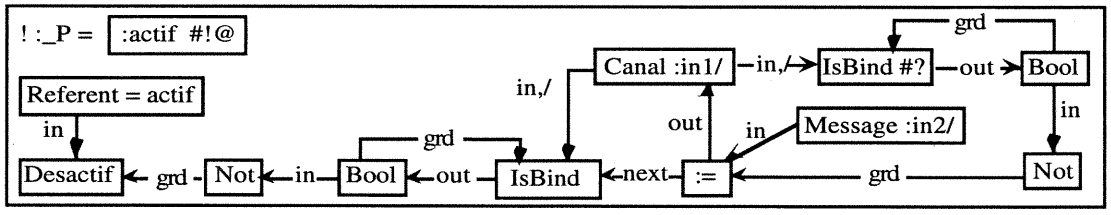


Figure 5.7 : Définition de l'opération de communication "!"

→ Rappelons la sémantique de l'opération Canal ? X : Un processus P qui exécute une réception de message via un canal Canal attend que le canal contienne une donnée, cette dernière est ensuite enlevée du canal et affectée à X.
La formulation en Synergy est presque immédiate (Figure 5.8) : l'affectation [:=] est déclenchée et se met en attente, tant que son argument en entrée n'a pas de valeur. Lorsque l'argument a une valeur, l'affectation consomme la valeur et l'affecte à la donnée (paramètre en sortie de "?"). L'activation "forcée" de l'opération "?" est ensuite neutralisée, par [Desactif].

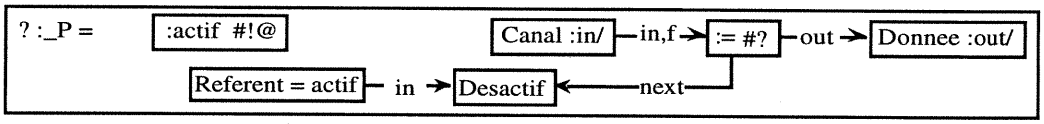


Figure 5.8 : Définition de l'opération de communication "?"

5.2 Programmation concurrente orientée objet dans Synergy

Synergy peut être utilisé comme langage de Programmation Concurrente Orientée Objet (PCOO).
Rappelons en premier qu'un programme en PCOO est généralement composé d'objets actifs qui s'exécutent en parallèle et qui intéragissent par envoi/réception de messages. Un objet actif possède en général une boîte de messages (appelée aussi "queue de messages"), une partie déclarative composée de variables et une partie procédurale composée de méthodes. Une méthode est composée en général d'une partie condition sur le message (message pattern) et d'une partie traitement, conditionnée éventuellement par des contraintes.
L'exécution au sein d'un objet est souvent séquentielle (comme dans ABCL) bien que certains modèles de PCOO supportent un parallélisme intra-objet ([Saleh et Gautron, 91], [Kaiser et al., 93], [Wakita, 93], [Sargeant, 93], [Liddle et al., 94]), en permettant par exemple à plusieurs méthodes de s'exécuter en parallèle. Enfin, certains modèles de PCOO supportent en plus les notions de classes, de hiérarchie de classes et d'héritage ([Papathomas, 91], [Matsuoka et Yonezawa, 93]).

Avant de considérer la PCOO offerte par Synergy, soulignons que ce dernier peut être utilisé comme un langage orienté objet (sans concurrence) basé sur les réseaux sémantiques.
Une telle utilisation des réseaux sémantiques a été proposée au sein de la communauté des GC ([Hines et al., 90], [Wuwongse et Ghosh, 92]) et en "dehors" également ([Mylopoulos et al., 93] et [Skuce et Lethbridge, 95]). Hines et al. ainsi que Wuwongse et Ghosh utilisent les graphes de flot de données proposés par Sowa pour représenter les méthodes, Mylopoulos et al. utilisent une formulation logique du traitement alors que Skuce et

Lethbridge adoptent une formulation "frame". Par ailleurs, Sowa ([Sowa, 93], [Ellist, 95]) propose les GC comme fondement logique pour la programmation orientée objet.

PCOO dans Synergy

Un objet actif (avec sa queue de message, ses parties déclarative et procédurale) est représenté par un concept et l'exécution est parallèle, aussi bien entre objets qu'au sein d'un objet. Enfin, Synergy permet la formation et l'exploitation d'une hiérarchie de classes d'objets actifs, avec possibilité d'héritage.

Puisque Synergy permet un parallélisme inter- et intra- objets; un objet actif peut donc recevoir en même temps plusieurs messages et des méthodes d'un même objet peuvent être exécutées en parallèle. Dès qu'un objet reçoit un message dans sa queue de messages, il cherche une méthode qui peut le traiter. Si la méthode est trouvée mais non disponible, le message est mis en attente dans la queue de la méthode (chaque méthode a une queue de "messages en attente d'être traités"). Si aucune méthode de l'objet ne peut traiter le message, l'objet délègue le traitement du message à son "concept-super" (s'il en a).

La réalisation de la PCOO en Synergy est basée sur des opérations primitives et des opérations prédéfinies de gestion (comme MessageDispatcher, MethodMgr et AgentMgr) et de communication (comme Send, SendSync, Receive et WaitMessage).

Ces opérations supposent une formulation particulière d'un objet actif. Nous commençons la description d'un objet actif en Synergy, nous considérons ensuite les opérations de gestion et de communication.

5.2.1 Formulation d'un objet actif en Synergy

La structure d'un objet actif en Synergy est une conséquence directe du cadre général décrit ci-dessus : un objet (et/ou une classe) est représenté par un GC (Figure 5.9) qui comprend un concept pour la queue de message, éventuellement un concept-super, une partie déclarative (représentée par une partie du GC) et une partie procédurale (représentée également par une partie du GC). Cette dernière peut comprendre les méthodes, un appel à la primitive MessageDispatcher (définie dans la prochaine section) et éventuellement le *corps procédural* de l'objet. En effet, un objet actif peut avoir, en plus des méthodes, un corps procédural [America, 87, 90].

Une méthode est décrite par un GC composé de trois concepts : un pour la partie condition sur le message [:condition =...], un autre pour la partie traitement [:behavior =...] et le troisième pour gérer l'activation de la méthode [MethodMgr :super]. Le type de ce dernier concept est prédéfini et fait donc partie du graphe de généralisation de base. Le programmeur n'a donc pas à le définir ni à connaître sa définition actuelle.

Note : les référents "condition", "behavior" et "message" sont des mots clés, le programmeur doit les utiliser dans la formulation et l'utilisation des méthodes.

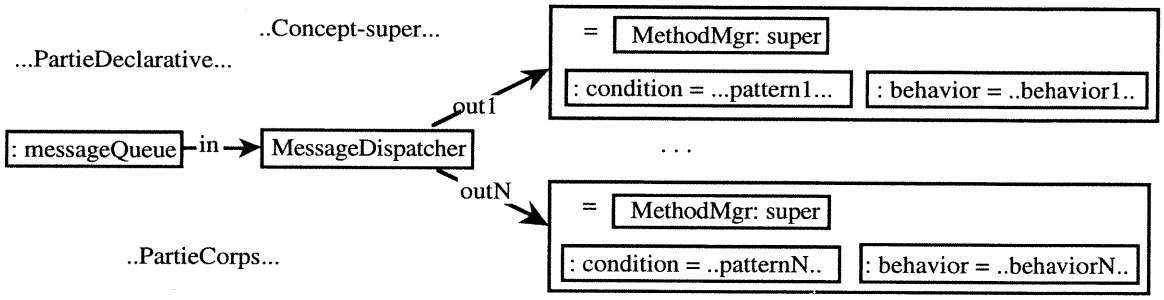


Figure 5.9 : Structure d'un objet actif

Remarque : tous les éléments d'un objet actif ne sont pas obligatoires, de sorte que différents types d'objets peuvent être considérés. Par exemple : 1) un objet peut être sans queue de message ni méthodes mais il peut toujours envoyer des messages, 2) un objet peut avoir une queue de message mais pas de méthodes, il peut donc envoyer et recevoir des messages, 3) un objet peut avoir une queue de message et des méthodes, dans ce cas il doit contenir également un appel à MessageDispatcher, comme indiqué par la Figure 5.9.

5.2.2 Les opérations de gestion : MessageDispatcher et MethodMgr

→ **MessageDispatcher :** C'est une opération primitive qui a la queue de messages en entrée et les méthodes en sortie (Figure 5.9), elle identifie les méthodes qui peuvent traiter les nouveaux messages. MessageDispatcher est déclenchée lors d'une insertion d'un message dans la queue de message de l'objet. Elle considère alors chaque nouveau message NouvMsg et tente d'unifier une copie de NouvMsg avec une copie de la partie condition d'une méthode. Si l'unification réussit pour une méthode, la copie unifiée du message est alors insérée dans la queue de la méthode; autrement, une copie du message NouvMsg est envoyée au concept-super de l'objet (s'il en a). Dans les deux cas, NouvMsg demeure dans la queue des messages de l'objet, seules les opérations Receive et WaitMessage peuvent enlever un message de la queue.

Remarques :

→ La queue de message d'un objet est divisée, par l'élément spécial "###", en deux parties, une pour les nouveaux messages et l'autre pour les messages déjà traités.

→ Si le message et la condition associée à la méthode sont des GC alors l'unification est une unification des GC, autrement elle se réduit à un test d'égalité.

→ Plusieurs méthodes d'un objet peuvent s'exécuter en parallèle. Des conflits d'accès à certains éléments de l'objet peuvent alors survenir. Synergy possède son propre gestionnaire de conflit pour les cas où plusieurs concepts s'apprêtent à affecter, en même temps, des valeurs à un même concept. Le programmeur peut toutefois définir son propre gestionnaire de conflit, qui pourrait être plus approprié au type de PCOO et à l'application en question.
→ Avec MessageDispatcher et MethodMgr, Synergy offre une gestion décentralisée des méthodes; chacune évolue selon son traitement et le contenu de sa queue.
→ Le routage (dispatching) peut s'effectuer pendant que certaines méthodes sont en exécution. Par exemple, MessageDispatcher peut insérer des éléments dans la queue d'une méthode pendant que celle-ci est en exécution.
→ A sa terminaison, MessageDispatcher déclenche les méthodes qui sont au repos. Si toutefois la queue d'une méthode est vide, alors cette dernière retourne à l'état repos.

→ **MethodMgr** (Figure 5.10) La définition de MethodMgr tient compte du "contexte d'utilisation" qui est celui d'une méthode (Figure 5.10). En effet, la définition de MethodMgr contient un concept pour représenter la queue des messages en attente de la méthode, une référence au traitement de la méthode et un concept pour le message en cours de traitement. Ce dernier concept permettra au programmeur d'accéder aux éléments du message, par une référence du type "message. ..".
Considérons maintenant le côté traitement de la définition (Figure 5.10) : si la queue des "messages à traiter par la méthode" [:queue] n'est pas vide, [RemoveHead] enlève le premier élément de la queue et le met dans [: message]. Le comportement de la méthode est ensuite activé (indirectement via l'activation du concept-référence [:behavior.]). Quand le comportement est terminé, il déclenche de nouveau [RemoveHead] qui cherche un autre élément de la queue.

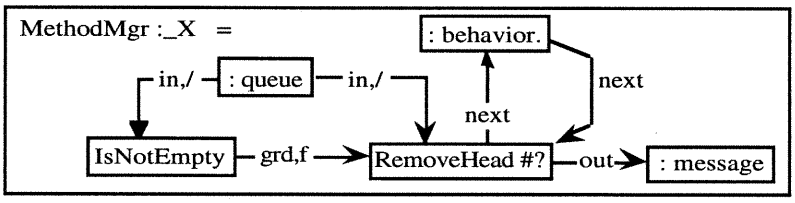


Figure 5.10 : Définition de MethodMgr

5.2.3 Communication "couplée" dans Synergy

La communication est *couplée* si les objets se "connaissent" entre eux, un objet sait à qui envoyer un message. Nous présentons dans cette section les opérations de communication "couplée". Considérons tout d'abord la structure d'un message (Figure 5.11), ce dernier

peut être décrit par un contenu et un ensemble d'attributs : l'identificateur du message, l'expéditeur, le receveur, le type du message (informer, demander, ..), la priorité et la date d'expédition (le nombre des attributs peut varier selon le domaine d'application).

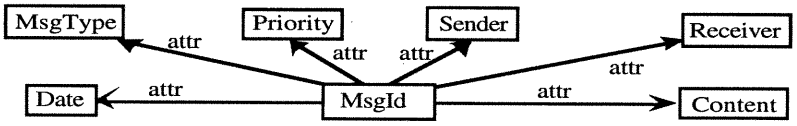


Figure 5.11 : Structure d'un message

Le langage offre quatre opérations de communication (la définition en Synergy est fournie dans les figures ci-dessous) :

- ***Send*** (Figure 5.12) : elle effectue un envoi asynchrone de message (envoi sans attente de réponse),
- ***SendSync*** (Figure 5.13) : elle effectue un envoi synchrone de message ; un objet actif O envoit le message M à un destinataire D et attend que D lui envoie un message de réponse concernant le message M. Notons que l'attente concerne D (et non un autre objet) et une réponse au message M (et non un autre message).
- ***Receive*** : elle effectue une réception asynchrone (si le message attendu n'est pas reçu, elle n'attend pas). Receive est fournie par Synergy comme une primitive :
  receive(**in** Message, Queue; **out** Boolean). Cette opération parcourt Queue à la recherche d'un élément qui puisse s'unifier avec Message, si un tel élément est trouvé, il est enlevé de Queue et l'opération retourne "true", sinon elle retourne "false".
- ***WaitMessage*** (Figure 5.15) : elle effectue une réception synchrone (elle attend que le message soit reçu).

→ Définition de **Send** (Figure 5.12) : Send ajoute le message dans la queue du receveur, la queue est ensuite déclenchée, les opérations qui attendent une nouvelle insertion dans la queue sont ainsi avisées et peuvent réagir en conséquence.

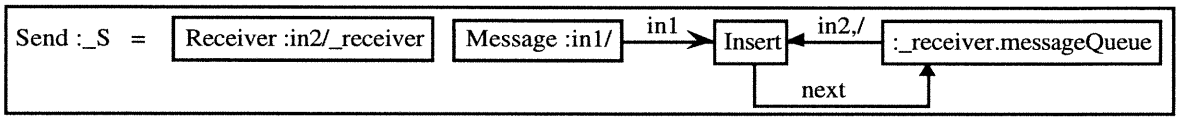


Figure 5.12 : Définition de *Send*

→ Définition de **SendSync** (Figure 5.13) : SendSync envoit le message au receveur et cherche ensuite l'identité de l'expéditeur, ce dernier va ensuite attendre le message que le receveur devrait lui envoyer.

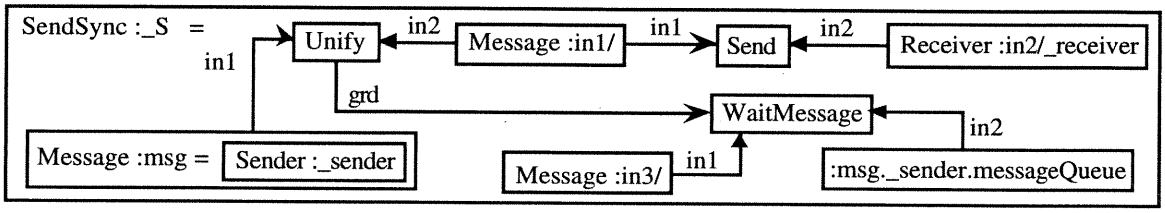


Figure 5.13 : Définition de *SendSync*

Exemple d'utilisation de *SendSync* (Figure 5.14) : l'agent Anne envoit un message à l'agent Jo lui demandant de bouger les jambes et attend ensuite une réponse de la part de Jo, pour ce message.

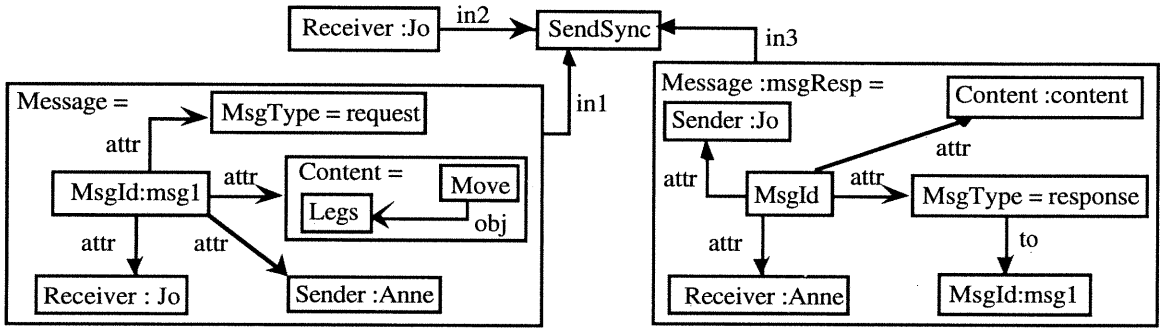


Figure 14 : Exemple d'utilisation de *SendSync*

→ Définition de **WaitMessage** (Figure 5.15) : Une fois activée, l'opération WaitMessage va le demeurer tant qu'elle n'a pas reçu un message qui puisse s'apparier au message attendu.

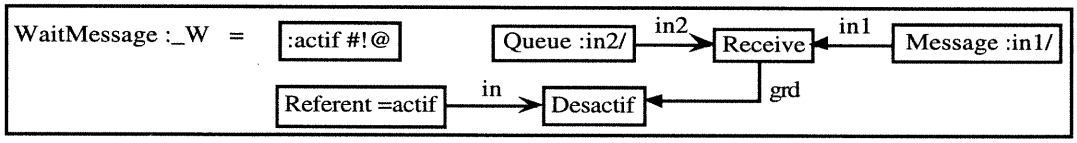


Figure 5.15 : Définition de *WaitMessage*

Remarques :

- Actuellement, l'insertion du message dans la queue ne tient pas compte de la priorité du message.
- D'autres opérations de communication peuvent être définies par le programmeur. Par exemple, on pourrait définir des variantes de Send et SendSync qui tiendraient compte de la

priorité et aussi des variantes de Receive et WaitMessage qui n'élimineraient pas le message choisi de la queue des messages.

5.2.4 Gestion des opérations de communication concernant un même objet

Nous allons considérer les trois cas suivants : envois simultanés à un même objet O1, réceptions simultanées par un même objet O1 et envois et réceptions simultanés par un même objet O1.

→ *Envois simultanés à un même objet O1*

Des envois simultanés à un même objet O1 s'apprêtent à insérer, en même temps, des messages dans la queue de message de l'objet, créant ainsi un conflit d'accès.
Synergy évite ce problème en réalisant une exécution séquentielle des insertions. Cette exécution séquentielle est ensuite ré-interprétée comme une exécution parallèle.

→ *Réceptions simultanées par un même objet O1*

Rappelons tout d'abord que l'opération Receive teste si la queue de message de l'objet contient un message qui puisse s'unifier avec son argument; si oui, le message est éliminé de la queue de message.
Maintenant, plusieurs Receive peuvent s'exécuter simultanément au sein d'un objet et il est possible que plusieurs Receive s'apprêtent à éliminer le même élément de la queue.
Pour éviter ce problème, Synergy exécute en séquence les Receive "simultanés" sans éliminer les éléments de la queue. Ce n'est qu'après l'exécution des Receive que les éléments sont éliminés. Ainsi, un même élément de la queue peut satisfaire plusieurs Receive qui y ont accédé en même temps.

→ *Envois et réceptions simultanés à/par un même objet O1*

Une situation problématique peut survenir si un envoi de message (Send par exemple) s'apprête à insérer un message M dans la queue de l'objet O1 en même temps qu'un Receive qui serait satisfait par M.
Synergy évite cette situation en exécutant les envois de message avant les opérations de réception, de sorte qu'un Receive pourrait considérer les messages qui sont en train d'être insérés dans la queue.

5.2.5 Application : modélisation orientée agent de l'unité des soins intensifs

L'annexe D présente une application qui exploite la PCOO-Synergy. Elle concerne une modélisation orientée agents de l'unité des soins intensifs où l'agent "infirmière" tente de

réaliser l'évaluation de l'agent "patient". L'évaluation porte sur différents systèmes et en particulier le système nerveux qui est considéré dans l'exemple de l'annexe D. Les deux agents communiquent par envoi de messages.
Un autre exemple est fourni dans la section suivante; il concerne l'implantation en PCOO-Synergy d'un agent "cognitif".

5.2.6 Communication "non-couplée" dans Synergy

Dans certaines applications, les objets actifs ne se "connaissent" pas, ils communiquent indirectement via *un(des) espace(s) commun(s) de données*. Les objets sont alors "non-couplés". Un objet actif peut écrire, lire ou détruire une donnée d'un espace commun. L'accès est effectué par appariement (matching). Au lieu d'envoyer un message à un autre objet particulier, un objet écrit le message dans un espace commun. De même, un objet qui a besoin d'une information n'attend pas qu'elle lui soit fournie par un autre objet, il va plutôt consulter l'espace commun pour "voir" s'il contient l'information.
La communication par "tableau-noir" [Jagannathan et al., 89].est un exemple de communication "non-couplée". Trois autres familles de langages se basent sur cette approche : 1) une famille "impérative" où les objets de l'espace commun sont des tuples, le groupe *Linda* est un exemple représentatif de cette famille ([Banâtre et Le Métayer, 91] et en particulier la partie sur Linda, [Carriero et al., 94]), 2) une famille "logique" où les objets de l'espace sont des termes [Ciancarini, 91, 94] et 3) une famille "contrainte" où les objets de l'espace sont des contraintes (comme les langages AKL [Janson et Haridi, 94] et Oz [Smolka, 95]). Notons par ailleurs la similarité de cette approche avec la notion de tableau-noir utilisée en intelligence artificielle [Jagannathan et al., 89].

Considérons plus en détail le cas de *Linda*, nous proposons ensuite une réalisation d'un SynergyLinda : Linda est un langage de coordination qui généralise certains aspects du partage de variables et de passage asynchrone de messages. Linda peut se "greffer" à un langage séquentiel (comme C) et permettre la formulation de processus concurrents partagant et communiquant via un espace de tuples.
L'espace de tuple contient des tuples passifs et des tuples actifs. Un tuple est de la forme : <tag, expr1, ..., exprN>, "tag" est un identificateur et exprI peut être une donnée (un entier, un booléen, etc) ou une expression (en l'occurrence un appel de fonction). Un tuple actif peut être évalué et il peut initier l'exécution de processus concurrents.
Pour communiquer de l'information et se synchroniser via un espace de tuples, les processus utilisent quatre opérations de base : "**out** t" (t représente un tuple) évalue les champs du tuple t et dépose le tuple résultat dans l'espace de tuples, "**in** t" extrait de l'espace de tuples un tuple qui puisse s'apparier à t. Le processus qui exécute un "**in** t" va attendre si

l'espace ne contient pas un tuple qui puisse s'apparier à t., "**rd** t" est similaire à "in" sauf qu'elle n'enlève pas le tuple de l'espace de tuples et "**eval** t" évalue les champs du tuple t, si un champ du tuple correspond à un appel de fonction alors celle-ci sera exécutée en parallèle avec le processus qui a exécuté le "eval".
Deux autres opérations "**inp**" et "**rdp**" sont offertes en complément et elles correspondent respectivement à "**in**" et "**rd**" sans l'attente : si l'espace de tuples ne contient pas un tuple qui puisse s'apparier à l'argument de l'opération "inp" ("rdp") alors cette dernière ne bloquera pas le processus qui l'a initié.

Une réalisation en Synergy de cette forme de communication et de synchronisation entre objets actifs peut être comme suit :

→ Le programmeur peut définir plusieurs espaces, un espace commun serait défini comme un objet actif [CGSpace :commSpace] avec une queue de message qui correspondrait à un *espace de GC* , implantée comme une liste de GC.

→ Les opérations sur un espace de GC commSpace sont définies comme des variantes des opérations de communication définies dans la section précédente :

- **put(in** Message, SpaceName) : dépose Message dans l'espace identifié par SpaceName. **put** correspond à un Send à commSpace,

- **getw(inout** Message; **in** SpaceName) : extrait de l'espace SpaceName un GC qui puisse s'apparier à Message. **getw** se mettra en attente si l'espace ne contient pas un GC qui puisse s'apparier à Message. **getw** correspond à un WaitMessage sur la queue de commSpace,

- **get(inout** Message; **in** SpaceName) : elle est similaire à *getw* sauf qu'elle n'attend pas. **get** correspond donc à un Receive sur la queue de commSpace,

- **seew(inout** Message; **in** SpaceName) : elle est similaire à *getw* sauf qu'elle n'enlève pas l'élément recherché. **seew** correspond donc à un WaitMessage sans élimination du message.

- **see(inout** Message; **in** SpaceName) : elle est similaire à *seew* sauf qu'elle n'attend pas. **see** correspond donc à un Receive sans élimination du message.

5.2.7 Rapport avec d'autres travaux

→ La "réalisation" de la PCOO dans Synergy est basée sur une interprétation particulière de la PCOO. D'autres interprétations sont possibles et certaines peuvent être réalisées également en Synergy : on pourrait modifier la définition des primitives offertes par Synergy pour réaliser la PCOO ou définir et utiliser d'autres. On pourrait aussi adopter l'approche de McAffer [McAffer, 95] qui propose une méta-architecture pour formuler les primitives selon différentes interprétations/définitions. Les primitives sont considérées elles-mêmes comme des objets. Pour atteindre le même objectif, Wakita [Wakita, 93] utilise la notion de "continuation".

→ La définition d'un objet actif dans Synergy inclut la possibilité qu'il ait un corps qui puisse s'exécuter en parallèle avec les méthodes de l'objet et/ou les corps des autres objets. L'association d'un corps à un objet actif à été introduite par la famille des langages POOL [America, 87, 90]. Dès qu'un objet est créé, son corps commence l'exécution. C'est la source principale de parallèlisme dans les programmes POOL (le traitement est séquentiel au sein d'un objet).

→ Notre interprétation de la PCOO considère la concurrence inter et intra-objets.
Cette possibilité est commune avec d'autres langages ([Saleh et Gautron, 91], [Kaiser et al., 93], [Wakita, 93], [Sargeant, 93], [Liddle et al., 94], [Guerraoui, 95]).
Différentes politiques de contrôle de la concurrence au sein d'un objet ont été proposées. Par exemple, le langage MELD [Kaiser et al., 93] offre deux politiques de contrôle de la concurrence (sérialisation des transactions et "bloc atomique" qui permet un accès exclusif à un objet durant une séquence d'instructions). MELD considère également l'interaction de ces deux politiques avec le cas d'une concurrence sans contrôle. Dans leur extension concurrente de C++, Saleh et Gautron [Saleh et Gautron, 91] proposent une autre politique de contrôle de la concurrence : associer une "attente conditionnelle" à une méthode. Une approche similaire a été proposée par McHale et Walsh [McHale et Walsh, 91]. Ces derniers introduisent trois "prédicats d'ordonnancement" (scheduling predicates) qui permettent différentes façons de synchroniser l'exécution des opérations.

→ Synergy offre une PCOO avec possibilité d'héritage.
Matsuoka et Yonezawa [Matsuoka et Yonezawa, 93] offre une discussion du rapport entre l'héritage et la concurrence ainsi que différentes solutions qui ont été proposées en littérature.

→ Synergy intègre la PCOO à d'autres modèles de traitement.

Sargeant [Sargeant, 93] propose un langage qui intègre la programmation fonctionnelle (avec les graphes de flots de données comme modèle sous-jacent) et la programmation concurrente orientée objet avec héritage. Kusakabe et Amamiya [Kusakabe et Amamiya, 94] proposent également un langage qui intègre les graphes de flots de données et la programmation orientée objet. L'intégration de la PCOO et la programmation avec contrainte a été proposée par Janson et Haridi (le langage AKL [Janson et Haridi, 94]) et Smolka (le langage Oz [Smolka, 95]) et enfin, l'intégration de la PCOO et la programmation procédurale "à la" Simula a été proposée par Madsen et al.(le langage BETA [Madsen et al., 93]).
Avec Synergy, nous proposons l'intégration de la PCOO à un modèle de traitement basé sur l'activation de Graphes Conceptuels.

→ Enfin, Papathomas [Papathomas, 91] fournit un cadre sémantique à la PCOO, basé sur le calcul de processus (process calculus). Il montre comment différents modèles de PCOO peuvent être analysés selon ce cadre.

5.3 Programmation orientée agents et systèmes multi agents

Après une brève introduction des notions d'agent "réactif", d'agent "cognitif" et de système multi-agent, nous considérons plus en détail chacune des trois notions ainsi que l'utilisation de Synergy dans leur réalisation.

5.3.1 Introduction

Plusieurs modèles, architectures et langages ont été proposés pour la programmation orientée agents ([Huhns, 87], [Bond et Gasser, 88], [Demazeau et Müller, 90a, 91a], [Maes, 91], [Wooldridge et Jennings, 95a]).
Nous présentons tout d'abord des "définitions" de la notion d'agent, puis quelques exemples de systèmes qui se trouvent "à cheval" entre la programmation concurrente orientée objet et la programmation orientée agents. Des systèmes/langages multi-agents sont ensuite introduit, selon qu'ils considèrent uniquement des agents "cognitifs", des agents "réactifs" ou des agents "hybrides".
Nous concluons enfin avec le type d'environnement multi-agents que Synergy pourrait implanter.

Sur les notions d'agent et de système multi-agents

Wooldridge et Jennings notent qu'il n'y a pas une définition unique de la notion d'agent mais plutôt un ensemble de schémas (différentes utilisations du terme agent) [Wooldridge et Jennings, 95a]. Même la distinction entre "intelligence artificielle distribuée" et "intelligence

artificielle décentralisée" dépend de la notion d'agent et de leur utilisation ([Gasser et Huhns, 89], [Bond et Gasser, 88], [Demazeau et Müller, 90a, 91a], [Doran, 92]). En effet, dans leur distinction de l'intelligence artificielle distribuée [Huhns, 87] et l'intelligence artificielle décentralisée [Demazeau et Müller, 90b], Demazeau et Müller notent que la première est concernée par la modélisation d'agents qui collaborent à trouver une solution à des problèmes globaux. L'agent peut correspondre à un processus élémentaire ou à des entités complexes avec un comportement "cognitif et rationnel". L'échange et le partage d'information entre les agents sont orientés par le but principal qui est de trouver, ensemble, la solution à certains problèmes. Les agents sont conçus afin de résoudre, de façon collective, un problème donnée.
L'intelligence artificielle décentralisée [Demazeau et Müller, 90a, 91a] est concernée par la notion d'agent "en soi" et par l'interaction qui résulterait entre plusieurs agents (en l'occurrence, les Systèmes Multi Agents). Les agents sont donc conçus en premier et on pourrait ensuite susciter un problème et observer leur comportement [Demazeau et Müller, 90b].

Dans leur revue des agents intelligents, Wooldridge et Jennings [Wooldridge et Jennings, 95a] distinguent deux usages de la notion d'agents :
→ une notion d'agent "faible" : les agents sont des "objets actifs" autonomes capables de communiquer entre eux (avec éventuellement un langage de communication entre agent comme celui défini dans [Genesereth et Ketchpel, 94]), ils sont “pro-actifs” (avec un comportement dirigé par les buts) et réactifs aussi (capable de percevoir et réagir aux changements de l'environnement).
→ une notion d'agent "forte" : selon cette vision, les agents sont en plus caractérisé par des notions "cognitifs" comme connaissance, croyance, intention, obligation, motivation et émotion.

Dans le domaine des SMA, la notion de système ou société est aussi importante que la notion d'agent. Elle a été considérée sous différentes formes par différents chercheurs :
→ Werner [Werner, 89] propose un cadre théorique général et formel pour décrire la structure d'une société. Sa théorie de la structure sociale est basée essentiellement sur la communication entre agents et les rôles de ces derniers dans la société.
La communication entre agents se base sur une théorie des actes de discours. La structure sociale est définie comme un ensemble de rôles, un rôle est un agent abstrait et un agent qui assume un rôle va l'intérioriser et s'y conformer (par exemple lorsqu'on demande à quelqu'un de jouer le rôle de témoin dans un mariage, la personne va intérioriser le rôle et il va se comporter conformement au rôle de témoin). Un agent peut avoir plusieurs rôles dans une société.

→ Singh [Singh, 91] fournit également une formulation théorique de la notion de structure sociale, mais sa formulation est plus générale que celle de Werner. Alors que ce dernier focalise sur des agents intentionnels et coopératifs, Singh considère également les agents réactifs et il ne se limite pas aux agents coopératifs. Singh définit la structure sociale en termes d'interactions "strategiques" (l'équivalent des rôles) et "réactives" de ses membres.
Singh a relevé par ailleurs l'importance de certaines "structures de contrôle" qui imposent des contraintes sur les activités des agents et sur leurs interactions (i.e. un agent doit attendre par exemple qu'une condition soit vérifiée, ou il doit attendre qu'un autre agent termine son activité, etc.).

Entre la PCOO et la programmation orientée agents

→ Dans **MACE** [Gasser et al., 87], un agent est un objet actif avec des capacités "intentionnelles" : la mémoire locale et le comportement de l'agent sont organisés en terme de connaissances, buts, capacités, plans, etc. La communication entre agents ne se base toutefois pas sur une théorie des actes du discours.

→ Dans **PANDORA-II** [Maruichi et al., 90], un agent est aussi un objet actif avec toutefois une certaine autonomie : alors que dans la PCOO, un objet qui reçoit un message doit le traiter, dans PANDORA-II chaque agent possède son propre interpréteur de message qui gère la queue des messages et décide quand et quel(s) messages traité(s) (ou ignoré) à un moment donné. Une autre extension à la PCOO est la notion de groupe d'agents (un groupe peut être formé également d'autres groupes). Un agent peut envoyer un message non seulement à un autre agent mais à un groupe d'agents et dans ce cas tous les membres du groupe recevront le message (les envois de messages électroniques en est un exemple). Un agent peut appartenir à plusieurs groupes en même temps. Il est à noter qu'un agent n'est pas obligé de connaître les membres d'un groupe (il lui suffit d'envoyer au groupe) et la formation du groupe est dynamique : des agents peuvent "rentrer" ou "sortir" d'un groupe.
Cette extension de la PCOO (interpréteur de message par agent et formation de groupe d'agents) forme les éléments de base du modèle de traitement organisationnel (organizational model of computation) proposé par Maruichi. L'implantation du modèle est réalisée en PCOO : un agent correspond à un objet actif et son interpréteur des messages est toujours actif. Un message est un objet "passif" transféré du transmetteur au(x) receveur(s) et un groupe (appelé aussi environnement) est aussi un objet passif référé par l'agent et contenant une liste de noms d'agents.

→ Le système **MAGES** [Bouron et al., 91] a été conçu pour les SMA hétérogènes et il a été réalisé en ACTALK; un langage d'acteur implanté en Smalltalk-80 offrant une hiérarchie de classes d'acteurs (acteurs selon le modèle de Hewitt et Agha, acteur selon le modèle de

Yonezawa, etc). Dans MAGES, la notion de groupe est modélisée comme un objet actif qui possède son propre interpréteur de messages et son propre comportement. Par ailleurs, un agent se déplace dans un contexte physique (spatio-temporel) qui correspond à son environnement (une zone dans une grille par exemple) et les agents peuvent réagir aux changements de leur environnement (par exemple, un agent qui a changé de place). Les environnements dans MAGES sont représentés par des agents et ils peuvent être utilisés pour implanter un comportement réactif (i.e. par exemple, un agent peut réagir si un autre agent rentre dans son environnement) [Bouron et al., 91].

Les notions d'agent avec gestionnaire autonome de messages, de groupe et d'agent mobile sensible aux changements dans son environnement spatio-temporel peuvent être formulées en Synergy comme suit :

→ Synergy propose un même gestionnaire (en l'occurrence MethodMgr) pour tous les objets actifs, mais d'autres gestionnaires peuvent être définis et utilisés pour différentes classes d'objets actifs. Une fois le nouveau gestionnaire est défini ; NouvMethodMgr, il suffit d'utiliser, dans la description d'une méthode, le concept [NouvMethodMgr :super] au lieu de [MethodMgr :super].

→ Un groupe peut être réalisé en Synergy comme un objet actif avec une queue de messages, un attribut pour la liste des identificateurs des agents actuellement membre du groupe et essentiellement trois méthodes, traitant respectivement les messages de souscription "subscribe" (ajouter l'identificateur d'un objet dans la liste du groupe), l'inverse "unsubscribe" (enlever l'identificateur d'un objet de la liste du groupe) et informer "inform" qui correspond aux envois de messages. La méthode pour ce dernier type de message envoyerait le même message à tous les objets membre du groupe.

→ Un objet actif peut être "mobile", avec une position spatio-temporelle et il peut être sensible aux déplacements des autres objets. En particulier, un objet peut "posséder" un territoire et réagir à toute "violation du territoire". En Synergy, on peut représenter un territoire comme un concept défini dans un contexte global et avoir des concepts-références dans le corps des objets mobiles qui "possèdent" le territoire et tout changement dans le territoire activera des actions dans les objets concernés (les objets sont ainsi enracinés et situés dans leur environnement).

5.3.2 **Agent réactif**

Un agent réactif ([Demazeau et Müller, 91b], [Maes, 91]) est décrit souvent comme un automate ou comme un "agent enraciné dans son environnement" ("agent embedded in its

environment" [Clark, 87]) avec des "actions situées" ("situated actions theory" [Suchman, 87], [Hickman et Shiels, 91], [Maes, 91]). Selon la théorie des actions situées, divers aspects d'une situation indexent directement un sous ensemble des actions possibles de l'agent [Connah et Wavish, 89], rendant l'agent "sensible" à son environnement. Il peut ainsi réagir directement à une situation qui correspond à un changement de son environnement, contrairement à l'approche "cognitive" (ou "intentionnelle") qui envisage une action comme étant le résultat délibéré d'une décision prise par un agent suite à une intention (un but).

Langages pour les SMAs homogènes "réactifs"

→ **LYDIA** [Connah et Wavish, 90] est un environnement pour les SMAs homogènes "réactifs" : la composante essentielle de LYDIA est le langage concurrent à base de règles **ABLE** (Agent Behaviour LanguagE). Un comportement est décrit à l'aide de règles qui forment un réseau et l'activation se fait par propagation-avant dans le réseau. ABLE permet également la description de dépendances entre les comportements et permet aussi de tenir compte du temps. Wavish et Graham [Wavish et Graham, 95] proposent une version propositionnelle de ABLE, appelée RTA, qui est spécialisée dans le contrôle du comportement d'un agent et non la simulation du SMA en totalité. RTA est conçu pour les SMA hétérogènes. Un programme en RTA est compilé en représentations de "circuits logiques asynchrones". Cette formulation, ainsi que celle qui utilise les diagrammes de transitions d'états [Balch et al., 95] sont très utilisées pour ce type de SMA [Wavish et Graham, 95].

→ Staniford et Paton [Staniford et Paton, 95] proposent une méthode de spécification pour la modélisation et la simulation de sociétés animales en termes d'agents adaptatifs (réactifs) capables de communiquer. Un agent possède des fonctions sensorielles (sensory functions), comme la vision, l'ouïe et l'odorat qui lui permettent de percevoir son environnement. L'ouïe permet à l'agent de communiquer/recevoir des messages à/d'autres agents. Un agent possède par ailleurs un état interne et il peut procéder à un certain ensemble d'actions. Une spécification est formulée selon une logique normative implantée en Prolog.
Les auteurs fournissent comme exemple le cas d'une colonie d'abeilles en défense suite à l'attaque d'un prédateur. Un agent "abeille" possède un état interne, décrit comme un triplet internal_state(current_disposition, actual_orientation, reference_orientation) ; current_disposition est défini sur l'ensemble {calm, aggressive, excited} et les deux autres arguments correspondent à des coordonnées. Parmi les actions possibles que l'agent peut réaliser, citons patrolling, flying et walking.
Voici un exemple de règle normative : une abeille calme, localisée à l'entrée de la ruche doit devenir excitée à la vue d'un objet qui s'approche avec un vol "foncé" quand son

isopentyl_acetate est inférieur au niveau de déclenchement (si son isopentyl_acetate est supérieur au niveau de déclenchement, elle passe à l'état agressif).

```
O( internal_state(excited, hive_entrance, hive_centre) /
    internal_state(calm, hive_entrance, hive_centre)
    seen(darting_flight_intruder)
    smelled(isopentyl_acetate(Level))
    isopentyl_acetate_level < alarm_trigger_level )
```

Cette règle est formulée (avec des détails d'implantation) en Prolog comme suit :

```
internal :-
    internal_state(calm, hive_entrance, hive_centre),
    seen("darting flight intruder"),
    smelled(isopentyl_acetate(Level)),
    alarm_level(Alarm),
    Level < Alarm,
    retract(internal_state(calm, hive_entrance, hive_centre)),
    assert(internal_state(excited, hive_entrance, hive_centre)),
    action .   /*** déclencher le processus responsable de la réaction de l'agent suite à son changement
                    d'état interne ***/
```

Une définition de "action" est fournie comme suit :

```
action :-
    internal_state(calm, hive_entrance, hive_centre),
    seen("darting flight intruder"),
    action(patrolling),
    smelled(isopentyl_acetate(Level)),
    trigger_level(Trigger),
    Level < Trigger,
    retract(action(patrolling)),
    assert(action(threatening)),
     do .
```

Les auteurs notent que les fonctions sensorielles opèrent indépendamment, en concurrence entre elles et avec les autres fonctions. Ils supposent toutefois, pour cet exemple, que les fonctions "internal" (qui teste l'état interne de l'agent), "action" (qui décide quelle action effectuer) et "do" (qui effectue l'action choisie) sont effectuées en séquence.
Considérons à présent la formulation de cet exemple en Synergy : le GC suivant (Figure 5.16) reformule la règle normative qui teste le changement d'état de l'abeille, de calme à

excitée. Il importe de noter que c'est un changement dans l'état interne "intState" ou dans un organe sensoriel (eyes ou ssmell) qui déclenche les opérations associées (Internal, See et Smelled), ce qui reflète fortement le caractère réactif de l'agent. Lorsque le concept [And] est déclenché, les valeurs de ses booléens en entrée sont "true" ou "false" selon que l'état interne de l'abeille corresponde ou non à l'état attendu, qu'elle ait vu ou non l'objet en question et qu'elle ait senti ou non l'odeur en question. Si les trois sont "true" et si la valeur de "alarm" est inférieure à Level, alors [And] retourne "true" et déclenche la mise à jour [UpDateInState], sinon la mise à jour n'est pas déclenchée.

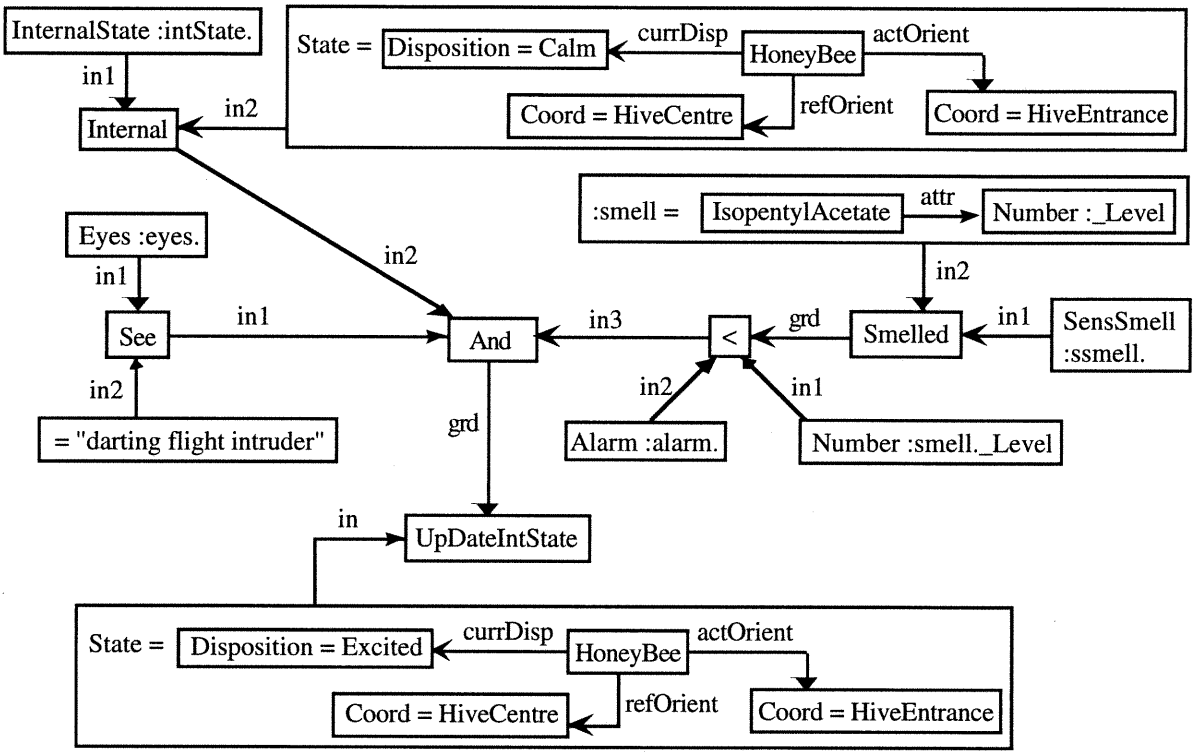


Figure 5.16 : Formulation en Synergy d'une règle normative

La formulation en Synergy de la règle pour l'action est similaire à la précédente.

5.3.3 Agent cognitif

Un agent cognitif est généralement décrit comme une entité dynamique et rationnelle ayant des connaissances, croyances, buts, plans, capacités de raisonnement, des liens "interpersonnels" avec les autres agents (un agent peut décider de collaborer avec tel agent et non avec tel autre) et éventuellement d'autres caractéristiques "cognitives". Au lieu d'échanges "ad hoc" de messages (comme en PCOO), la communication entre agents cognitifs est basée sur la théorie des actes de discours qui formalise la communication humaine ([Werner, 90], [Campbell et D'inverno, 90], [Cohen et al., 90], [Demazeau et

Müller, 90a, 91a], [Shoham, 93], [Genesereth et Ketchpel, 94]). Par ailleurs, la communication peut être "personnalisée" (un agent envoi un message à un autre agent) ou "anonyme" (un agent communique à tous les membres d'une famille d'agents). Les agents peuvent également communiquer via une structure commune appelée tableau noir (blackboard) [Engelmore et Morgan, 88] ; un agent écrit dans le tableau l'information à communiquer et celui qui a besoin d'une information consulte le tableau.

Langages de programmation orientée agents

Shoham [Shoham, 93] considère la programmation orientée agent comme une spécialisation de la programmation concurrente orientée-objet ; un objet actif est utilisé pour implanter un agent, la mémoire locale est utilisée pour implanter l'état mental de l'agent (en termes de connaissances, croyances, capacités, etc.) et la communication par envoi de messages est utilisée pour réaliser une communication par actes de discours.
En l'occurrence, dans le langage AGENT-0 [Shoham, 93], l'état mental est un ensemble de faits temporels que l'agent croit, les capacités correspondent aux actions que l'agent peut réaliser et les règles d'engagement (commitment rules) décrivent le comportement ou la réaction de l'agent suite à la réception d'un message (l'agent s'engage à réaliser des actions, sous certaines conditions, suite à la reception d'un type de message).

La figure 5.17 montre une description partielle en Synergy de l'agent AgentDeVoyage, proposé initialement par [Shoham, 93]. Nous l'avons formulé directement en PCOO "à la" Synergy, en y ajoutant certains éléments "cognitifs" :
→ L'état mental de l'agent est représenté par un concept dont la valeur est la base de faits temporels représentés par une liste de GC. Cette base peut être mise à jour durant l'activité de l'agent et peut être consultée par l'opération "Believe", définie en Synergy, qui vérifie si la base contient un fait qui puisse s'unifier avec son argument.
→ Une règle d'engagement est composée de trois éléments : MessageCondition, MentalConditions et Actions. Si l'agent reçoit un message qui peut s'unifier avec MessageCondition et si l'état mental vérifie MentalConditions, alors l'agent s'engage à réaliser les actions Actions.
Une règle d'engagement peut être implantée comme une méthode avec sa partie "pattern" qui correspond à MessageCondition et son comportement permettant l'activation des Actions conditionnée par MentalCondition.

L'exemple ci-dessous illustre cette formulation pour la règle d'engagement suivante (Figure 5.17): si un message correspond à une demande pour un billet d'embarquement et si l'état mental de l'agent contient le fait "au temps T il reste un nombre de places Nbre pour le vol Vol" et si Nbre > 0, alors l'agent AgentDeVoyage s'engage à réaliser deux actes de discours : un

pour lui-même (il s'engage à réaliser une action à un moment donné), l'autre pour l'agent Client (il s'engage à l'informer de l'état de sa demande).

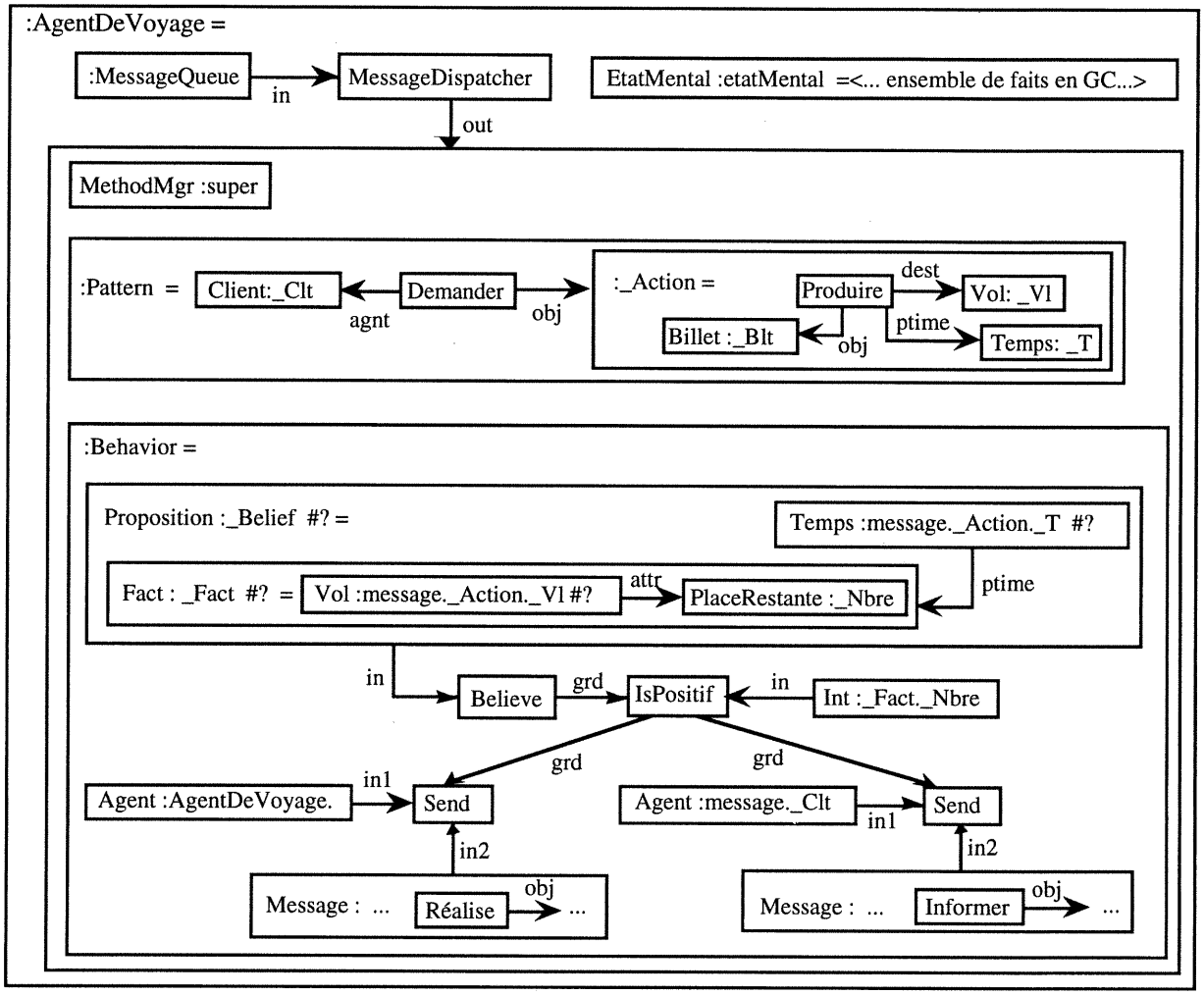


Figure 5.17 : Formulation partielle en Synergy d'un agent "cognitif"

Commentaires sur l'exemple ci-dessus :

→ L'exemple illustre aussi l'utilisation de la co-référence en Synergy. Par exemple, dans la description de Behavior, on réfère à des éléments du message qui a été unifié avec le Pattern de la méthode. C'est le cas de l'identité du client, du numéro de vol et du temps.

→ Un autre point important est le traitement de l'argument de [Believe] : avant de tester si la position existe dans la base, il faut résoudre les co-références car l'unification ne le fait pas. Il faut donc demander à certains concepts de se délencher afin de déterminer leur valeur. C'est le cas par exemple de [Vol :message._Action._Vl #?]. Maintenant, pour que ce concept soit considéré par l'interpréteur, il faut que son contexte soit aussi déclenché afin d'activer et interpréter son contenu. C'est le cas du contexte [Fact :_Fact = ... #?] et aussi du contexte qui

l'englobe [Proposition :_Belief = ... #?]. La structure est ainsi évaluée afin de la compléter (en déterminant les concepts référés) et la préparer pour l'unification.
→ L'agent envoit un message à lui même. Ceci est réalisé facilement en Synergy : il spécifie son propre nom comme destinataire (comme c'est le cas avec le email !).

Autres langages orientés agents

→ Thomas [Thomas, 95] a proposé le langage **PLACA** (PLAnning Communicating Agents) comme une extension de AGENT-0. Contrairement à ce dernier, un agent dans PLACA peut avoir non seulement des actions mais aussi des plans. Il peut donc s'engager à réaliser un plan (et non seulement à effectuer une action). Une règle d'engagement en PLACA est de la forme :

(MessageCondition, MentalCondition, MentalChange, MessageList). Par exemple :

```
( (TO Fred, FROM ?ag, REQUEST, (?time (did ?act))),  /*** si l'agent reçoit ce type de message ***/
    (AND (BELIEVE (*now* (friendly ?ag)))  /*** et si son état mental vérifie ces conditions ***/
        (NOT (PLAN-NOT-DO ((- ?time 1) ?act)))),
    ((ADOPT (INTEND (?time (did ?act))))),  /*** alors il adopte l'intention de réaliser l'acte ***/
    ((TO ?ag, FROM Fred, INFORM, (*now* (intend (?time (did ?act)))))) /** et il envoit un
                                        message pour informer l'agent qui lui a envoyé le message ***/
```

→ Comme nous l'avons effectué pour Synergy, Poggi [Poggi, 95] a reformulé l'exemple proposé par Shoham en **MAPL** (Multiple Agent Programmer Language). Le comportement d'un agent y est décrit comme un "programme", avec les structures de contrôle usuelles (seq, if, while, := et appel de procédures) et des opérations de communication. Ainsi, le comportement de l'agent "airline_clerk" est décrit par une séquence d'instructions "if" qui testent les types de messages que l'agent peut recevoir, avec le traitement associé (en Synergy, cette séquence des "if" correspondrait aux méthodes de l'agent). Voici un exemple :

```
(if (received inmsg '(?.customer request ? (book (flight + .flinfo)) :reply_with ?.label ))
    (if (is_free_seat (cons 'flight flinfo))
        (accept customer :in_reply_to label)
        (reject customer :in_reply_to label) ))
```

→ Une approche similaire est adoptée par le groupe Weerasooriya et al. [Weerasooriya et al., 95]. Ces derniers notent que la recherche dans le domaine des SMA est assez mûre pour pouvoir extraire ce qui caractérise la notion d'agents : 1) la présence d'un état mental complexe qui inclut la croyance, les désires (buts), les plans et les intentions, 2) comportements proactif (dirigé par les buts) et réactif, 3) communication via des messages structurés ou des actes de discours, 4) possibilité d'être distribuée sur un réseau, 5) capacité d'agir et de réagir en temps-réel (de quelques secondes à quelques minutes) aux changements

de l'environnement, 6) exécution concurrente de plans au sein d'un agent et entre agents, et 7) capacité de méta-raisonnement ou raisonnement réflexif.

Les auteurs proposent la conception (la réalisation reste à faire) d'un langage, appelé **AgentSpeak**, qui incorpore les caractéristiques ci-dessus (en fait quelques caractéristiques).

En particulier, un agent dans AgentSpeak est créé par instanciation d'une classe d'agents. Un agent possède une base de croyances (appelée base de relations), des buts (appelés services) et des plans pour réaliser les buts. Une relation est définie par un prédicat typé (par exemple: position(int Xpos ; int Ypos)) et une relation particulière est créée par instanciation (par exemple: create-relation-instance(position :: Xpos:3, Ypos:4)). Enfin, on peut accéder à un champ de la relation (par exemple: position.Xpos).

Au plan sont associés un nom, le service/but à réaliser, le contexte d'application (qui correspond aux pré-conditions du plan), le corps du plan et enfin les relations à ajouter à la base des relations de l'agent. Le contexte d'application est composé d'un ensemble de relations à vérifier (par rapport à la base des relations de l'agent).

Le corps d'un plan est une séquence "d'instructions", dont chacune peut être une assignation, une alternative (if-then-else), une itération (while), une disjonction non-déterministe d'instructions, un but à réaliser ou un acte de discours. Un plan correspondrait à une "procédure cognitive" et un but à réaliser à l'appel au plan.

Les instructions "actes de discours" permettent la communication entre agents par envois de messages. AgentSpeak offre trois actes (les auteurs notent que des actes plus complexes, comme ceux offerts par **KQML** [Genesereth et Ketchpel, 94] peuvent être définis facilement à l'aide de ces trois actes) : *inform* qui correspond à un envoi asynchrone de message (ce dernier correspond à une relation), *request-with-wait* qui correspond à un envoi synchrone de message (l'agent envoit le message avec la relation que le destinataire doit lui retourner, il attend ensuite la réponse) et *request with-no-wait* où l'agent envoit au destinataire le message avec la relation à retourner mais n'attend pas la réponse, ce n'est que lorsqu'il en aura besoin qu'il attendra.

Un agent peut communiquer avec un autre agent dont il connait le nom, tous les agents instances d'une classe ou un agent (dont le nom lui est inconnu) d'une classe d'agents.

Alors que les langages précédents ont une syntaxe "à la" Lisp, AgentSpeak a une syntaxe "à la" C++. Considérons quelques extraits de l'exemple du robot, bolt-robots, fourni par les auteurs :

```
agent-family bolt-robots {
    public :
        database {position; rivet-box; bolt-box}
        services {make-bolts; deliver-bolts}
    private :
        database {..autres relations..}
        services {move, grasp-rivet, ...}
        plans {rivets-to-bolts; ...; pick; ...}
}
```

```
relation position (int  X-pos; int Y-pos)
relation rivet-box (int Quantity; int X-pos;  int Y-pos)
...
service  make-bolts  (achieve  bolt-box)  /*** un service peut réaliser une relation, demander si la relation
                                              existe ou informer l'agent d'une nouvelle relation **/
...
plan rivets-to-bolts {
    invoke  on  make-bolts(achieve,  bolt-box(Quantity, X-pos, Y-pos))
    with  context
       ((rivet-box.Quantity  > 0)  And  ...  And  holding(false, rivet))
    perform
       while  (rivet-box.quantity  > 0)  do  {
          move(achieve, position(rivet-box.X-pos,  rivet-box.Y-pos)) ;
          grasp-rivet(achieve, holding(true, rivet)) ;
          ...
          request-with-wait(Lathe-inst, position-request, 1, position(Lathe-x-pos,  Lathe-y-pos)) ;
          move(achieve, position(Lathe-x-pos,  Lathe-y-pos)) ;
          ...
          ungrasp-bolt(achieve, holding(false, bolt))
       }
}
```

La base des relations et les plans constituent donc deux éléments essentiels d'un agent AgentSpeak ; les deux peuvent être formulés en Synergy :

→ La base des relations et les primitives associées (créer une relation par instanciation, accéder à un champ de la relation, ajouter, éliminer et chercher une relation dans la base) peuvent être formulées en Synergy.

→ Le plan qui réalise un service et qui correspond à une procédure "cognitive" avec des structures de contrôle, des appels de procédures et des opérations de communication. Tous ces éléments ont leur correspondants en Synergy.

Notons en l'occurrence la similitude de l'extrait ci-dessus avec notre exemple sur la modélisation orientée agent de l'unité des soins intensifs (voir l'annexe 7). Dans les deux cas on a des types d'agents (et non seulement des agents particuliers); un agent possède une partie structurelle ou descriptive (en AgentSpeak c'est l'ensemble des relations, en Synergy c'est une partie du GC) et est capable de réaliser différentes tâches, dont chacune peut être décomposée en sous-tâches et/ou actions (dans AgentSpeak ce sont les services/plans), avec possibilité d'envoi et/ou attente de messages à/d'autres agents.

Les quatre langages (AGENT-0, PLACA, AgentSpeak, MAPL) permettent la conception de système multi-agent composé d'agents "cognitifs" capablent de communiquer par des actes de discours. Nous allons considérer dans la prochaine section les SMA hétérogènes, composés d'objets "statiques" et d'agents réactifs et pro-actifs.

5.3.4 Système multi agents

Un *système multi agent* (SMA) est un groupement (société) d'agents. Les SMA diffèrent selon la nature des agents qui les composent. On distingue par ailleurs les *SMA homogènes* (tous les agents sont d'un même type) des *SMA hétérogènes* (les agents peuvent être de types différents).

Dans les SMA où les agents sont "purement" réactifs, il n'y a pas de communication entre agents [Hickman et Shiels, 91]. Si leur interaction ressemble à une coopération, elle résulte en fait du rapport étroit qui relie les actions de chacun à l'environnement qu'ils partagent. En d'autres termes, les actions de certains agents changent l'environnement et un tel changement affectera d'autres agents [Connah et Wavish, 90].

En général, l'environnement dans un SMA "réactif" comprend des agents aussi bien que des objets "statiques" et c'est un changement dans l'état d'un objet qui pourrait susciter la réaction d'un ou plusieurs agents. Par contre, l'environnement dans la plupart des SMA "cognitifs" se réduit à un ensemble d'agents "cognitifs" qui communiquent entre eux par envoi de messages.

Les SMA hétérogènes adoptent une vision plus "réaliste" de la notion d'environnement : un agent peut être réactif, cognitif ou les deux à la fois et l'environnement peut contenir également des objets "statiques".

Langages pour les SMAs hétérogènes

→ **MYWORLD** [Wooldridge, 95] comprend un langage agents, un monde et son gestionnaire (world shell and world manager) et un scénario. Le monde est composé d'objets "statiques" et d'agents situés dans un cadre spatio-temporel. Un scénario correspond à une expérimentation particulière.

Le langage agent de MYWORLD permet de caractériser un agent par : 1) un ensemble de croyances (des faits "à la" Prolog) qui peut être modifié par des règles de croyances, par une fonction de révision de croyances ou par des actions de l'agent, 2) un ensemble d'intentions avec des règles d'adoption d'intentions.

Les règles de croyances sont activées par chaînage avant : dès que la partie condition d'une règle est vérifiée, la partie action est exécutée produisant une mise à jour de l'ensemble des croyances. La fonction de révision de croyances encode la capacité de l'agent à percevoir son monde et à modifier ses croyances et réagir en conséquence.

Les intentions représentent les désirs que l'agent voudrait réaliser. Une intention est composée d'un but, d'une motivation et du degré de priorité. La motivation correspond aux préconditions du but ; ce que l'agent doit croire pour que le but soit maintenu. Le but avec la plus grande priorité constitue le but courant. Les intentions sont générées par les règles

d'adoption d'intentions, chacune est composée d'une condition d'adoption (qui est à apparier avec les croyances de l'agent) et d'une intention paramétrée.
Enfin, des règles stratégiques spécifient pour une intention un "plan procédural" qui peut la réaliser.
Le comportement d'un agent consiste donc à mettre à jour ces croyances et ces intentions, à considérer l'intention la plus prioritaire et à exécuter ensuite le plan qui lui est associé.

Remarque : Bien que les langages AgentSpeak et MAPL soient proposés pour des SMA hétérogènes (agent réactif aussi bien que pro-actif), le côté réactif de l'agent n'est pas clair, contrairement à MYWORLD et en particulier, à sa notion de fonction de révision de croyances qui relie explicitement la perception aux croyances de l'agent (l'agent possède "explicitement" une capacité de perception, avec son impacte sur les croyances de l'agent).
Aussi, la notion de système est réduite dans AgentSpeak et MAPL à l'ensemble des agents, sans considérer la présence d'objets et leur manipulation par les agents.
Toutefois, les détails concernant les plans/actions, les objets "statiques" et la manipulation des objets par les agents ne sont toutefois pas spécifiés dans [Wooldridge, 95]. Aussi, Wooldridge ne considère ni la notion de groupe d'agents et la structure socials dans les SMA, ni la possibilité d'avoir des processus (qui peuvent simuler des machines) dans l'environnement, en plus des agents et des objets "statiques", et l'interaction entre les processus et les agents.

→ Une approche différente a été adoptée dans **RATMAN** [Bürckert et Müller, 91] qui concerne également la modélisation des SMA hétérogènes. En adoptant une formulation logique, Bürckert et Müller décrivent un agent "rationnel" comme une base de connaissance hiérarchique à plusieurs niveaux : 1) un niveau sensoriel qui concerne les organes sensoriels de l'agent, 2) un niveau "physique" qui concerne les connaissances sur le temps, l'espace, le sens commun et une certaine expertise (connaissance d'un domaine particulier), 3) un niveau action qui concerne les actions possibles, 4) un niveau communication basé essentiellement sur l'utilisation d'un tableau noir, 5) un niveau planification qui concerne les plans de l'agent, 6) un niveau "méta" qui concerne la connaissance de l'agent sur lui-même et sur d'autres agents et 7) le niveau apprentissage.
En principe, chaque niveau fait appel aux niveaux inférieurs et l'hétérogéneité des agents provient de la présence ou non de tel ou tel niveau (un agent réactif aura uniquement les niveaux 1 et 3).
En plus des agents, un SMA dans RATMAN contient également des objets "statiques" décrits dans une partie du tableau noir. Enfin, en tant que société d'agents, un SMA lui est consacré un module pour décrire le type d'organisation ou de groupement entre les agents.

Plusieurs autres SMA hétérogènes, se rapprochant de l'un ou l'autre des deux modèles ci-dessus, ont été proposés en littérature ([Lizotte et Moulin, 89], [Cardozo et al., 93], [Ferguson, 95], [Huang et al., 95], [Wavish et Graham, 95]). Notons, pour rendre justice et homage à Hendrix, qu'une première modélisation "ad hoc" d'un système multi agents hétérogène a été proposé par Hendrix [Hendrix, 73].

Synergy comme langage pour les SMA

Synergy n'est pas un langage dédié à une forme particulière de programmation orientée agents, c'est un langage multi-paradigme qui peut être utilisé pour définir et expérimenter différents modèles d'agents et de systèmes multi-agents (nous considérons cette généralité comme un avantage du langage)

Des systèmes multi-agents hétérogènes, similaire à MYWORLD peuvent être formulés en Synergy. En fait, le caractère multi-paradigme de Synergy permettrait la formulation et l'expérimentation de SMAs "plus" hétérogènes : l'environnement peut être composé d'objets "statiques", d'agents de différents types (agents purement réactifs, purement cognitifs et/ou agents "hybrides") et de processus (des machines, des systèmes informatiques, etc.).

Pour formuler un agent réactif, capable de percevoir et de réagir aux changements de son environnement, l'utilisateur de Synergy peut utiliser les liens in/out et/ou la co-référence, les deux permettent l'implantation d'une dépendance entre l'environnement et l'agent, dépendance qui est à la base de la "perception".

Aussi, un agent réactif (ou certaines de ses actions) peut être soumi à certaines contraintes "sociales" ; attendre qu'un autre agent (ou une machine) termine son activité, attendre qu'une certaine condition qui porte sur des éléments de l'environnement soit respectée, etc. De telles contraintes peuvent s'exprimer en Synergy en utilisant les relations procédurales du langage.

Les capacités cognitives d'un agent peuvent aussi être formulées en Synergy : la base de croyances correspondrait à un concept [BaseCroyance : Croyances = GCs] dont la valeur est une collection de GCs. Des opérations de recherche, d'ajout et d'élimination de croyances peuvent être définie en Synergy. Le langage Prolog+CG peut être utilisé pour gérer les inférences reliées à une base de croyance, ayant ainsi un exemple d'intégration très forte de Synergy et Prolog+CG.

Enfin, différentes formes de plans et d'actes de discours peuvent être définies en Synergy (pour les actes de discours, ils peuvent être implantés à l'aide des opérations de communication offertes par le langage).

Notons par ailleurs que le travail qui a été effectué par plusieurs chercheurs sur le traitement du langage naturel avec les GC peut être utilisé pour munir les agents (en Synergy) de capacités linguistiques (compréhension, génération et discours). Enfin, si nous considérons notre modèle d'organisation et de formation dynamique de la mémoire, un agent pourrait avoir une mémoire dynamique et il serait capable d'apprendre et de réaliser une "programmation orientée mémoire".

5.4 Résumé

Nous avons montré dans ce chapitre la possibilité d'utiliser Synergy pour la programmation de processus concurrents, puis nous avons introduit une réalisation de la programmation concurrente orientée objet en Synergy. Des exemples ont été proposés ainsi qu'une discussion mettant en rapport notre réalisation avec d'autres travaux dans le domaine. Nous avons ensuite considéré le domaine des systèmes multi-agents. Après une brève introduction du domaine, nous avons considéré les agents cognitifs, les agents réactifs et les systèmes multi-agents. Pour chacun des cas, nous avons présenté des exemples de langages proposés dans la littérature et nous avons souligné comment Synergy peut être utilisé pour réaliser le même type de traitement. Enfin, nous avons conclue avec une note "optimiste" pointant à une perspective de recherche à long terme; le développement d'un environnement général pour les systèmes multi-agents, basé sur Synergy (et les autres composantes de notre système).

Concluons cette partie sur le langage Synergy (les chapitres 3 à 5), en précisant que ce dernier incorpore une vision simple et uniforme du traitement, basée sur la représentation de toute connaissance en GC (aussi bien déclarative que procédurale) et sur une interprétation par activation de graphe. Le langage fournit un minimum d'éléments de base qui sont communs aux utilisations possibles du langage (comme langage procédural, fonctionnel, ou orienté objet, etc) et, pour plus d'efficacité, fournit quelques éléments de "contrôle" (en particulier les attributs optionnels reliés aux relations).
La structure de base de Synergy, les GC, peut être interprétée comme : une structure "déclarative", une procédure, un démon, une fonction, un processus, une tâche, un système, un agent, ..., selon les besoins du programmeur. Evidemment, ce dernier n'a pas à les connaître tous pour utiliser les GC selon son besoin !
Selon son intérêt, l'utilisateur peut utiliser Synergy comme : un langage procédural séquentiel, un langage procédural parallèle, un langage fonctionnel, un langage pour les "frames", un langage orienté événement, un langage concurrent orienté objet, etc. L'utilisateur peut donc avoir une vue et un besoin "mono"-paradigme et "ainsi soit-il".
Par ailleurs, des applications nécessitant l'utilisation conjointe de plusieurs paradigmes inciterait l'utilisateur à *chercher* et à exploiter le caractère multi-paradigme du langage, et encore; s'il est interéssé uniquement par la programmation fonctionnelle et procédurale, il n'a pas à se soucier des possibilités de programmation concurrente orientée objet fournies par Synergy.
Enfin, souligons qu'une "connaissance globale" de "l'approche" Synergy est souhaitable, on comprendrait mieux pourquoi il peut "réaliser" plusieurs paradigmes et comment il peut "réaliser" d'autres.

Chapitre 6

Formation incrémentale de la mémoire à long terme

Ce chapitre présente un modèle de formation dynamique et incrémentale de la mémoire à long terme (MLT), appelé Mémoire Basée sur l'Intégration (MBI). En réponse à un flot continu de nouvelles informations, le modèle spécifie comment la MLT se développe et se réorganise.
Le modèle constitue ainsi une alternative à la gestion explicite de la MLT par le programmeur Synergy et il peut constituer également la "primitive" de base dans une programmation orientée mémoire.
Après l'introduction, nous présentons une description générale du modèle concernant essentiellement les types d'information à intégrer en MLT et les principes de base sous-jacents au modèle MBI. Une description algorithmique complète et précise la description générale. Dans la dernière partie, nous décrivons brièvement l'environnement graphique, qui supporte le processus d'integration, ainsi que les utilisations possibles du modèle.

6.1 Introduction

Un agent intelligent intègre constamment de l'information dans sa mémoire. Qu'elle est cette information ?, comment est-elle integrée en mémoire ? et quelle est l'organisation de cette dernière ? Dans le cadre de la théorie des graphes conceptuels [Sowa, 84], ces questions peuvent être reformulées comme suit : comment de nouvelles structures conceptuelles (définitions de types de concepts, schémas, canons et instances) sont-elles intégrées en mémoire (en l'occurrence dans une base de connaissances) ?, comment cette intégration peut-elle susciter la formation d'autres structures en mémoire ?, comment cette dernière est réorganisée suite à de tels ajouts ?, quel est le rapport entre les structures conceptuelles (par exemple entre "définition", "canon" et "schéma"), un schéma peut-il devenir une définition ?, comment la notion d'attention ou de focus (importante en psychologie cognitive) peut-elle être considérée dans ce cadre ?, quel est l'impact de la notion de focus sur le processus d'intégration ?, etc.
Cette problématique n'a pas été considérée dans la communauté des GC, elle se présente toutefois, naturellement, dans le cadre de notre environnement graphique et dans celui de

Synergy en particulier ; la mémoire à long-terme d'une application Synergy peut être formée manuellement : le programmeur utilise le mode édition pour ajouter, éliminer et/ou modifier son contenu. Elle peut par ailleurs être formée de façon dynamique, avec le modèle de formation de la mémoire que nous proposons.
Cette problématique est aussi fondamentale pour le traitement du langage naturel : relations des mots aux concepts, définition et utilisation des mots, formation et "évolution" des "sens" des mots, problèmes de synonymie, etc. Elle est importante également pour les systèmes à base de cas ([Riesbeck et Schank, 89], [Slade, 91], [Trappl, 92], [Kolodner, 93], [Wess et al., 93], [Veloso et Aamodt, 95]) qui utilisent *une mémoire dynamique* [Schank, 82, 86, 88, 91] et aussi en apprentissage symbolique et plus particulièrement en apprentissage multi-stratégies [Tecuci, 94, 95].

Notre objectif de base est donc de concevoir et développer, dans le cadre de notre environnement graphique et dans Synergy en particulier, un modèle général de formation incrémentale d'une mémoire dynamique, qui puisse être utilisé dans différents domaines et par différents processus (séparement ou conjointement).

Dans le chapitre 2 et en particulier la section 2.6, nous avons résumé notre revue des travaux en spécifiant les points à retenir : formation incrémentale et non-supervisée de la mémoire, importance d'une mémoire basée sur la généralisation et importance de la notion d'éléments pertinents sans toutefois les réaliser avec une préindexation (selon un réseau de descrimination) ; la mémoire devrait être non-indexée. Le modèle que nous proposons est concerné par la problématique mentionnée ci-dessus (intégration automatique de structures conceptuelles dans une base de connaissances) et intègre les "points à retenir" pour la construction d'un modèle de la mémoire dyanmique.

Brièvement, nous proposons un modèle de la *mémoire basée sur l'intégration* (MBI) dans lequel la "mémoire" correspond à la mémoire à long terme (MLT) de Synergy et une nouvelle "information" à intégrer peut correspondre à une définition, à un schéma, à une description d'instance ou à une déclaration de synonymie de type. Le processus d'intégration reçoit en entrée une nouvelle information, y applique une procédure "d'explicitation" des connaissances, demande ensuite à la source de l'information (qui peut être l'utilisateur ou un autre processus, en l'occurrence un processus de traitement du langage naturel) les éléments pertinents (en-focus) dans la nouvelle information et il amorçe ensuite l'intégration de l'information en mémoire. Les éléments pertinents, qui sont des concepts, "guident" le processus d'intégration : ce dernier localise en mémoire "les nœuds" associés aux éléments pertinents et la nouvelle information est propagée dans la mémoire via ces nœuds. Enfin, au cours

de l'intégration, les similitudes et les différences entre la nouvelle information (ou structure) et celles qui existent déjà en mémoire sont établies.

Remarques :

→ Contrairement au modèle de la mémoire dynamique de Schank [Schank, 82, 91], les éléments pertinents sont identifiés pour chaque nouvelle information (et non pour toute l'application) et ils ne sont pas réalisés dans le modèle comme des indexes (préfixés ou non) mais correspondent plutôt à des concepts qu'il faut localiser dans la mémoire. L'intégration de la nouvelle information en mémoire est effectuée selon ces concepts. Une telle interprétation des éléments pertinents "libère" le modèle d'une mémoire dynamique du besoin d'une mémoire indexée, tout en préservant les raisons pour lesqels l'indexation a été adoptée (les éléments pertinents doivent influencer et guider le processus d'intégration).

→ Un canon pour un type de concept est considéré, dans le cadre actuel de notre modèle, comme un schéma général "proche" du concept. Un concept peut avoir plusieurs canons (et non un seul) et, plus un schéma est général et est en commun à plusieurs situations, plus il peut être "canonique".

6.2 Description générale du modèle de la mémoire basée sur l'intégration (MBI)

Comme nous l'avons souligné ci-dessus, le but du processus d'intégration est d'intégrer une nouvelle information en Mémoire à Long Terme (MLT). Une nouvelle information peut être la définition d'un type, la description d'une instance ou d'un schéma, ou la déclaration de synonymie entre types de concept.

La prochaine section introduit certains points concernant l'information à intégrer, des spécifications pour le processus d'intégration sont alors identifiées.

6.2.1 Caractéristiques de l'information à intégrer

Nous soulignons dans cette sections quelques points reliés aux types de d'information qui peut être intégrée en mémoire. En particulier :

♦ Un *schéma* représente l'utilisation de types de concepts et d'instances dans la description d'une structure de connaissances qui peut être spécifique ou générique et qui peut correspondre à différents types de connaissances (fait, action, script, plan, situation, structure d'explication "explanation pattern ", structure d'une histoire "story pattern", etc.).

♦ La *déclaration de synonymie de type* est importante pour le processus d'intégration car plusieurs identificateurs peuvent être utilisés pour désigner le même type. Elle l'est encore plus si la source d'information est multiple.

Par exemple, le terme "AssociativeNetwork" peut être utilisé dans un schéma et "SemanticNetwork" dans un autre bien que les deux aient la même signification.
Pour identifier les similitudes et les différences entre les structures dans la MLT, le processus d'intégration doit considérer la synonymie des types de concept.

♦ *Une nouvelle information peut contenir la définition d'un type*. Par exemple, le processus d'intégration pourrait avoir en entrée le GC suivant qui représente la situation "Karl observes a bird that eats fish and possesses a high beak" :

[BOY:Karl]<-agnt-[OBSERVE]-object->[BIRD]<-agnt-[EATER]-obj->[Fish]
|-possession->[BEAK]-size->[MEASURE: High].

Et dans la MLT on pourrait avoir la définition suivante d'un Pelican :

[Pelican :_P = [BIRD]<-agnt-[EATER]-obj->[Fish]
|-possession->[BEAK]-size->[MEASURE: High]].

Cette situation peut survenir si par exemple, la source d'information est multiple : la MLT peut contenir des définitions de types qui sont inconnues par la source actuelle de l'information. Le processus d'intégration doit tenir compte de cette différence de formulation de l'information. Ainsi, il doit reconnaître que "Karl observes a bird that eats fish and possesses a high beak" est identique à "Karl observes pelican" et il doit informer éventuellement la source d'information de l'existence de la définition de pélican dans la MLT.

Une phase *d'explicitation de l'information* est donc nécessaire : le processus d'intégration devrait analyser la nouvelle structure conceptuelle, identifier les définitions contenues dans la structure et ensuite rendre *explicite,* dans la structure, les types correspondants aux définitions. Ainsi, dans l'exemple précédent, le processus d'intégration devrait remplacer [BIRD] par [PELICAN].

Note : Bien que "la racine" est commune, notre utilisation du terme "explicitation" est différente de celle adoptée en acquisition des connaissances [Diaper, 89] où elle correspond à un ensemble d'activités, réalisées par une personne "knowledge elicitor", visant à obtenir du matériel de toute source importante, à analyser, à interpréter ce matériel et à mettre le résultat en une forme pré-codée [Cordingley, 89].

♦ *Remarques sur la relation entre "définition" et "schéma"*

La mémoire contient aussi bien les définitions que les schémas et une interaction dynamique existe entre les deux types de connaissances. En effet, la définition peut être

une spécialisation d'un schéma et un schéma peut être ré-interprété comme une définition. Par exemple :

→ un utilisateur peut intégrer le schéma suivant (le fait "there are birds with big beak") : [BIRD]-possession->[BEAK]-size->[MEASURE:High]. Par la suite, une nouvelle information, fournie par le même utilisateur ou par un autre, pourrait être la définition de PELICAN : "a pelican is a bird that eats fish and has high beak". Le processus d'intégration devrait identifier et établir que cette définition est une spécialisation du schéma précédent.

→ Après une suite de généralisations, le processus d'intégration pourrait créer un schéma décrivant le fait : "there are birds that eat fish and have big beaks". Par après, la définition de PELICAN pourrait être intégrée, le processus d'intégration devrait alors noter que la définition correspond au schéma. Comme le montre la section 6.3, cette mise à jour pourrait impliquer d'autres modifications.

♦ *Les informations (définition, schéma et synonymie) peuvent être fournies comme entrée au processus d'integration et elles peuvent être aussi "inférées ou suggérées" par ce même processus.*

→ Une nouvelle structure propagée en mémoire est comparée aux structures existantes en MLT, si les deux structures comparées ont seulement des parties en commun, le processus d'intégration *crée un nouveau schéma* avec comme contenu les parties communes aux deux structures. Au moins trois scénarios sont possibles concernant "l'évolution" d'un tel schéma : 1) il peut demeurer un schéma, 2) l'utilisateur pourrait intégrer par la suite la définition d'un type et le processus d'intégration découvre que la définition est identique au contenu du schéma, ou 3) un mécanisme de focus qui observe l'évolution de la mémoire peut découvrir "l'importance" du schéma (par exemple, il est spécialisé par plusieurs définitions et/ou schémas) et informer l'utilisateur d'une telle "observation" (afin d'envisager le schéma comme une définition d'un type). Certaines questions d'enfants ne résultent-elles pas d'observations similaires ?

→ Un utilisateur peut intégrer la définition d'un type T et le processus d'intégration découvre que la définition existe déjà en mémoire mais pour un autre type T1. Le processus d'intégration devrait alors enregistrer sa "découverte" et en informer l'utilisateur ; lui indiquer que "T est en fait un synonyme de T1".

6.2.2 Description générale du processus d'intégration

Le processus d'intégration comprend un *mécanisme de connexion* (effectué par défaut) qui est généralement complété et raffiné par un *mécanisme d'intégration par propagation*.
Le mécanisme de connexion est une opération d'intégration de base : elle consiste à indexer la nouvelle information sous les concepts qui la compose. La Figure 6.1 montre un exemple d'intégration d'un schéma basée uniquement sur le mécanisme de connexion.

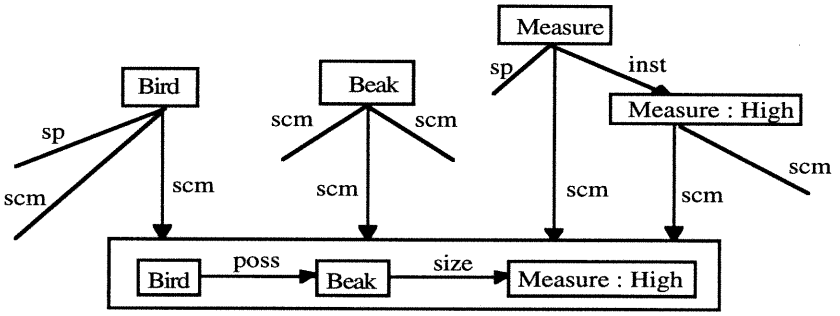


Figure 6.1 : Intégration d'un schéma basée uniquement sur le mécanisme de connexion

Ainsi, pour tout type (ou instance) déclaré dans la MLT nous avons immédiatement et explicitement, via les liens "scm", les structures (définitions, instances ou schémas) dans lesquelles le type ou l'instance est utilisé. Si une nouvelle information est intégrée et si, les similitudes et "rappels" (reminding) détectées par le mécanisme d'intégration par propagation ne suffisent pas, on peut toujours alors explorer les "similitudes directes"; celles offertes par le mécanisme de connexion. En intégrant par exemple une nouvelle information qui contient le concept [Measure], toutes les autres informations qui contiennent également ce concept peuvent être directement accédées. Sans le mécanisme de connexion, on ne peut identifier dans la mémoire les structures qui contiennent le concept [Measure], à moins d'une recherche exhaustive, qui est inappropriée.

Remarque : L'organisation de la mémoire dans le "raisonnement basé mémoire" (Memory Based Reasoning [Waltz, 90], [Kettler et al., 94]) est basée sur le premier mécanisme alors que le modèle de la mémoire basée sur la généralisation (Generalization-Based Memory [Schank 82], [Lebowitz 86], [Schank et Reisbeck 89], [Kolodner, 94]) peut être considéré comme une forme restreinte du "mécanisme d'intégration par propagation" proposé dans ce chapitre.

Considérons maintenant le mécanisme d'intégration par propagation, il est initié par les éléments en focus et il porte uniquement sur une entrée qui est une *définition* ou un

schéma, l'intégration d'une *déclaration de synonymie de type* consiste simplement à ajouter le synonyme dans la liste des synonymes du type (s'il n'y est pas déjà) et l'intégration d'une *description d'instance* utilise uniquement le mécanisme de connexion.
L'intégration d'une structure S qui correspond à *la définition d'un type* ou *la description d'un schéma* est basée sur trois étapes : 1) explicitation de la structure S, 2) intégration de S selon le mécanisme de connexion et 3) propagation éventuelle de S en MLT. Considérons chacune de ces étapes.

→ *Explicitation*

Dans notre formulation antérieure du modèle de formation de la mémoire [Kabbaj, 95], l'opération d'explicitation est effectuée durant la propagation. Dans la présente version, les deux traitements sont séparés ; l'explicitation précède la propagation de l'information en mémoire. La définition et la réalisation de la propagation sont rendues plus simples et plus efficaces. Aussi, la propagation peut se faire à présent en parallèle alors que dans la formulation antérieure elle devait être séquentielle.
L'explicitation d'une structure conceptuelle S est un traitement itératif qui considère à chaque cycle si la structure S à intégrer contient la définition d'un type T qui existe déjà en MLT. Si oui, le type T remplace dans S le type correspondant et les parties de S qui correspondent à la définition de T sont identifiées (en l'occurrence, elles sont colorées comme le précise la section sur l'environnement graphique). Reprenons l'exemple de Karl et du Pelican, la situation S pourrait être le schéma suivant :

[BOY:Karl]<-agnt-[OBSERVE]-object->[BIRD]<-agnt-[EATER]-obj->[Fish]
|-possession->[BEAK]-size->[MEASURE: High].

L'opération d'explicitation, appliquée sur le GC ci-dessus va constater qu'il contient la définition de Pelican :

[Pelican :_P = [BIRD]<-agnt-[EATER]-obj->[Fish]
|-possession->[BEAK]-size->[MEASURE: High]].

Elle va donc remplacer dans S le type BIRD par le type Pelican et identifier les parties qui correspondent à sa définition :

[BOY:Karl]<-agent-[OBSERVE]-object->[Pelican]**<-agnt-[EATER]-obj->[Fish]**
|-possession->[BEAK]-size->[MEASURE: High].

Après ce traitement itératif, le processus d'explicitation demande à l'utilisateur de mettre à jour (s'il le désire) la structure S ; les parties qui ont été identifiées lui montrent les redondances dans S. L'opération d'explicitation n'élimine pas automatiquement ces parties car chacune d'elle peut être utile à la connexité de la structure ou servir à identifier des

similitudes entre structures. Toute partie redondante peut donc être utile et c'est à l'utilisateur de décider s'il faut l'éliminer ou la garder.

→ *Intégration par connexion*

Après l'explicitation, la structure S (éventuellemment mise à jour) est ajoutée en MLT en la reliant aux concepts qui la décrivent.

→ *Propagation éventuelle en MLT*

Dans notre modèle, l'utilisateur (ou autre source d'information) peut considérer certains concepts de la structure S à intégrer comme *importants* ou *pertinents* (relevants) ; certains concepts sont plus *en focus* que d'autres.

Si l'utilisateur fournit une telle information pour S, alors l'indexation par défaut sera complétée et raffinée par un processus de propagation et "d'intégration en profondeur". En effet, nous supposons que si un concept est jugé important ou pertinent, alors il serait approprié de comparer S avec les structures existantes dans la MLT, qui contiennent le concept en question. C'est une interprétation opérationnelle de la notion de "focus" ou "attention", notion importante en psychologie cognitive. Notre interprétation est opérationnelle car elle montre comment la notion de focus peut influencer le processus d'intégration et donc la formation et la réorganisation de la mémoire.

En particulier, si des concepts de S sont indiqués comme importants (ou en focus), alors S sera propagée dans la MLT via ces concepts : pour chaque concept en focus, le processus d'intégration localise dans la MLT le noeud qui définit le type du concept (et du référent si c'est un identificateur d'instance) et propage la structure S aux "fils " du noeud localisé afin de comparer S avec chacun des fils et poursuivre la propagation selon le résultat de la comparaison.

La structure S peut être plus-général-que (>), égal-à (=), plus-spécifique-que (<), avoir-des-éléments-en-commun-avec (Int) ou n'a-pas-d'éléments-en-commun-avec (<>) la description du noeud courant.

La Figure 6.2 illustre les transformations générales associées aux cas ">", "<" et "Int", quand la structure S est propagée du noeud P au noeud fils N. Si S > N (S est plus général que N) alors S est placée entre les noeuds P et N, si S < N (S est plus spécifique que N) alors S est propagée aux fils de N, si S "Int" N (S et N ont seulement quelques éléments en commun) alors un nouveau noeud est créé et placé comme indiqué dans la figure, son contenu est le résultat de la généralisation de S avec N.

L'algorithme dans la section 6.3 fournit le détail des processus d'intégration et de propagation.

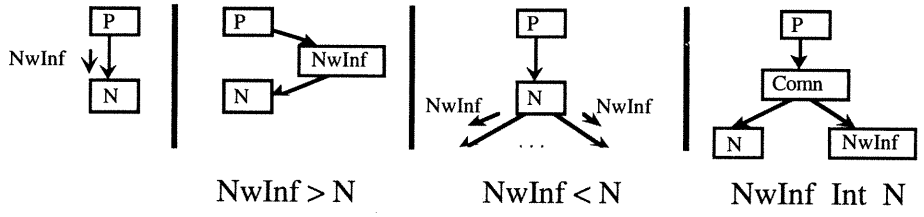


Figure 6.2 : Transformations générales associées aux cas ">", "<" et "Int"

Remarques :

→ La même information intégrée avec des éléments pertinents différents aurait des effets différents sur la mémoire et activerait des "rappels" différents (des nœuds différents peuvent être visités).

→ L'intégration d'une information en mémoire est hautement "parallèle" ; intégration via différents chemins (initialement via les éléments pertinents et ensuite la propagation à tous les fils d'un nœud) qui peut se faire en parallèle. Cela permet une génération rapide de plusieurs rappels qui peuvent être évalués et combinés de différentes façons. Rappelons ici la note de [Kettler et al., 94] sur le caractère rapide et hautement associatif de la recherche analogique chez l'humain (human analogy retrieval) et du fait que plusieurs rappels sont générés.

6.3 Description algorithmique du processus d'intégration

Afin de préciser et de compléter la description générale du processus d'intégration, nous présentons dans cette section la formulation algorithmique du modèle.

procedure Integrate1(in Input: ConcOr2Types, LFocus: ListOf Concept) :

la procédure traite trois cas : déclaration de synonymie, description d'instance et description d'une définition ou d'un schéma.

→ Pour la description d'instance, nous considérons le cas où l'utilisateur donne la description d'une instance qui existe déjà en MLT. Le cas général : l'instance est à intégrer pour la première fois, implique la création du lien "-inst->" entre le nœud en MLT qui définit le type de l'instance et le nouveau nœud. Si la description de l'instance est fournie [Type :InstId =Descr], alors la liste des indexes LIndex est construite à partir des concepts qui se trouvent dans Descr. La procédure Integrate2 complète ensuite l'intégration de la description.

Pour la description d'une définition ou d'un schéma, la procédure Elicitate est appelée, impliquant une mise à jour éventuelle de la structure à intégrer ainsi que la détermination de la liste des focus (suite à l'interaction avec l'utilisateur). La liste des indexes LIndex est déterminée en fonction de la liste des focus (les concepts qui ne sont pas en focus sont considérés comme des indexes). La nouvelle structure est ensuite intégrée, avec Integrate2.

Note: les entrées possibles pour Integrate1 sont :

Déclaration de synonymie : <Type, Synonym>. Définition d'un type : [Type :Referent =Value].

Description d'instance : [Type :Referent] or [Type :Referent =Value].

Description de schéma : [:Referent =Value].

```
procedure Integrate1(in Input: ConcOr2Types; LFocus: ListOf Concept) is
   if Input corresponds to synonymy declaration : <Type, Synonym> then
      AddSynonym(Synonym, Type)
   elseIf IsInstance(Input/NewConc) then
                        /** Input corresponds to a concept NewConc **/
      if there is already a node N in LTM for NewConc then
         {Exit with ErrMsg if N.Value <> nil ;
          N.Value := Value }
      else
         {Add NewConc to LTM ;
          Add to LTM the branch: ConceptDef(NewConc.Type) -inst-> NewConc ;
          if NewConc.Value <> nil then
               LIndex := All the concepts of NewConc.Value except
                                 the concept with referent = NewConc.Referent
          else LIndex := nil
          endIf ;
          Integrate2(NewConc, LFocus, Lindex) /** LFocus is nil in this case **/}
      endIf
   else
      {Elicitate(NewConc) ;
            /** LFocus could be nil, some or all the concepts of NewConc.Value ***/
       Add NewConc to LTM ;
       LIndex := All the concepts of NewConc.Value except those that exists in LFocus ;
            /** LIndex could be nil, some or all the concepts of NewConc.Value ***/
       Integrate2(NewConc, LFocus, LIndex) }
   endIf
end Integrate1.
```

AddSynonym(Synonym, Type) : ajoute Synonym dans la liste des synonymes de Type, si Synonym n'y est pas déjà.

procedure Elicitate(**in** NewConc: Concept) :

La procédure effectue un traitement itératif à la recherche de définitions de types dans la structure en entrée.

→ Un cas particulier est celui où toute la structure en entrée correspond à la définition d'un type. Si la structure est présentée (à Elicitate) comme un schéma, l'utilisateur est informé que sa donnée correspond en fait à une définition de type, si elle est présentée comme une définition alors ou bien, cette même définition existe déjà dans la MLT, ou bien la définition existe déjà mais pour un autre type (le système identifie un cas de synonymie).

→ Le cas général est que la structure en entrée puisse contenir la définition d'un type T, le type correspondant dans la structure est remplacé par le type T et l'information redondante (information qui existe dans la structure et dans la définition) est colorée.

→ L'itération de la procédure Elicitate permet d'identifier les définitions qui sont basées sur la reconnaissance préalable d'autres définitions. Par exemple, la définition de PelicanScientist est basée sur la définition de Pelican (PelicanScientist est un scientifique qui étudie les Pelican) et la nouvelle structure à intégrer pourrait contenir l'information ".. un scientifique qui étudie les oiseaux avec un grand bec ..", la présence dans cette information de la définition de PelicanScientist ne peut être identifiée qu'après celle de Pelican.

```
procedure Elicitate(in NewConc: Concept) is
    Stop := false ;
    while Not Stop And nil <> (CChild := ContainDefinition(NewConc.Value, C)) do :
          if EqCG(CFils.Value, NewConc.Value) then
             {Stop := true ;
              if IsASchema(NewConc) then
                    message("The schema corresponds in fact to a known definition.")
              elseIf IsSynonym(NewConc.Type, CFils.Type) then
                          message("This definition exists already in memory.")
              else
                 {AddSynonym(NewConc.Type, CFils.Type) ;
                  message("Definition of ", NewConc.Type, "corresponds to the définition of ",
                                    CFils.Type, "and of its synonyms".) ;
                  message("I infer that", NewConc.Type, " is a synonym of ", CFils.Type) }
              endIf ;
              Exit from the integration process }
          else
              {message("The input contains the definition of ", CFils.Type) ;
               C.Type := CFils.Type ;
               Colour in NewConc.Value any information that is contained in the definition ;
          endIf ;
    endWhile ;
    message("Elicitation finished, you can update the input.");
end Elicitate.
```

IsSynonym(TypeS, Type) : elle vérifie si TypeS = Type ou si TypeS est un synonyme de Type.

function ContainDefinition(**in** G: CG; **out** C, ConcFilsP: Concept) : Boolean

Cette fonction regarde si la structure en entrée G contient une définition de type : elle cherche un concept C dans G et elle cherche de la MLT les définitions spécifiées par rapport au type de C, elle vérifie ensuite si une des définitions est contenue dans G. La recherche des définitions de la MLT est effectuée par la procédure DescendantsDef.

```
function ContainDefinition(in G: CG; out C: Concept): Concept is
    Found := false ;
    for any concept C of G And Not Found do :
       DescendantsDef(ConceptDef(C.Type), ConcFilsL) ;
       while Not (ConcFilsL.IsEmpty Or Found) do
          {RemoveHead(ConcFilsL, ConcFils) ;
           E1 := Localise(ConcFils.Value, "super") ;
           matchCG(cplteContract, E1, ConcFils.Value, C, G, nil, Found) }
       endWhile ;
       Free(ConcFilsL) ;
    endFor ;
    if Found then return ConcFils else return nil endIf ;
end ContainDefinition.
```

procedure DescendantsDef(**in** NodeMem: Concept; **inout** ConcFilsL:ListOf Concept) :

La procédure collecte les définitions qui ont le type du concept NodeMem comme super-type. Dans sa recherche des définitions, la procédure tient compte des relations entre les définitions et les schémas. Par exemple, on pourrait avoir le chemin suivant :

DefTyp1 -scm-> Schema -sp-> Schema -sp-> DefTyp2. DefTyp2 doit être considérée également comme une définition dérivée de DefTyp1.

```
procedure DescendantsDef(in NodeMem: Concept; inout ConcFilsL:ListOf Concept) is
    for any relation NodeMem -Rel-> Conc   with   Rel = sp Or scm,  do :
        if Rel = sp And IsADefinition(Conc)  then
            if Not  member(Conc, ConcFilsL)  then  Insert(Conc, ConcFilsL)  endIf
        else DescendantsDef(Conc, ConcFilsL)
        endIf
    endFor ;
end DescendantsDef.
```

procedure Integrate2(**in** NewConc: Concept ; LFocus, LIndex : ListOf Concept) :

Pour chaque élément F (qui est un concept) de la liste des focus LFocus, la procédure Integrate2 localise dans la MLT un nœud définition N pour le type de F et propage la description représentée par NewConc à chaque fils connecté à N par le lien "sp" ou "scm". Si l'élément F est une référence à une instance (par exemple [Homme :Marc.]) alors Integrate2 propage NewConc à chaque fils du nœud qui décrit l'instance. Si NewConc est incomparable à tous les fils du nœud N alors il est ajouté comme un nouveau fils du nœud N.

Ensuite, la procédure Integrate2 considère chaque élément de la liste des indexes LIndex, localise son nœud définition N dans la MLT et y ajoute ensuite la relation N -scm-> NewConc.

```
procedure Integrate2(in NewConc: Concept ;  LFocus, LIndex : ListOf Concept) is
    for each element F of LFocus do :
        Desc := false ;
        for each concept C : ConceptDef(F.Type)  -sp/scm-> C  do
            Desc := Desc Or  Propagate(NewConc, F, C, ConceptDef(F.Type)) ;
        endFor ;
        if  reference(F.Ref) And there is  ConceptDef(F.Type) -inst-> ConceptDef(F.Ref)  then
            for each concept C : ConceptDef(F.Ref)  -scm-> C  do
                Desc := Desc Or Propagate(NewConc, F, C, ConceptDef(F.Ref)) ;
            endFor ;
        endIf ;
        if not Desc then
            if F.Ref = "super" then  addto LTM the relation ConceptDef(F.Type) -sp-> NewConc
            else
              { addto LTM the relation ConceptDef(F.Type) -scm-> NewConc ;
                If  reference(F.Ref) And there is  ConceptDef(F.Type) -inst-> ConceptDef(F.Ref)  then
                        addto LTM the relation  ConceptDef(F.Ref) -scm-> NewConc
                endIf }
            endIf
        endIf ;
    endFor ;
    for each element I of Index  do :
        if there is no relation in LTM : ConceptDef(I.Type) -scm-> NewConc  then
            addto  LTM the relation  ConceptDef(I.Type) -scm-> NewConc
        endIf ;
```

```
        if reference(I.Ref) And there is  ConceptDef(I.Type) -inst-> ConceptDef(I.Ref) then
            addto  LTM the relation  ConceptDef(I.Ref) -scm-> NewConc ;
        endIf ;
    endFor
end Integrate2.
```

function Propagate(**in** NewConc, Focus, CurrentConc, FatherConc: Concept) : Boolean

♦ Si NewConc a été déjà comparé avec CurrentConc, alors aucun traitement n'est effectué et le résultat de la comparaison antérieure est retourné ici. Ce cas peut survenir car la nouvelle information NewConc peut atteindre le même nœud par différents chemins.

Puisque l'élicitation est effectuée avant l'intégration, la description de NewConc ne sera pas modifiée au cours de l'intégration. Ainsi, NewConc peut atteindre le même nœud par différents chemins mais toujours *avec la même description*, une seule comparaison entre NewConc et CurrentConc est donc suffisante. Dans notre formulation antérieure du modèle [Kabbaj, 95], l'élicitation s'effectue durant la propagation et donc, NewConc pourrait changer de description, impliquant une comparaison entre NewConc et CurrentConc chaque fois que le premier atteint le second.

♦ Si NewConc atteint CurrentConc pour la première fois, alors les deux sont comparés et le résultat est enregistré dans ResComparison. NewConc est ensuite intégré à partir de CurrentConc, l'intégration va dépendre du résultat de la comparaison :

→ si NewConc et CurrentConc sont incomparables, la propagation n'aura pas lieu.

→ si NewConc is-more-general-than CurrentConc, alors la branche FatherConc -R-> CurrentConc est remplacée par FatherConc -Rel1-> NewConc -Rel2-> CurrentConc. Les types de Rel1 et Rel2 dépendent des types des nœuds : si FatherConc est un schéma ou si NewConc est une définition, alors Rel1 serait "sp", sinon elle serait "scm". Si NewConc est un schéma ou si CurrentConc est une définition, alors Rel2 serait "sp", sinon elle serait "scm".

La procédure Propagate considère ensuite le cas où NewConc est une définition. Puisque NewConc is-more-general-than CurrentConc, alors CurrentConc contient la description de NewConc et il faudrait donc contracter cette dernière de CurrentConc afin d'avoir une formulation plus concise et aussi afin de refléter le fait qu'à présent NewConc est le père de CurrentConc.

→ Si CurrentConc est lui-même une définition alors l'opération de contraction ci-dessus ne sera pas "propagée" aux fils de CurrentConc car ces derniers sont indexés par le type et non par le contenu de la définition (par exemple, PelicanScientist est indexé par Pelican et même si la définition de ce dernier est reformulée, la définition de PelicanScientist ne sera pas affectée "directement" ; elle contient toujours le concept/terme Pelican et c'est la "définition" de ce dernier qui a changé).

→ Si CurrentConc est un schéma alors l'opération de contraction sera propagée aux fils de

CurrentConc car dans ce cas les fils sont indexés par le contenu du nœud père. Par exemple, si nous avons cette configuration : ... NewConc -Rel2-> CurrentConc -sp-> Schema1 , la description de CurrentConc est contenue dans Schema1 et puisque la procédure vient de constater que la description de NewConc est contenue dans celle de CurrentConc alors elle est contenue aussi dans Schema1, il faudrait donc contracter aussi NewConc de Schema1. Cette "propagation" de la contraction de NewConc continue, de façon récursive aux fils des fils, si un fils est une définition alors il n'y aura pas de propagation via ce fils.

→ si NewConc is-equal-to CurrentConc alors les deux nœuds sont fusionnés comme suit : CurrentConc est éliminé et ses relations sont maintenant connectées à NewConc. Des cas particuliers sont considérés lors de la fusion. Par exemple, si l'un est un schéma et l'autre une définition alors un "réajustement conceptuel" des descendants du nœud schéma est nécessaire : la description de NewConc doit être contractée des descendants puisque le nœud père est envisagé à présent comme une définition.

→ si NewConc is-more-specific-than CurrentConc, alors NewConc est propagé aux fils de CurrentConc.

→ si NewConc has-some-elements-in-common-with CurrentConc, alors un nœud schema N est créé avec comme contenu le GC qui résulte de la généralisation de NewConc avec CurrentConc et la relation FatherConc -R-> CurrentConc est remplacée par

FatherConc -Rel-> N --sp--> CurrentConc
└---sp--> NewConc.

où Rel serait "sp" si FatherConc est un schéma, sinon elle correspondrait à "scm"

```
function Propagate(in NewConc, Focus, CurrentConc, FatherConc: Concept) : Boolean is
 BRes := true ;
 if there is <NewConc, CurrentConc, Res> in ResComparison then BRes := (Res = "comparable")
       /*** ResComparison = ListOf (Concept, Concept, String) ***/
 else
   {Compare(NewConc, Focus, CurrentConc, FatherConc, ResComp, Oper) ;
    if Oper = "inc" then Res := Oper else Res := "comparable" endIf ;
    Insert(<NewConc, CurrentConc, Res>, ResComparison) ;
    case Oper
      = "inc" : BRes := false ;
      = ">" :
         {if IsASchema(FatherConc) Or IsADefinition(NewConc) then Rel1:= "sp" else Rel1 := "scm";
          if IsASchema(NewConc) Or IsADefinition(CurrentConc) then Rel2:= "sp" else Rel2:= "scm";
```

```
    Eliminate from LTM the relation FatherConc -Rx-> CurrentConc ;
    addTo LTM the relation FatherConc -Rel1-> NewConc ;
    addTo LTM the relation NewConc -Rel2-> CurrentConc ;
    if IsADefinition(NewConc) And IsASchema(CurrentConc) then
        /* the case where CurrentConc is a definition is considered implicitely by compare procedure */
        Compressions(NewConc, Children(CurrentConc))
    endIf }
= "=" :
   {if IsASchema(CurrentConc) And IsADefinition(NewConc) then
        {WriteMessage("Definition body is already known, but as a schema.") ;
         Father := CurrentConc ;
         ChildL := Children(Father) }
    elseIf IsASchema(NewConc) And IsADefinition(CurrentConc) then
        {NewConc.Type := CurrentConc.Type ;
         WriteMessage("The schema corresponds in fact to the definition of :", CurrentConc.type) ;
         NewConc.Ref := "_X" ;
         C1 := LocalizeConc(CurrentConc.Value, "super") ;
         C2 := is the concept of NewConc.Value that matched C1 ;
         C2.Ref := "super" ;
         Father := NewConc ;
        ChildL := Children(Father) }
    else /*** NewConc and CurrentConc are schemas ***/
        {WriteMessage("The schema exists already in the LTM.") ;
         ChildL := nil }
    endIf ;
    for each relation R that links CurrentConc, no matter the direction, to a concept Cx do :
      if there is another relation R1 with the same name as R, And
        R1 is linked to NewConc and to Cx in the same direction as R
      then Eliminate the relation R
      else Replace in R CurrentConc by NewConc ;
    endFor ;
    Eliminate from LTM the concept CurrentConc ;
    if Not ChildL.IsEmpty then
      { for each element Child of ChildL do :
          if IsASchema(Child) then
              Replace "sp" par "scm" for the relation between Father and Child ;
          endIf ;
        endFor ;
```

```
                Compressions(NewConc, ChildL) }
            endIf }
        = "<" :
            {Desc := false ;
             for each concept C : CurrentConc -sp-> C do
                 Desc := Desc Or Propagate(NewConc, Focus, C, CurrentConc) ;
             endFor ;
             if not Desc then addto LTM the branch CurrentConc -sp-> NewConc endIf }
        = "int" :
            { addTo LTM the Concept Cn := [=ResComp] ; /** it represents a schema ***/
              Eliminate from LTM the relation FatherConc -Rx-> CurrentConc ;
              if IsASchema(FatherConc) then Rel := "sp" else Rel := "scm" endIf ;
              addTo LTM the relation FatherConc -Rel-> Cn ;
              addTo LTM the relation Cn -sp-> CurrentConc ;
              addTo LTM the relation Cn -sp-> NewConc }
    endCase }
  endIf ;
  return BRes ;
end Propagate.

function Children(in Conc: Concept) : ListOf Concept ;
    return all concepts C : Conc -sp/scm-> C ;
                        /*** any concept C related to Conc by "sp" or "scm" link **/
end Children.

procedure Compressions(in NewConc: Concept; Children: ListOf Concept) is
    for each element Child of Children do
        { ContractionSp(NewConc, Child, false) ;
          if IsASchema(Child) then Compressions(NewConc, Children(Child)) endIf }
    endFor ;
end Compressions.
```

procedure ContractionSp(**in** NewConc: Concept, **inout** CurrentConc: Concept ; **in** IsCompared: Boolean)

Cette opération contracte la description de NewConc de celle de CurrentConc. Certains concepts de CurrentConc peuvent être éliminés suite à la contraction et donc, CurrentConc ne devrait plus être indéxé sous ces concepts. Pour effectuer ce traitement, l'opération de contraction enregistre dans la liste globale LTypeConcElim chaque concept éliminé durant la contraction.

```
procedure ContractionSp(in NewConc: Concept, inout CurrentConc:Concept;in IsCompared: Boolean) is
    Contract2(NewConc, CurrentConc.Value, IsCompared) ;
    for each element Indice in LTypeConcElim do :
        if there is a relation ConceptDef(Indice) -R-> CurrentConc in LTM
        then Eliminate the relation endIf ;
    endFor ;
end ContractionSp.
```

procedure Contract2(**in** C: Concept, **inout** G: CG, **in** IsCompared: Boolean)

Cette procédure tente de contracter la description de C de celle de G. Pour cela, elle identifie les points d'entrée pour les deux graphes et compare ensuite ces derniers (si ce n'est déjà fait). Si le premier graphe est contenu dans le second, la procédure remplace le super-type de G par le type défini, met le super-type dans la liste LTypeConcElim afin d'indiquer qu'il n'indexe plus la structure G et active ensuite la contraction.

```
procedure  Contract2(in C: Concept, inout G: CG, in IsCompared: Boolean)  is
   E1 := LocalizeConc(C.Value, "super") ;
   if non IsCompared  then
        {Put in LConc all the concepts Ej of G with : Ej.Type = E1.Type ;
         BRes := false ;
         while Not (BRes  Or  LConc.IsEmpty) do :
              matchCG(cplteContract, E1, C.Value, RemoveHead(LConc), G, nil, BRes) ;
         endWhile ;
         Free(LConc) }
   endIf ;
   if  IsCompared Or BRes  then
        { Search in CMatchL the concept E2 of G that matched with E1 : <E1, E2, _, _, _> ;
          Insert(E2.Type, LTypeConcElim) ;
          Replace the type of E2 by C.Type ;
          ContractG(G) }
   endIf ;
end Contract2.
```

procedure Compare(**in** NewConc, Focus, CurrentConc, FatherConc: Concept,

out ResComp: CG, Oper: String)

La procédure compare la description de NewConc avec celle de CurrentConc, avec Focus comme point d'entrée pour le premier graphe; pour le second graphe, la procédure cherche tous les concepts Ej qui ont le même type que celui de Focus, elle cherche ensuite la "meilleure" généralisation entre les deux graphes. En effet, la procédure effectue la généralisation des deux graphes en considérant les différents Ej, la meilleure généralisation est celle qui produit la plus "grande" généralisation et le graphe avec le plus grand nombre de branches.

Le résultat de la généralisation est ensuite utilisé pour effectuer la comparaison entre les deux graphes :

→ les deux graphes sont incomparables si le résultat de la généralisation est égal à la description du père (ils n'ont rien en commun excepté la description du même père).

→ les deux graphes sont identiques (égaux) s'ils ont le même nombre de concepts, si les concepts en correspondance sont égaux, s'ils ont le même nombre de relations et si le graphe résultat a le même nombre de relations que les autres.

→ le premier graphe est plus général que le second s'il a le même nombre de concepts que le graphe résultat de la généralisation, si les concepts en correspondance sont égaux et s'ils ont le même nombre de relations. Si le premier graphe (la description de NewConc) est une définition, il est alors contracté du second graphe (la description de CurrentConc).

→ le premier graphe est plus spécifique que le second si ce dernier et le graphe résultat ont le même nombre de concepts, si les concepts en correspondance sont égaux et s'ils ont le même nombre de relations.

→ si les cas précédents ne s'appliquent pas, alors conclure que les deux graphes ont uniquement certains éléments en commun, éléments qui correspondent au résultat de la généralisation.

```
procedure Compare(in NewConc, Focus, CurrentConc, FatherConc: Concept,
                  out ResComp: CG, Oper: String) is
   Put in LConc all the concepts Ej of CurrentConc.Value with : Ej.Type = Focus.Type ;
   BRes := false ;
   BestEntry := nil ;
   MaxSize := 0 ;
   while Not LConc.IsEmpty do :
      E2 := RemoveHead(LConc) ;
      matchCG(generalize, Focus, NewConc.Value, E2, CurrentConc.Value, ResComp, BRes) ;
      if Length(LRMatch) > MaxSize then
         {MaxSize := Length(LRMatch) ;
          BestEntry := E2 }
      endIf ;
   endWhile ;
   matchCG(generalize, Focus, NewConc.Value, BestEntry, CurrentConc.Value, ResComp, BRes) ;
   if EqCG(ResComp, FatherConc.Value) Or
      ResComp is a graph of one concept with Type = Focus.Type then Oper := "inc"
   elseIf NbreConcs(NewConc.Value) = NbreConcs(CurrentConc.Value) And EqMatchConc(1, 2) And
       ( NbreRels(NewConc.Value) = NbreRels(CurrentConc.Value) = NbreRels(ResComp) )
       then Oper := "="
   elseIf NbreConcs(NewConc.Value) = NbreConcs(ResComp) And EqMatchConc(1, 3)
          And NbreRels(NewConc.Value) = NbreRels(ResComp) then
      { Oper := ">" ;
       if IsADefinition(NewConc) then ContractionSp(NewConc, CurrentConc, true) endIf }
   elseIf NbreConcs(CurrentConc.Value) = NbreConcs(ResComp) And EqMatchConc(2, 3)
          And NbreRels(CurrentConc.Value) = NbreRels(ResComp) then Oper := "<"
   else Oper := "int"
   endIf ;
end Compare.
```

```
function EqMatchConc(in Rg1, Rg2: integer): Boolean is
    EqualConc := true ;
    for each element E of LCMatch  And  EqualConc  do :
        RangeConc(Rg1, E, Conc1) ;
        RangeConc(Rg2, E, Conc2) ;
        EqualConc := (Conc1.Type = Conc2.Type  And  EqRef(Conc1.Referent, Conc2.Referent)  And
                      EqValue(Conc1.Value, Conc2.Value) );
    endFor ;
    return EqualConc ;
end EqMatchConc.

procedure RangeConc(in Rg: integer, E: ElemOfCMatchL; out Conc) is
    case Rg
         = 1 : return E.ConcMatched1 ;
         = 2 : return E.ConcMatched2 ;
         = 3 : return E.ResOfMatch ;
    endCase  ;
end RangeConc.

function EqRef(in Ref1, Ref2 : Referent) : Boolean is
    return Ref1 = Ref2  Or  (Variable(Ref1)  And  Variable(Ref2)) ;
end EqRef.

function EqValue(in Val1, Val2: Value) : Boolean is
    if cg(Val1) And cg(Val2) then return EqCG(Val1, Val2)  else return Val1 = Val2 ;
end EqValue.

function EqCG(in G1, G2): Boolean is
    return  NbreConcs(G1) = NbreConcs(G2)  And  NbreRels(G1) = NbreRels(G2)  And
          matchCG(cplteContract, nil, G1, nil, G2, nil, BRes) And BRes = true And  EqMatchConc(1,2);
end EqCG.
```

6.4 Environnement graphique pour le modèle de formation de la MLT

L'environnement graphique qui supporte Synergy supporte également toutes les étapes du processus d'intégration : il fournit des facilités graphiques pour la saisie de l'information, pour visionner le résultat de la procédure d'explicitation pendant que celle-ci met à jour la nouvelle information (en changant les types de concepts et en colorant des parties de l'information), pour permettre à l'utilisateur de spécifier les éléments en focus, pour visionner et suivre le processus d'intégration et enfin pour explorer le résultat de l'intégration. En effet, seuls les concepts et les relations de la MLT qui sont considérés (visiter) durant l'intégration seront visibles. L'environnement graphique offre en plus l'option "Voisinage" afin de rendre visible le voisinage d'un concept (les concepts adjacents au concept). Ainsi et selon l'intérêt de l'utilisateur, ce dernier peut explorer plus en profondeur des parties de la MLT qui sont reliées au résultat du processus d'intégration, il peut ainsi "agrandir", selon ses intérêts, la vue partielle de la MLT produite par le processus d'intégration.

Note: Pour maximiser la "visibilité" de la mémoire, les liens "sm" créés par le mécanisme de connexion sont invisibles; ils sont rendus visibles s'ils sont utilisés.

6.5 Utilisation du modèle de la mémoire et de son environnement

La formation et l'utilisation de la mémoire peuvent être considérées séparement ou conjointement. Le modèle que nous avons présenté est formulé dans le cadre de la première approche :

→ Le processus de formation de la mémoire dynamique, décrit dans ce chapitre, montre comment une nouvelle information est intégrée en mémoire. Cette dernière peut être ensuite utilisée par différents processus "cognitifs" comme le traitement du langage naturel, le raisonnement et la planification. Une utilisation de base serait la consultation de la mémoire; demander si cette dernière contient une information donnée. Au lieu de répondre par un "oui/non", le processus de recherche tenterait de localiser le "voisinage" relatif à l'information recherchée ; cette dernière peut se trouver intégralement ou partiellement dans la mémoire, dans les deux cas il serait approprié d'identifier toute information similaire et "proche" de l'information recherchée. Une réalisation possible de ce processus de recherche serait de "classer" la demande comme une nouvelle information : intégrer l'information par propagation dans la mémoire et à sa comparaison avec une information InfDjExt déjà existante, considérer uniquement les cas où la nouvelle information est plus générale, plus spécifique ou égale à InfDjExt. En général, la nouvelle information ne doit pas être ajoutée à la mémoire ni impliquer un changement (car il s'agit de savoir si elle existe ou non dans la mémoire et non de l'y intégrer). Mais, une question peut véhiculer une nouvelle information qui serait intégrée en mémoire comme "effet de bord" [Dyer, 83], comme par exemple : "le chapeau que portait Michel à la soirée d'hier, était-il bleu ou noir ? ".

Une autre utilisation est présentée dans l'annexe E, illustrant l'utilisation de la mémoire dans la "reconstruction" d'un concept.

→ Parmi les développements futurs du modèle, nous envisagons de considérer conjointement la formation et l'utilisation de la mémoire. Dans [Kabbaj et Frasson, 93] nous présentons quelques "réflexions" qui indiquent comment des inférences peuvent être effectuées *durant* l'intégration de la nouvelle information en mémoire. Il faudrait approfondir ces "réflexions" et considérer les travaux effectués dans cette direction, en particulier ceux de Michalski et Tecuci sur l'apprentissage multi-stratégies ([Michalski, 93, 94], [Tecuci, 94, 95]).

Classification d'une instance

L'intégration des instances en mémoire se réduit au mécanisme de connexion, l'instance est relié au type de l'instance ainsi qu'aux autres concepts qui composent la description. Le traitement est effectué ainsi car la catégorie (ou le type) de l'instance est fournie. Le problème

de classer une instance pour identifier son(ses) types peut être traité dans notre modèle selon une approche "sémantique" et non "probabiliste". La première approche consiste à chercher les types les plus spécifiques dont la définition est plus générale que la description de l'instance. Dans ce cas, les types (et leurs définitions) sont prioritaires aux instances; les premiers servent à classer les seconds. Dans l'approche "probabiliste" c'est l'inverse : à partir d'un ensemble d'instances, il faut produire une hiérarchie de classes qui peut guider par la suite la classification d'une nouvelle instance. Plusieurs algorithmes ont été développés pour la formation dynamique d'une hiérarchie de concepts selon un modèle probabiliste ([Wasserman, 85], [Fisher, 87], [Gennari et al., 89], [Bareiss, 89], [Bergadano et al., 91], [Fisher et al., 91]). Dans certains domaines d'application, le modèle probabiliste constitue une hypothèse appropriée pour calculer la pertinence de chaque valeur particulière d'un attribut, calculer la valeur moyenne pour l'attribut, etc. L'approche probabiliste est basée sur une "définition par prototype" d'un concept [Franconi et al., 92], elle montre comment les prototypes peuvent se former et comment on peut juger qu'un objet est une instance de tel ou tel concept. Dans un modèle général de la mémoire dynamique, les deux approches "sémantique" et "probabiliste" doivent co-exister. Leur intégration dans notre modèle fait partie des travaux futurs.

6.6 Résumé

Nous avons présenté dans ce chapitre un modèle de formation dynamique et incrémentale de la mémoire spécifiant comment une nouvelle information est intégrée en mémoire.
En réponse à un flot continu de nouvelles informations, le modèle développe et réorganise la mémoire à long terme (MLT).
La "mémoire" correspond à la MLT de Synergy et une nouvelle "information" à intégrer correspond à une définition, la description d'une instance ou d'un schéma, ou à une déclaration de synonymie de type. Le processus d'intégration reçoit en entrée une nouvelle information, y applique une procédure d'explicitation, demande ensuite à la source de l'information les éléments en-focus dans la nouvelle information et amorçe l'intégration de l'information en mémoire. Les éléments en-focus "guident" le processus d'intégration : ce dernier localise dans la mémoire, "les nœuds" associés aux éléments en-focus et la nouvelle information est propagée dans la mémoire via ces nœuds. Enfin, au cours de l'intégration, les similitudes et les différences entre la nouvelle information (ou structure) et celles qui existent déjà en mémoire sont établies.
Le modèle constitue une alternative à la gestion explicite de la MLT par le programmeur Synergy et peut également constituer (à long terme) la "primitive" de base dans une *programmation orientée mémoire* où les "programmes" correspondent aux "processus cognitifs".

Chapitre 7

Prolog+CG : une extension contextuelle et orientée objet de Prolog, utilisant les GC

Nous décrivons dans ce chapitre le langage Prolog+CG, une extension conceptuelle, contextuelle et orientée objet du langage Prolog. L'extension conceptuelle se base sur les GC alors que l'extension orientée objet se base sur les deux premières extensions ainsi que sur une formulation logique de la programmation orientée objet. Prolog+CG offre de plus une algèbre pour les GC et il étend Prolog sans le masquer ; un programme Prolog est aussi un programme Prolog+CG.

Après un aperçu sur le langage, nous présentons la notion d'objet en Prolog+CG. Puis nous présentons respectivement la syntaxe du langage, des exemples, l'héritage entre objets et la possibilité d'instanciation, et enfin des détails sur l'environnement et le "compilateur" de Prolog+CG.

7.1 Aperçu sur le langage Prolog+CG

Dans notre proposition d'une "extension Prolog pour les GC", nous considérons les extensions suivantes, apportées au langage Prolog par la communauté "programmation en logique" :

→ La *programmation logique conceptuelle* est caractérisée par l'utilisation de structures plus "complexes et conceptuelles" que les prédicats/termes utilisés en Prolog. La programmation logique conceptuelle est illustrée par des extensions de Prolog comme Login [Aït-Kaci et Nasr, 86], Life [Aït-Kaci et al., 92], L-Lilog ([Pletat, 91] et [Herzog et Rollinger, 91]) et Netlog [Voinov, 92] où des "structures typées avec attributs" [Carpenter, 92] sont utilisées.

→ La *programmation logique contextuelle* est caractérisée par l'organisation modulaire d'un programme logique, considéré alors comme une collection de contextes (ou modules, objets, théories), chacun est composé d'un ensemble de règles. La programmation

logique contextuelle est illustrée par des extensions de Prolog comme Multilog [Kauffmann et Grumbach, 86], Clp [Monteiro et Porto, 89] et W-Prolog [Dichev, 93].

→ La *programmation logique orientée objet* est une extension de la programmation logique contextuelle où la notion de contexte est utilisée pour représenter des classes et des instances, avec possibilité d'héritage. Les méthodes d'une classe sont formulées comme des règles et l'envoi d'un message correspond à la satisfaction d'un but. La programmation logique orientée objet est illustrée par des langages comme L&O [McCabe, 92] et Prolog++ [Moss, 94].

Ces différentes extensions de Prolog sont intégrées à Prolog+CG comme suit : un but en Prolog+CG peut être représenté par un terme Prolog ou un GC, diverses opérations sur les GC sont offertes comme des opérations primitives, un programme en Prolog+CG est organisé en une collection de contextes, chacun est composé d'une suite de règles. Les contextes peuvent former en plus une hiérarchie avec possibilité d'héritage et un contexte peut être utilisé pour représenter une classe et pour spécifier une instance d'une classe : certaines règles d'un(e) contexte/classe pourraient correspondre à la partie déclarative d'une classe alors que d'autres aux méthodes de la classe, la satisfaction d'un but concernant le contexte correspondrait alors à un envoi de message.

Prolog+CG considère une autre forme de contexte, en particulier la possibilité d'avoir des GC imbriqués, permettant ainsi la formulation de règles d'inférence et de manipulation de contextes et donc, une certaine *programmation modale et d'ordre supérieur.*

Par ailleurs, la compilation de plusieurs fichiers à la C-Prolog, fournit un autre moyen de programmation modulaire : une "grande" application en Prolog+CG peut être constituée de plusieurs fichiers Prolog et Prolog+CG.

Enfin, notons que Prolog+CG est "ouvert" à Prolog : un fichier Prolog+CG peut contenir des règles Prolog, une règle Prolog+CG peut contenir non seulement un envoi de message mais aussi un but qui correspond à une primitive ou qui est défini par des règles Prolog. Enfin, une règle Prolog dans un fichier Prolog+CG peut contenir un envoi de message.

Le manuel du langage Prolog+CG [Kabbaj, 95] est disponible par ftp et une mise à jour est en cours. Une première implantation d'un sous-ensemble du langage a été réalisée en Janvier 1994 ([Kabbaj et al., 94], [Kabbaj et Frasson, 94]). Une implantation plus complète et plus efficace est en cours.

Note: Tous les exemples dans ce chapitre sont extraits, sans traduction, du manuel de Prolog+CG (rédigé en anglais).

7.2 Elements de base du langage Prolog+CG

Note: Dans le reste du présent chapitre, nous utilisons le terme "contexte" pour désigner des GC imbriqués et le terme "objet" pour désigner un contexte composé d'une suite de règles.

Cette section présente la notion d'objet en Prolog+CG, spécifie brièvement les objets primitifs et indique quelques utilisations possibles de la notion d'objet et des variables en Prolog+CG. La syntaxe du langage est présentée dans la section 7.3.

7.2.1 Objets en Prolog+CG

Définition 1 : *Programme Prolog+CG*
Un programme Prolog+CG est composé de règles Prolog et/ou d'objets. ♦

Définition 2 : *Objet Prolog+CG*
Un *objet* est formé d'un *descripteur* et d'une collection de *règles*, le descripteur est représenté par un *terme* et il préfixe chacune des règles de l'objet. ♦

La forme générale d'un objet est comme suit :

```
Descr' :: Regle1.
Descr'':: Regle2.
...
```

Note: nous avons ajouté des primes à Descr pour signaler qu'ils ont le même nom de prédicat et le même nombre de paramètres, les paramètres par contre peuvent être différents.

Définition 3 : *Terme Prolog+CG*
Un terme Prolog+CG peut être une constante, une variable, un GC, une liste de termes, ou un prédicat dont les arguments sont des termes. Une constante est soit un atome (un identificateur qui commence par une minuscule) ou un nombre. ♦

Remarque : Le fait que le descripteur puisse être un terme et donc un prédicat avec des paramètres confère une grande puissance d'expression au langage. Il en est de même pour le descripteur qui préfixe chaque règle de l'objet ; les paramètres du descripteur peuvent différer d'une règle à une autre.

Définition 4 : *Règle dans un objet*

Une règle dans un objet est composée d'une tête (représentée par un prédicat ou un GC) et d'une queue optionnelle formée d'une conjonction de termes Prolog et/ou d'*envois de messages*. Une règle peut être commune à (partagée par) plusieurs objets et dans ce cas, elle serait préfixée par une conjonction de descripteurs d'objets. ♦

La forme d'une règle d'un objet est soit " ObjDescr::Tete Queue. " si la règle est propre à un objet, ou " [ObjDescr1, ... , ObjDescrN]::Tete Queue. " si elle est commune à plusieurs objets. Tete est représentée par un prédicat ou un GC et la Queue peut être vide ou être de la forme " :- But1, ... , ButN. " avec Buti qui peut être un terme ou un envoi de message.

Définition 5 : *Envoi de message à un(des) objet(s)*

L'envoi d'un Message à un objet avec un descripteur ObjDescr, noté ObjDescr::Message, est un but à satisfaire selon le contenu de l'objet, Message peut être représenté par un prédicat ou un GC et il peut être envoyé à un seul objet : ObjDescr::Message, à un objet parmi plusieurs : [ObjDescr1; ... ; ObjDescrN]::Message ou à plusieurs objets : [ObjDescr1, ... , ObjDescrN]::Message. ♦

Définition 6 : *Interprétation d'un envoi de message*

L'interprétation d'un envoi de message ObjDescr::Message consiste à chercher une règle avec un descripteur qui puisse s'unifier avec ObjDescr. Si l'unification réussit, le message Message est unifié avec la tête Tete de la règle (selon que Message et Tete soient des prédicats ou des GC) et si cette seconde unification réussit, l'interpréteur tente alors de satisfaire la queue de la règle. ♦

Note : Cette définition sera étendue par la suite afin de tenir compte de la possibilité d'héritage entre les objets.

Exemple : *Objet non paramétré*

L'objet hamza (dans la réalité, ce n'est pas un "objet" mais un beau garçon) possède deux règles, la première est un fait (de la forme ObjDescr::GC.) qui décrit les attributs de l'objet et la seconde définie une "méthode", age(Age), associée à l'objet.

```
hamza::{[person]-
            chrc->[age]-ptim->[date : (6, december, 1995) ] ,
            chrc->[birth]-ptim->[date : (5, april, 1995) ] }.
hamza::age(Age) :-
        hamza::{[person]-chrc->[age]-ptim->[date : D1]} ,
        hamza::{[person]-chrc->[birth]-ptim->[date : D2]} ,
        dtDiff(D1, D2, Age) .
```

On peut ensuite demander à hamza son age :

```
I: hamza::age(Age).
Age = (1, 8, 0).
yes
```

Ou lui demander s'il a une voiture :

```
I: hamza::{[person]-has->[car]}.
no
```

Exemple : *Objet paramétré*

L'objet person, décrit ci-dessous correspond en fait à une généralisation de l'objet hamza. Notons qu'avec un descripteur paramétré, nous n'avons plus besoin de chercher explicitement l'information de la partie déclarative de l'objet.

```
person(D1, D2) ::{[person]-
                        chrc->[age]-ptim->[date : D1] ,
                        chrc->[birth]-ptim->[date : D2]}.
person(D1, D2) ::age(Age) :- dtDiff(D1, D2, Age).

I: person((6, december, 1995), (5, april, 1995))::age(Age) .
Age = (1, 8, 0).
yes
I: person(_, _)::{[person]-chrc->[birth]} .
yes
```

L'utilisation des GC présuppose une hiérarchie des types de concepts.

Définition 7 : *Description en Prolog+CG d'une hiérarchie des types de concepts*

La hiérarchie des types de concepts définit la relation de généralisation entre les types et spécifie les extensions éventuelles des types. Chaque type peut avoir une extension (qui est soit une suite d'individus ou le nom d'un prédicat) et un ou plusieurs sous-types.

La description en Prolog+CG d'une hiérarchie de types est comme suit :

```
DeclHier =  Type  [ "="  Extension ]  [ ">" SequenceDeSousTypes ]  "." .
Extension = (Individu1, ..., IndividuN)  |  NomPredicat .
SequenceDeSousTypes  = SousType1, ..., SousTypeM.
```

♦

Exemple :

universal > object, action, attribute.

object > animate, inanimate.

inanimate > table, disk, drive, nail, plier, book, text.

plier > mecPlier, elecPlier.

animate > person.

person > man, woman.

man = (john).

woman = (suzie).

action > extract, grip.

extract = (extr1, extr2, extr3).

attribute > strong.

7.2.2 Objets primitifs

Les objets primitifs de Prolog+CG correspondent aux opérations fournies par C-Prolog sur les données primitives (entier, chaîne, etc.) et aux objets **typeOperations** et **cgOperations** qui offrent respectivement les méthodes/opérations sur la hiérarchie des types et les méthodes/opérations sur les GC. Un message à typeOperations ou à cgOperations a respectivement la forme typeOperations::GC ou cgOperations::GC avec GC qui représente un appel à une méthode.
Les opérations sur la hiérarchie des types et les GC sont définies en annexe A, avec des exemples d'utilisation.

Prolog+CG fournit par ailleurs deux primitives de lecture/écriture pour les GC :

→ **readCG(G)** lit un GC selon la notation linéaire de Prolog+CG et retourne dans son argument G la représentation interne du graphe,

→ **writeCG(G)** reçoit dans son argument une représentation interne d'un GC et produit, comme effet de bord, sa forme linéaire.

Evidement, comme toute autre structure de donnée, un GC peut être ajouté et sauvegardé dans la base (avec un *assert*) et on peut y accéder par la suite.

Exemple : *Utilisation de readCG et writeCG*

| ?- readCG(G), assert(dataCG(G)).

```
|: {[write]-
|:      agnt->[man:john]
|:     obj->[book]-subj->[cognitiveScience]
|:                  price->[dollarUS: 45] ,
|:     ptime->[year:84]}.

G = cg([(ptime,[(1,6)]),(obj,[(1,3)]),(price,[(3,5)]),(subj,[(3,4)]),(agnt,[(1,2)])],
       concs((write,s0z),(man,john),(book,s0z),(cognitiveScience,s0z),(dollarUS,45),(year,84)),varsCG)
yes

| ?- dataCG(G),  writeCG(G).
{[write]-
    agnt->[man: john]
    obj->[book: _*mr3]-
                subj->[cognitiveScience]
                price->[dollarUS: 45] /
    ptime->[year: 84] ,}

G = cg([(ptime,[(1,6)]),(obj,[(1,3)]),(price,[(3,5)]),(subj,[(3,4)]),(agnt,[(1,2)])],
         concs((write,s0z),(man,john),(book,s0z),(cognitiveScience,s0z),(dollarUS,45),(year,84)),varsCG)
yes
| ?-
```

Pour permettre une mise à jour des objets, Prolog+CG offre deux variantes des opérations "assert" et "retract" du langage Prolog :

→ **assertObj(DescrObj)::Fait** est une opération qui ajoute à la description de l'objet DescrObj le fait Fait.

→ **retractObj(DescrObj)::Fait** est une opération qui enlève de l'objet DescrObj le(s) fait(s) qui peu(ven)t s'unifier avec Fait.

Par exemple, le but assertObj(hamza)::{[person]-has->[car]} ajoutera à l'objet hamza le fait qu'il a une voiture. On aura donc comme changement dans le programme :

```
hamza::{[person]-has->[car]}.
hamza:: ... description antérieure .
```

7.2.3 Utilisations de la notion d'objet

Un objet en Prolog+CG peut être utilisé pour formuler différentes notions, par exemple : une théorie avec ses axiomes et ses règles, un module qui offre différents services, la description d'un type de concept en termes de structures conceptuelles, ou une classe avec ses attributs et ses méthodes. Pour ce dernier cas, la partie déclarative d'une classe (l'ensemble de ses attributs) peut être représentée par un GC, comme c'est le cas des objets hamza et person décrits dans la section 7.2.1.

Un type de concept peut être associé à une *définition*, à des *canons* et à des *schémas*. La définition d'un type décrit ce qui caractérise le type de son(ses) super-type(s) direct(s), un

canon d'un type décrit les contraintes sémantiques qui doivent être vérifiées pour une utilisation correcte du type et un schéma décrit une situation commune dans laquelle des types de concepts sont utilisés. Un schéma peut être associé à plusieurs types de concepts.
La forme générale d'un objet qui décrit un type de concept pourrait être comme suit :

```
Type(...)::CGDef :- ... .

Type::canon(Canon1) .
...
Type::canon(CanonN) .

Type::schemas(ListIdentSchemas) .


IdentSchema::CGSchema .
```

Remarques :

→ Avec la hiérarchie des types de concepts et la description d'un type en termes de structures conceptuelles, le programmeur peut définir une base de connaissances qui peut être exploitée par la suite.

→ Un identificateur est associé un schéma et dans la description d'un type on spécifie la liste des identificateurs de ses schémas.

→ Dans la formulation d'une définition d'un type, nous proposons d'utiliser une règle et non un fait et ce, afin de considérer le cas où une définition comprend des contraintes à vérifier. Par exemple, dans la définition d'une "maison luxueuse", on pourrait avoir comme contrainte "le nombre de chambre doit dépasser 15". Nous illustrons cet aspect par la suite, dans le cadre de l'opération d'instanciation. Cette dernière illustre aussi (et justifie) l'utilisation d'un descripteur paramétré pour la "règle-définition".

→ Nous avons encadré la forme ci-dessus par un rectangle "pointillé" pour indiquer qu'elle correspond à une simple suggestion ; elle ne fait pas partie de la syntaxe du langage et une autre formulation peut être utilisée.

→ Les structures conceptuelles utilisées dans Prolog+CG sont celles définies par Sowa [Sowa, 84]. Nous y apportons toutefois les modifications suivantes :

- un schéma peut être associé à plusieurs types de concepts et non à un seul type.
- un concept peut avoir plusieurs canons.
- pour tenir compte du cas où un type est défini comme une spécialisation de plusieurs super-types (et non d'un seul super-type), nous proposons la formulation suivante pour une définition : " [NouvType : super]-...la spécialisation... "; le concept [NouvType : super] indique que NouvType est défini par rapport à son(ses) super-type(s), spécifié(s) dans la description de la hiérarchie des types. C'est comme si le type NouvType dans [NouvType : super] représente la liste de ses super-types. Le concept [NouvType : super] remplace le concept "genre" que l'on

retrouve dans la définition par "genre et différence spécifique", utilisée par Sowa et qui spécifie "le" super-type du nouveau type.

Exemple :

```
person > student, employee.
student > researchAssist.
employee > researchAssist.
researchAssist::{[researchAssist : super]-has->[mailBox]}. /*** definition of researchAssist ***/

plier :: {[grip]-instr->[plier]}. /*** only the canon is provided, for the type plier **/
extract:: canon({[extract]- agnt->[person] obj->[inanimate]}).
extract::schemas([sch1Extr, sch2Extr]).
sch1Extr :: {[extract]- agnt->[person] obj->[text] cible->[book] }. /*** a schema sch1Extr ***/
sch2Extr :: {[extract]- agnt->[person] obj->[inanimate *mr1] manr->[strong]
                        cible->[inanimate]-on->[inanimate *mr1] }.
```

7.2.4 Utilisation des variables en Prolog+CG

Pour une plus grande puissance d'expression, Prolog+CG permet l'utilisation des variables dans différentes situations :

→ *Une variable comme message.* Voici un exemple qui illustre ce type d'utilisation de variables :

```
| ?- quest.
|: plier::Descr.                    /*** the variable Descr as a message ***/
Descr = {[grip]-instr->[plier]}
yes
```

Autre exemple : une règle peut contenir dans sa queue un appel à l'opération jointure qui produirait en sortie un GC, ce dernier est ensuite envoyé comme message. Ceci est rendu possible grâce à la possibilité d'avoir une variable comme message :

```
... ,
cgOperations::{[maximalJoin]- in1->[cg: G1] in2-> [cg: G2] out->[cg: G3]} ,
ObjDescr::G3 ,
... ,
```

→ *Une variable comme tête d'une règle.* Exemple : vérifier si une information G est contenue dans un des schémas d'un type T.

```
checkSchemas(T)::G :-
    T::schemas(ListOfIdSchemas) ,          /*** get the schema identifiers for the type T ***/
    member(SchemaId, ListOfIdSchemas) , /*** take an identifier from the list ***/
    SchemaId::Schema ,                     /*** ask for the description of the schema ***/
    cgOperations :: {[subsume]- in1->[cg: G]  in2->[cg: Schema]  out->[cg: _ ]}.  /*** check if G is
                                                                    subsumed by the schema ***/
```

Utilisation de cet objet :

```
| ?- quest.
|: checkSchemas(extract)::{[extract]-  agnt->[person]    manr->[strong]}.

yes

| ?- quest.
|: checkSchemas(extract)::{[extract]-  agnt->[person]    manr->[strong]  instr->[plier]}.

no
```

Une variable comme tête d'une règle peut être utilisée aussi pour le traitement d'exception : un objet peut avoir une variable comme tête de sa dernière règle et si, suite à un envoi de messages à l'objet, aucune des règles (autre que la dernière) ne peut être utilisée, alors la dernière règle le sera et l'objet pourra alors analyser cette exception.

→ *Une variable comme destinataire d'un message.* L'exemple précédent illustre cette situation en posant la variable T comme destinataire d'un message. Voici un autre exemple : l'objet *transform(X)* demande à *casTransform* de transformer un message G envoyé à un(des) objet(s) X, en un autre message G1 qui doit être envoyé ensuite à ce(s) même(s) objet(s).

```
transform(X)::G :-
    casTransform(G, G1),     X::G1 .

casTransform({[accept]-pat->[person:P]  obj->[entity: E]},
            {[negociate]-pat->[person:P]   with->[person]   obj->[entity: E]} ).
    ...
```

Une règle peut alors contenir cet envoi de message :

```
..., [ socialRel ; transform([socialRel; businessRel]) ]::{[accept]-pat->[man]  obj->[divorce]} , ...
```

on envoit en premier le message {[accept]-pat->[man] obj->[divorce]} à l'objet socialRel, si ce dernier ne peut répondre, le message est envoyé à transform qui doit le transformer et le re-envoyer à socialRel ou l'envoyer à businessRel.

Remarques :

→ Une telle puissance d'expression est rendue possible grâce à l'utilisation de termes comme descripteur d'un objet et de variables comme destinataires de messages et comme têtes de règles.

→ Une variable comme destinataire d'un envoi de messages doit avoir une valeur quand l'interpréteur s'apprête à envoyer le message. Ainsi, dans la règle ci-dessus de transform(X)::G, la variable X doit avoir une valeur avant l'envoi du message X::G1. Autrement, ce dernier aura le sens : "chercher un objet X qui puisse répondre au message G1". Cette possibilité n'est pas offerte par Prolog+CG.

7.3 Syntaxe du langage Prolog+CG

Comme pour le langage Synergy, nous utilisons ici la notation EBNF. [X] signifie que l'élément X est optionnel, X | Y signifie X ou Y, {X} signifie qu'on peut avoir une ou plusieurs occurrences de X, [X]* signifie qu'on peut avoir zéro ou plusieures occurrences de X, les () sont utilisées pour grouper des éléments et % ..% pour les commentaires.

Syntaxe d'un terme

```
Term          = Number | Variable | CG | List | (Atom [ ListOfArgs ]).
List          = '[' [Term ["," Term]*]  ']'.
ListOfArgs    = '(' Term ["," Term]* ')'.
Atom          = LowerLetter [(Letter | digit)]* .
Variable      = (UpperLetter | '_')  [(Letter | digit)]*.
Number        = {digit}.
```

Syntaxe d'un GC

Pour simplifier l'analyse syntaxique et la génération d'un GC, la notation pour ces derniers en Prolog+CG diffère de celle de Sowa [Sowa, 84] par les points suivants :

→ un GC est délimité par '{' et '}' qui annoncent le début et la fin d'un GC. Ils servent aussi pour différencier par exemple entre la liste [personne] et le GC {[personne]} qui est composé d'un concept.

→ une relation n'est pas délimitée par '(' et ')' qui n'ont aucun rôle dans l'analyse. En plus, le GC demeure très lisible même sans ces délimiteurs.

→ pour un concept, seules les branches en sortie sont spécifiées, les branches en entrée sont des branches en sortie d'autres concepts. Par conséquent :

→ un GC est décrit par une collection d' "arbres conceptuels" séparés par '&'.
→ deux concepts en co-référence partagent la même variable comme référent.
→ un référent peut être un concept. Dans ce cas, le concept est précédé par le symbole "!" afin de le différencier d'une liste ; différencier entre la liste [person] et le concept [person].
→ Si un concept est spécifié plusieurs fois dans le même arbre et/ou dans des arbres différents, alors ses occurrences doivent être marquées par une *référence-multiple* qui a la forme *atomic.

Exemple :

```
{[eat]-                                    ou    {[spoon *mr1]-matr->[shell *mr2]  &
   agnt->[monkey]                                  [eat]-
   obj->[walnut]-part->[shell *mr1] ,                 agnt->[monkey]
   inst->[spoon]-matr->[shell *mr1]}                  inst->[spoon *mr1]
                                                     obj->[walnut]-part->[shell *mr2]}
```

Note : Les symboles "-" et "," délimitent les branches en sortie d'un concept. Ainsi, dans :

```
[eat]-
   obj->[walnut]-part->[shell *mr1] ,
   inst->[spoon]-matr->[shell *mr1]
```

la virgule "," indique que inst->[spoon] n'est pas une branche de [walnut] mais de [eat].

Si la forme linéaire d'un GC contient plusieurs occurrences d'un même concept, la référence-multiple éviterait au programmeur de répéter le référent du concept à chaque fois. Par exemple, la proposition "Karl s'envoit le message *la vie est belle* " peut être formulée comme suit :

```
{[envoyer]-
    agnt->[personne : karl ]
   obj->[message: {[vie]-attr->[belle]} *mr2]  &
[recevoir]-
      pat->[personne : karl]
      obj->[message *mr2] }.
```

Notons que nous utilisons un atom est non une variable pour indiquer la référence multiple. Nous réduisons ainsi le nombre de variables à gérer.

Remarque : Comme le montre la syntaxe ci-dessus et pour augmenter la puissance d'expression du langage, des variables peuvent être utilisées à la place d'un concept du GC, d'une relation et du type d'un concept. Elles doivent avoir toutefois une valeur quand une opération d'appariement est appelée pour traiter des GC qui contiennent de telles variables.

Voici maintenant la syntaxe de la notation linéaire d'un GC dans Prolog+CG :

```
CG             = '{'  CGtree  ['&'  CGtree]*  '}'.

CGtree         =  Concept  [ '-'  {OutComeBranch}  [','] ].

OutComeBranch =  RelationType  '->'  CGtree.

Concept        =  Variable |
                  ( '[' ConcType  [':' Referent]  ['*' ManyRef]  ']' ).
ConcType,
RelationType = Atom | Variable.

ManyRef        = Atom | Number.

Referent       =  Atom | Number | Variable | CGList | ('!' Concept) | CG.

CGList         = '[' [ Referent  [','  Referent]* ]  ']'.
```

Syntaxe d'un programme en Prolog+CG

```
Prolog+CGProgram       = {Prolog+CGRule} ".".

Prolog+CGRule  = Head ("." | (":-" PrlgPCGGoal ["," PrlgPCGGoal]* ".")).

Head                   = Predicate |  (HDescriptor_s "::" MethodHead).

PrlgPCGGoal            = Predicate | Variable | SendMessage.

HDescriptor_s          = ObjDescriptor | HConjDescrs.

HConjDescrs            = "[" ObjDescriptor {"," ObjDescriptor} "]".

SendMessage            = Descriptor_s "::" MessageCont.

MethodHead,
MessageCont            = Predicate | CG | Variable.

Descriptor_s           = DescrOrVar | ConjDescrs | DisjDescrs.

ConjDescrs             = "[" DescrOrVar {"," DescrOrVar } "]".

DisjDescrs             = "[" DescrOrVar {";" DescrOrVar} "]".

DescrOrVar             = ObjDescriptor | Variable.

ObjDescriptor          = Predicate.

Predicate              = Atom [ ListOfArgs ].
```

7.4 Exemples de programmation en Prolog+CG

Cette section présente des exemples illustrant la programmation par objets en Prolog+CG.

Exemple 1 : *programme #1 sur "maladie et sentiment "*

Commentaires sur les règles du programme #1 ci-dessous :

→ La première règle spécifie que pour l'objet seriousState, une personne P1 est mourante si l'objet seriousDesease peut prouver que la personne P1 est sensible à une dispute et si l'objet socialRel peut prouver qu'une personne s'est disputée avec la personne P1.

→ La seconde règle spécifie que pour l'objet seriousDesease, une personne P est sensible à une dispute si l'objet disease prouve que P est atteinte d'une maladie cardiaque et que son père est mort à cause d'une maladie cardiaque, si l'âge de P est supérieur à 40 et si les trois derniers tests effectués par P montrent un taux décroissant inférieur à 10.

→ La troisième règle spécifie qu'une personne P1 est en conflit avec une personne P2 si P1 est le conjoint de P2, P1 aime une autre personne P3 et P2 n'accepte pas le divorce.

```
seriousState::{[beDying]-pat->[person: P1]} :-
  seriousDesease::{[sensitive]-pat->[person:P1]  obj->[dispute]},
  socialRel::{[dispute]-agnt->[person]  target->[person: P1]}.

seriousDesease::{[sensitive]-pat->[person:P]  obj->[dispute]} :-
  disease::{[gotDisease]-obj->[diseaseCardiac]
                    pat->[person:P]-fatherOf->[man:M] &
          [beingDead]-pat->[man:M]  cause->[diseaseCardiac]},
  P::{[person:P]-ageOf->[age: A]},
  sup(A, 40),
  disease::{[checkUp]-pat->[person:P]  obj->[test:T]},
  lastElements(3, T, T1),   /** copy in T1 the three last elements of T **/
  rateDecr(T1, Rate),
  inf(Rate, 10).

socialRel::{[dispute]-agnt->[person:P1]  target->[person: P2]} :-
  socialRel::{[person:P2]-conj->[person:P1]},
  socialRel::{[love]-pat->[person:P1]  obj->[person:P3]},
  dif(P2, P3),
  non [socialRel; transform(socialRel)]::{[accept]-pat->[person:P2]   obj->[divorce]}.
...

[disease, seriousDisease]::
            {[checkUp]-pat->[man:jo]-fatherOf->[man:M] ,
                    obj->[test:[80,70,50]]   &
             [gotDisease]-pat->[man:jo]  obj->[diseaseSevere] &
             [beingDead]-pat->[man:M]  cause->[diseaseCardiac] }.

socialRel::{[love]-pat->[woman:valerie]
               obj->[man:andre]-friend->[man:jo]-conj->[woman:valerie]  &
          [refuse]-pat->[man:jo]  obj->[divorce] }.

jo::{[man:jo]-ageOf->[age: 45]}.
.
```

Programme #1 : "maladie et sentiment"

Après compilation du programme ci-dessus, on pourrait poser les questions suivantes :

```
| ?- quest.
|: seriousState::{[beDying]-pat->[person: X]}.
X= jo

yes
| ?- quest.
|: seriousState::{[beDying]-pat->[person: X]-profession->[politician]}.

no
| ?- quest.
|: socialRel::{[dispute]-agnt->[woman: valerie]  target->[man: jo]}.

yes
| ?- quest.
|: socialRel::{[dispute]-agnt->[woman: Agent]  target->[man: Target]}.
Agent = valerie
Target = jo

yes
| ?- quest.
|: socialRel::{[dispute]-agnt->[woman: valerie]  target->[man: andre]}.

no
```

Remarque : L'utilisation des GC permet d'exprimer d'une façon très naturelle, des contraintes structurelles (représentées par les relations entre les concepts) et sémantiques (représentées par les types des concepts). Ce dernier type de contraintes est toutefois rendu flexible grâce à l'exploitation de la hiérarchie des types lors de l'unification.

Ainsi, la seconde règle peut être utilisée pour montrer que "l'employé Antar est sensible aux injures" ; seriousDisease::{[sensitive]-pat->[employee:Antar] obj->[insult]}, qui peut s'unifier à la tête de la règle (Employee est un sous-type de Person et Insult est un sous-type de Dispute). La même règle peut être utilisée aussi pour montrer qu'une "femme W est sensible aux disputes conjugales" :

```
seriousDisease::{[sensitive]-pat->[woman:W]  obj->[conjugalDispute]}.
```

La troisième règle illustre également cette puissance d'expression : dans certaines sociétés, différents couples sont possibles (Homme/Femme, Homme/Homme, Femme/Femme); la troisième règle peut s'appliquer à chacun des cas : une femme F1 peut se disputer avec sa conjointe F2, car F1 aime un homme H, ou : un homme H1 peut se disputer avec sa femme F car H1 aime un autre homme H2, etc.

Par ailleurs et comme le montre les buts qui composent cette troisième règle, l'objet/contexte relSocial comprend une connaissance stratégique sur certaines règles sociales : dispute, mariage, amour, divorce, ... Ceci illustre l'avantage d'organiser une base de règles en plusieurs objets/contextes, chacun représente une expertise particulière.

Exemple : *programme #2 sur la programmation contextuelle en Prolog+CG*

L'exemple montre comment les GC sont utilisés pour représenter, manipuler et raisonner sur une information modale et contextuelle. Il concerne en particulier une situation de fiction, adaptée d'un film et illustre la gestion de croyance. Une formulation en "langue naturelle" des règles du programme sont présentés ci-dessous.

```
investigation::{[isSuspected]-pat->[person:P] obj->[felony:F]} :-
   fanaticBelief(P)::
         {[proposition: {[achieve]-pat->[person]   obj->[action:F]} ]-attr-[modality:possible]}.

fanaticBelief(P)::
   {[proposition: {[achieve]-pat->[person]   obj->[intentionalAct:A]}]-attr->[modality:possible]} :-
   agent(P)::{[believe]-
                  agnt->[person:P]
                   obj->[proposition:{[animate:M]-has->[power]-attr->[magic]}] },
   agent(P)::{[believe]-
               agnt->[person:P]
                obj->[proposition: {[command]-
                                       agnt->[animate:M]
                                     target->[animate]
                                       obj->[intentionalAct:A]}]-
                                                    attr->[modality:possible]
                                                  ctxte->[alteredState]}.

agent(P):: {[believe]- agnt->[person:P]
                    obj->[proposition: S]- attr->[modality:possible]
                                       ctxte->[alteredState]          } :-
   eventOccured(P)::
      {[experienced]- agnt->[person:P]   obj->[alteredState : S]}.

eventOccured(anne):: {[experienced]-
                            agnt->[woman:anne]
                             obj->[lucidDream:
                                      {[command]-
                                            agnt->[tree: anneSpTree]
                                          target->[woman:anne]
                                            obj->[physicalAct:{[kill]-obj->[boy:tomy]}]
                                      }]
                     }.

agent(anne)::
   {[believe]- agnt->[woman:anne]
           obj->[proposition: {[tree:anneSpTree]-has->[power]-attr->[magic]}] }.
```

Programme #2 : situation de fiction et croyance

→ Première règle : une personne est suspectée d'avoir commis un crime si elle croit de façon "fanatique", qu'une personne puisse envisager un tel crime.

→ Deuxième règle : la proposition "une personne peut envisager une action A" est une croyance fanatique tenue par une personne P si :

P croit qu'une entité animée M possède un pouvoir magique et,

P croit que l'entité M peut commander un acte à une entité animée se trouvant dans un état altéré de conscience.

→ Troisième règle : Une personne P croit que la proposition S peut se réaliser dans un état altéré de conscience si elle a expérimenté S durant cet état.

Voici maintenant deux faits (les règles quatre et cinq):

→ Anne a vécu un rêve lucide dans lequel un arbre proche de sa maison lui a commandé de tuer le garçon "tomy".

→ Anne pense que l'arbre proche de sa maison possède un pouvoir magique.

Si on pose la question : "Can we suspect Anne of having killed tomy ?", on aurait la réponse suivante :

```
| ?- investigation::{[isSuspect]-pat->[woman: anne]
                        obj->[felony:{[kill]-obj->[boy: tomy]}] }.
yes
```

Exemple : *quelques aspects de la programmation logique "orientée agent"*

Considérons en premier la notion de groupe d'agents. Un agent A1 peut envoyer un message M1 au groupe Hoppy. Ce dernier doit déterminer l'agent A2, membre du groupe, qui devrait recevoir le message. Par exemple, si le message concerne les multimédia alors il doit être acheminé au Pr. Gagol, s'il concerne la vision alors c'est le Pr. Samuel qui doit le recevoir. Le message peut être ensuite analysé par l'agent selon qu'il s'agisse d'une question, d'une information, ou d'un autre type de message.

La formulation de "l'agent A1 envoit un message M1 au groupe Hoppy" pourrait être comme suit : hoppy::{[message: M1]-sender->[agent: A1]}.

```
hoppy::{[message: {[content:M]-subject->[multimedia]}]-sender->[agent:A]} :-
    gagol::{[content:M]-sender->[agent: A]}.

hoppy::{[message: {[content:M]-subject->[vision]}]-sender->[agent: A]} :-
    samuel::{[content:M]-sender->[agent: A]}.

samuel::{[question: Q]-sender->[agent: A]} :- ... .

samuel::{[information: Q]-sender->[agent: A]} :- ... .
```

Considérons maintenant le domaine de planification et en particulier l'aspect méta-planification. Supposons qu'un plan est décrit par une précondition qui correspond à une situation, le corps du plan et l'intervalle de temps pour réaliser le plan :

[plan:P1]-
precond-> [situation:C1]
body->[scenario:S1]
duration->[period:T1]

Considérons maintenant la règle de méta-planification suivante (la règle est "méta" puisqu'elle ne s'applique pas sur un plan particulier, mais plutôt sur l'interaction entre des plans et autres structures "cognitives") : un agent A1 doit réaliser un plan P1, qu'il ne peut effectuer mais il connait un agent A2 qui doit aussi réaliser un plan P2 et A1 sait que le corps du plan P2 est une partie du corps de P1.
L'agent A1 va demander alors à A2 de réaliser le corps du plan P1 au lieu du corps de P2 (en d'autres termes: il va lui demander de réaliser les parties de P1 qui ne sont pas contenues dans P2) si :
- les deux plans ont des préconditions très similaires, ainsi si l'agent A2 peut réaliser le plan P2 il peut réaliser aussi le plan P1,
- la période de réalisation du plan P2 ne précède pas celle de P1,
- l'agent A1 croit que l'agent A2 peut réaliser les parties du corps de P1 qui ne sont pas contenues dans le plan P2,
- l'agent A2 est "moralement obligé" d'accepter de rendre service à A1.

Nous supposons en plus que cette règle est commune à trois agents (jo, libya et petrolium) :

[jo, libya, petrolium]::
{[toAsk]-agnt->[agent:A1] target->[agent: A2]
obj->[action: {[achieve]-agnt->[agent:A2] obj->[plan:P1]}] } :-
A1::{[know]-agnt->[agent:A1]
obj->[situation:{[unable]-pat->[agent:A1] obj->[action:{[achieve]-obj->[plan:P1]}] }] },
A1::{[behavior]-agnt->[agent:A1] obj->[plan:P1]},
P1::{[plan:P1]-precond->[situation:C1] body->[scenario:S1] duration->[period:T1]},
A1::{[know]-agnt->[agent:A1] obj->[situation:{[achieve]-agnt->[agent:A2] obj->[plan:P2]}] },
P2::{[plan:P2]-precond->[situation:C2] body->[scenario:S2] duration->[period:T2]},
cgOperations::{[subsume]-in1->[cg: S2] in2->[cg: S1] out->[cg:_]}
cgOperations::{[match]-in1->[cg:C1] in2->[cg: C2] out->[cg: C3]},
definedCGOperation::{[similarity]-in1->[cg:C1] in2->[cg: C2] in3->[cg: C3] out->[score:high]},
non before(T2,T1),

cgOperations::{[partialContract]-in1->[cg:S2] in2->[cg: S1] out->[cg: S3]},
A1::{[believe]-
agnt->[agent:A1]
obj->[proposition: {[able]-pat->[agent:A2] obj->[action: {[execute]-obj->[scenario:S3]}] }]},
socialRel::{[willing]-pat->[agent:A2] towards->[agent:A1]}.

Cette règle peut être utilisée par exemple, par Jo qui doit, entre 2h et 4h, ouvrir le bureau 112 et allumer le chauffage. Jo ne peut toutefois pas accomplir son plan mais il connait une personne qui va aussi, à 3h, ouvrir le bureau 112. Dans une telle situation, Jo peut utiliser la règle ci-dessus.
La même règle peut être utilisée dans un contexte politique : Libya veut établir un embargo sur Tanzania et si le gouvernement ne se retire pas, un commando devra être envoyé pour éliminer le président. Libya ne peut toutefois pas réaliser son plan, par contre l'Irak, qui voulait aussi effectuer un embargo sur Tanzania, peut le faire. Libya peut alors utiliser la règle ci-dessus.

7.5 Héritage et instanciation

Des extensions orientées objets de Prolog utilisent l'approche suivante pour réaliser, en logique, l'héritage ([Zaniolo, 84], [Gallaire, 86], [Lou, Ozsoyoglu, 91], [McCabe, 92], [Hui et al., 93]) : la relation "Obj1 est-plus-général-que Obj2" est formulée en générale comme une règle, Obj2 :- Obj1. ; tout ce qui est vrai sur Obj1 le sera aussi sur Obj2. Nous utilisons cette approche pour permettre l'héritage dans une hiérarchie de généralisation entre les objets.

Définition 8 : *Hiérarchie d'objets et sa description en Prolog+CG*
Les objets peuvent être organisés en une hiérarchie d'objets et un super-objet Os d'un objet O est spécifié par la "règle d'héritage" : DesrObjO <- DescrObjOs., DesrObjO et DescrObjOs correspondent respectivement aux descripteurs des deux objets O et Os. ♦

Si un objet a plusieurs super-objets (héritage multiple), il lui sera associé plusieurs règles d'héritage. Ces dernières doivent être spécifiées avant les règles qui définissent l'objet.

Exemple : *Objets et règles d'héritage*
Cet exemple illustre trois points : la spécification de la généralisation entre deux objets, la définition avec contrainte d'un objet (en l'occurrence: la longueur d'un rectangle ne doit pas être inférieure à sa largeur) et l'utilisation des paramètres dans la formulation de la spécialisation d'un objet par rapport à un autre (en l'occurrence, le cas du carré qui est une spécialisation d'un rectangle).

```
rectangle(Length, Breadth)::
        {[rectangle : super]-
                has->[perimeter]
                has->[surface]
                length->[number : Length]
                breadth->[number : Breadth] }  :-  not inf(Length, Breadth) .
rectangle(Length, Breadth)::perimeter(P) :-  P is *(2, +(Length, Breadth)) .
rectangle(Length, Breadth)::surface(S) :-  *(Length, Breadth) .

square(Cote) <- rectangle(Cote, Cote) .
square(_)::{[square]-attr->[beautiful]}.
```

Définition 9 : *Héritage entre objets*

Si un message est envoyé à un objet et les règles qui définissent ce dernier ne peuvent y répondre, alors les règles d'héritage de l'objet sont considérées une à la fois selon l'ordre d'écriture, afin de "déléguer" aux super-objets de l'objet la responsabilité de répondre au message. ♦

Exemple : *Envoi de messages et héritage*

Considérons la question suivante : square(6)::surface(S) . L'interpréteur cherche en premier une méthode associée à l'objet "square(X)" qui puisse s'unifier avec surface(S) , dans notre cas square(X) n'a pas une telle méthode, l'interpréteur considère donc les règles d'héritage de square(X) , s'il en a. Il trouve une règle d'héritage pour square(X) et transforme donc la question en une nouvelle requête : rectangle(6,6)::surface(S).

```
I: square(6)::surface(S) .
S = 36
yes
```

La requête précédente illustre l'héritage des méthodes, le même traitement s'applique aussi à l'héritage des attributs. Par exemple :

```
I: square(_)::{[square]-has->[perimeter]} .
yes /*** la hiérarchie des types de concepts contient la déclaration : rectangle > square ***/
```

Considérons maintenant le "processus d'instanciation" et sa formulation logique. Rappelons en premier qu'une instance "hérite" de sa classe, aussi bien la description/définition que les méthodes. Par ailleurs, l'instance se caractérise en général de la classe par les valeurs particulières données à certains attributs. Ces deux aspects sont formulés conjointement et de

façon "naturelle" par la règle d'héritage. Par exemple, la règle monRectangle <- rectangle(8,6) spécifie que monRectangle est une instanciation de rectangle avec Longueur := 8 et Largeur := 6 comme valeurs particulières et permet également à monRectangle d'hériter de rectangle.
Ainsi, dans la formulation logique, au lieu de créer explicitement une instance, par une copie de la définition (ou de la partie descriptive) de la classe et de l'ajouter au programme, il suffit d'établir "une implication" entre le nom de l'instance et un appel particulier de sa classe.
"Créer" une instance consiste donc à ajouter une règle d'héritage au programme. Prolog+CG offre une opération primitive, appelée *createInstance*, qui effectue un tel traitement.

Définition 10 : *Instanciation avec* createInstance
La création d'une instance Inst d'une classe avec DescrClasse comme descripteur, est effectuée par un appel à l'opération primitive **createInstance** : createInstance(Inst, DescrClasse). ♦

Exemple : *Créer et interagir avec des instances*

```
ProgP :-
    ...
    createInstance(salim, person((6, december, 1995), (8, june, 1993)) ) ,
    ...
    salim::age(Age) ,
    ... .
```

L'opération createInstance dans ProgP va ajouter au programme la règle suivante :

```
salim <- person((6, december, 1995), (8, june, 1993)).
```

Comme dans les exemples précédents, pour satisfaire le but salim::age(Age) le langage commence par chercher la description de l'objet salim, il trouve uniquement la règle d'héritage qui est alors directement utilisée. Le but précédent est donc transformé comme suit :

```
person((6, december, 1995), (8, june, 1993))::age(Age) .
```

Remarques :
→ Un message peut être envoyé à une instance ou à une classe, les deux sont représentées par des objets et l'envoi de message est défini en terme d'objet, peu importe ce qu'il représente.
→ Rappelons que des contraintes peuvent être associées à la définition d'une classe. La "création" d'une instance peut "échouer" si les contraintes associées à la définition ne sont pas respectées. Par exemple, createInstance(maFigure, rectangle(5, 7)) ; maFigure ne peut être un rectangle.

Exemple *: mise à jour des classes et des instances*

Considérons la classe suivante (on spécifie uniquement la partie déclarative).

```
room(T, C)::{[room]-
                has->[temperature : T]
                poss->[heater]-has->[checkP : C]
                has->[amountChange : 1]}.
```

On pourrait ensuite créer des instances de chambre comme par exemple : createInstance(room2, room(13, 15)), et si on demande par la suite : room2::{[amountChange : X]}, on aurait comme réponse : X = 1. Si par la suite, on met à jour l'objet/classe room comme suit : assertObj(room(T,C))::{[room]-has->[amountChange : 2]}, et si on repose la question : room2::{[amountChange : X]}, on aurait alors comme réponse : X = 2.

Notons par ailleurs que la valeur 1 (ainsi que 2) joue le rôle de valeur par défaut. Les différentes instances de chambre "hérite" cette valeur.

Au lieu de spécifier la mise à jour au niveau de la classe, on pourrait la spécifier au niveau d'une instance particulière (les autres instances ne seront donc pas affectées par un tel changement). Par exemple : assertObj(room2)::{[room]-has->[amountChange : 3]}. Suite à cette opération, on aurait alors dans la base :

```
room2 <- room(13, 15).
room2::{[room]-has->[amountChange : 3]}.
```

7.6 Environnement et "compilateur" de Prolog+CG

La Figure 7.1 décrit la composition générale de l'environnement de Prolog+CG, une brève description des fichiers suit.

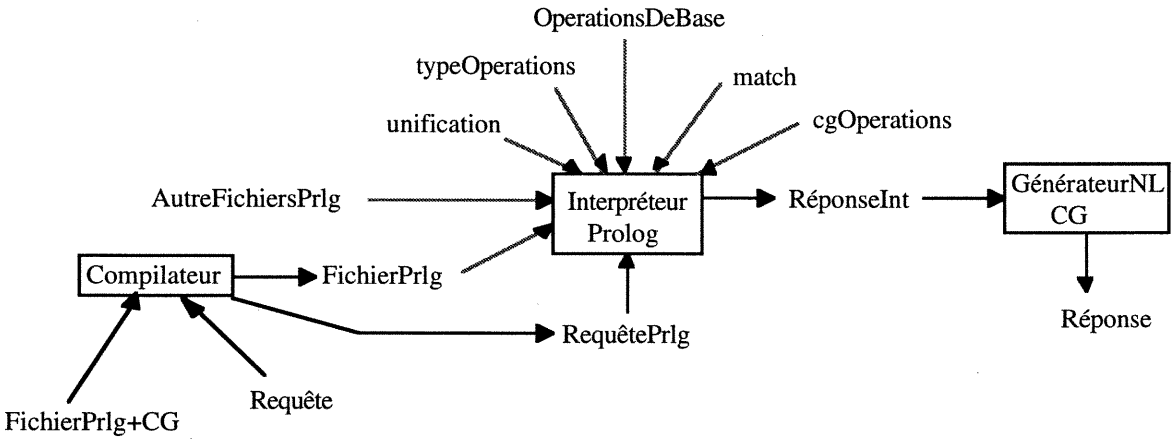


Figure 7.1 : Description générale de l'environnement de Prolog+CG

Le compilateur compile un programme Prolog+CG (ou une requête) en un programme Prolog (ou requête "Prolog") équivalent. La requête compilée est ensuite interprétée. La compilation d'un programme Prolog+CG produit un fichier Prolog qui est sauvegardé par la suite. Ce dernier peut être par la suite directement chargé (par "reconsult" de C-Prolog), avec d'autres fichiers de Prolog sans passer par le fichier source. Bien sûr, une mise à jour de ce dernier impliquerait sa recompilation et donc une mise à jour du fichier objet.
Le générateur produit les GC, résultant d'une requête, selon la grammaire de Prolog+CG.
Cinq fichiers pour les opérations primitives de Prolog+CG sont utilisés au besoin : opérations de base comme member, append, ..., opérations sur la hiérarchie des types, opération match, opérations sur les GC dérivées de match et enfin l'opération d'unification de deux GC.

Considérons maintenant les règles de traduction (noté: CodeSource ==> CodeCible) d'une règle Prolog+CG, d'un envoi de message et d'une règle d'héritage.

→ Une règle Prolog+CG est traduite comme suit (**traduire** souligne l'application récursive des règles de traduction) :

t(P1, ..., PN)::Tete Queue. ==> t(X, P1, ..., PN) :- unify(X, Tete) et **traduire** Queue .

X est une variable introduite par le compilateur. Si le descripteur est un "atom", il est alors transformé en un prédicat avec un argument.
Le cas d'une règle partagée par plusieurs objets :

```
[ObjDescr1, ... , ObjDescrN]::Tete Queue  ==>
                                        traduire  ObjDescr1::Tete Queue.
                                        ...
                                        traduire  ObjDescrN::Tete Queue.
```

L'opération unify(X, Y) est définie comme suit :

```
unify(X, Y) :-
    (isVariable(X) ; isVariable(Y) ; (isPredicate(X), isPredicate(Y)) ), !,
    X = Y.
unify(X, Y) :-
    isCG(X), isCG(Y), !,
    unifyCG(X, Y), !.
```

→ Un envoi de message simple est traduit comme suit :

t(a1, ..., aN)::Message ==> t(Message, a1, ..., aN).

On peut facilement constater que cette traduction est fidèle à la sémantique du langage Prolog+CG, concernant l'évaluation d'un envoi de message : le destinataire du message est à unifier avec le descripteur préfixant une règle et le message est à unifier avec la tête de la règle. Si nous considérons la traduction d'une règle et d'un envoi de message, nous avons en effet t(Message, a1, ..., aN) à unifier avec : t(X, P1, ..., PN) :- unify(X, Tete). Suite à l'unification de t(Message, a1, ..., aN) avec t(X, P1, ..., PN), X sera instanciée à Message qui est ainsi acheminé à unify(X, Tete).

L'envoi d'un message à plusieurs objets est traduit en une conjonction d'envoi de messages (le même message) :

[ObjDescr1, ..., ObjDescrN]::Message ==>

traduire la conjonction de buts Prolog+CG ObjDescr1::Message, ..., ObjDescrN::Message.

De même, l'envoi d'un message à un objet parmi plusieurs est traduit en une disjonction d'envoi de messages :

[ObjDescr1; ...; ObjDescrN]::Message ==>

traduire la disjonction de buts Prolog+CG (ObjDescr1::Message ; ... ; ObjDescrN::Message).

→ Une Règle d'héritage est traduite comme suit :

DescrObj1 <- DescrObj2 ==> **traduire** la règle Prolog+CG DescrObj1::X :- DescrObj2::X .

X est une variable ajoutée par la procédure de traduction, elle permet de transférer à l'objet DescrObj2 la question adressée à DescrObj1.

7.7 Résumé

Nous avons introduit dans ce chapitre le langage Prolog+CG ; une extension conceptuelle, contextuelle et orientée-objet du langage Prolog. L'extension conceptuelle se base sur la théorie des GC et l'extension orientée-objet se base sur l'interprétation logique de la programmation orientée-objet. Ces extensions de Prolog sont intégrées à Prolog+CG comme suit : un but en Prolog+CG peut être représenté par un terme Prolog ou un GC, diverses opérations sur les GC sont offertes comme des opérations primitives, un programme en Prolog+CG est organisé en une collection d'objets, chacun est composé d'une suite de règles. Les objets peuvent former en plus une hiérarchie avec possibilité d'héritage et un objet peut être utilisé pour représenter une classe ou pour spécifier une instance d'une classe : des règles d'un(e) objet/classe peuvent correspondre à la partie déclarative d'une classe alors que d'autres aux méthodes de la classe, la satisfaction d'un but concernant l'objet correspondrait alors à un envoi de messages.

Prolog+CG offre aussi la possibilité d'avoir des GC imbriqués, permettant ainsi la formulation de règles d'inférence et de manipulation de contextes (un contexte est représenté par un GC).
Par ailleurs, la compilation de plusieurs fichiers à la C-Prolog fournit un autre moyen de programmation modulaire : une "grande" application en Prolog+CG peut être constituée de plusieurs fichiers Prolog et Prolog+CG.
Enfin, notons que Prolog+CG est "ouvert" à Prolog : un fichier Prolog+CG peut contenir des règles Prolog, une règle Prolog+CG peut contenir non seulement un envoi de messages mais aussi un but qui correspond à une primitive ou qui est défini par des règles Prolog et une règle Prolog dans un fichier Prolog+CG peut contenir un envoi de message.

Différentes applications peuvent être réalisées en Prolog+CG. Enfin, ce dernier est ouvert à plusieurs extensions et développements futurs.

Chapitre 8

Applications, originalités, limites et développements futurs

Ce chapitre présente les applications, originalités, limites et développements futurs pour les différentes composantes de notre système: opérations sur les GC, langage Prolog+CG, langage Synergy et le modèle de la mémoire.

8.1 Applications du système

8.1.1 Applications des opérations sur les GC

Considérons tout d'abord l'utilisation d'opérations sur les GC au sein même de notre système : l'unification est essentielle pour l'interpréteur de Prolog+GC, elle est utilisée également par Synergy pour la reconnaissance d'un message suite à des envois/réceptions de messages. La généralisation est utilisée par notre modèle de la mémoire, ainsi que la jointure et la contraction.

Dans Kabbaj et Frasson [Kabbaj et Frasson, 93a] nous avons discuté des utilisations possibles des opérations en acquisition des connaissances. La généralisation est une opération essentielle en apprentissage et l'opération EstPlusGeneral est essentielle pour un système de recherche d'information.
Des chercheurs ont utilisé des opérations similaires aux notres dans différents domaines ([Nagle et al., 92], [Pfeiffer and T. E. Nagle, 92], [Mineau et al., 93], [Tepfenhart et al., 94] et [Ellis et al., 95]). Par exemple, dans un système de traitement du langage naturel, les sens d'un mot peuvent être représentés par des GC et le sens d'une phrase peut être composé par des jointures successives des sens des mots [Sowa et Way, 86]. On peut y joindre aussi des connaissances d'arrière-plan représentées également par des GC. L'opération d'analogie peut aussi être utilisée pour traiter la métaphore. Enfin, la contraction peut être utilisée dans la génération de texte afin de produire des textes plus abstraits et plus concis.

8.1.2 Applications de Prolog+CG

Prolog+CG peut être utilisé comme langage de base pour le développement de logiciel et de base de connaissances, pour le traitement du langage naturel, pour les bases de données déductives orientées objets, pour les systèmes de planification, pour le développement d'un "shell" de systèmes experts comprenant l'acquisition et l'explicitation des connaissances, pour les systèmes tutoriels intelligents où Prolog+CG peut être utilisé pour représenter et gérer la connaissance de l'étudiant, de l'expert et du tuteur et en particulier, pour les systèmes tutoriels construits autour des systèmes experts comme GUIDON et ses descendants [Clancey, 87].

8.1.3 Applications de Synergy

Nous avons montré dans les chapitres 4 et 5 des exemples d'utilisation de Synergy. Dans ([Kabbaj, 93] et [Kabbaj et Frasson, 93]) nous avons souligné que les opérations sur les GC (ou autres opérations définies en Synergy) peuvent être utilisées pour produire un GC qui sera ensuite activé. Des connaissances procédurales (comme des tâches) peuvent être ainsi composées dynamiquement et ensuite exécutées.

Sowa [Sowa, 84] propose un exemple d'utilisation des acteurs en bases de données. Il propose une méthode dynamique pour composer le graphe de travail à partir des schémas afin de trouver la réponse à la question (requête) posée par l'usager. Nous avons montré dans le chapitre 4 comment sa méthode peut être implantée en Synergy. D'autres utilisations de Synergy en bases de données (actives) peuvent être envisagées, comme celles proposées dans [Gehani et al., 92].

La famille des diagrammes de transition d'états (et en particulier les réseaux de transition augmentés) est très utilisée en génie logiciel et dans les méthodes de conception ([Harel, 90], [Pressman, 92], [Mellor et Shlaer, 92], [Coleman et al., 92]), en intelligence artificielle (par exemple, dans l'analyse du langage naturel [Kabbaj, 91]) ainsi que dans les systèmes tutoriels intelligents ([Wenger, 87], [Psotka et al., 88], [Frasson et al., 92]). Dans le chapitre 4 nous avons montré comment Synergy peut être utilisé pour représenter et exécuter des diagrammes de transition d'états.

Un des domaines majeurs d'application de Synergy est la simulation, aussi bien quantitative ([Schruben, 83], [Szymankiewicz et al., 88], [Pidd, 88], [Törn, 88], [Widman et al., 89], [Hoover et et Perry, 89]) que qualitative ([Widman et al., 89], [Weld et de Kleer, 90], [Singh and Travé-Massuyès, 91]). En effet, on peut décrire, simuler et diagnostiquer un système dynamique en utilisant Synergy. Quelques caractéristiques des systèmes

dynamiques sont le traitement orienté propagation (de données et de signaux), la modularité dans la description (afin de gérer la complexité de ces systèmes), les données qui varient avec le temps et les boucles rétroactives (feedback loops). Ces caractéristiques (ainsi que d'autres) peuvent être formulées et analysées à l'aide de Synergy [Kabbaj et Frasson, 94]. Les systèmes dynamiques sont étudiés dans divers domaines comme la simulation, les systèmes d'aide à la décision [Singh and Travé-Massuyès, 91] et la simulation qualitative.
Synergy peut être utilisé aussi pour développer un "système de raisonnement situé", comme celui proposé par Ingrand et al. [Ingrand et al., 92]. Leur système est capable de raisonnement orienté but en tenant compte, en temps-réel, de l'environnement qui change continuellement. Dans les chapitres 4 et 5 nous avons montré comment Synergy peut être utilisé pour réaliser un "traitement situé".

Plusieurs environnements tutoriels, basés sur la simulation des systèmes, ont été développés dans le cadre des systèmes tutoriels intelligents (SOPHIE [Brown et al., 82], STEAMER [Hollan et al., 84], SHERLOCK [Lesgold et al., 92], voire aussi [Kieras, 88, 90] et [Psotka et al., 88]).
Enfin, la simulation est aussi à la base des environnements d'apprentissage comme GIL [Reiser et al., 88], LISPITS [Corbett et Anderson, 92] et PROUST [Sack et Soloway, 92] et des environnements de formation et d'acquisition des connaissances procédurales ([Towne et Munro, 88], [Bourne et al., 88], [Frederiksen et al., 90], [Tiberghien et Mandl, 92], [Kabbaj et Frasson, 93]). Synergy peut être utilisé pour ce type de simulation.
Il peut être aussi utilisé comme environnement tutoriel pour enseigner les concepts de base de différents modèles de programmation ainsi que les nouveaux concepts qui résultent de l'intégration.
Enfin et comme nous l'avons souligné dans le chapitre 5, Synergy peut être utilisé comme une plate-forme générale pour la conception et la réalisation de différents modèles d'agents et de systèmes multi-agents.

8.1.4 Applications du processus de formation de la mémoire dynamique

Les applications possibles du modèle de la mémoire sont très variées : la recherche d'information, le traitement du langage naturel, la planification, le raisonnement et l'apprentissage multi-stratégies (dans [Kabbaj et Frasson 93d] nous avons discuté de cet aspect plus en détail) font partie des domaines d'application.
Une autre application majeure (à très long terme) serait d'utiliser la mémoire, en conjonction avec Synergy, dans un cadre multi-agents : développer un système multi-agents où chaque agent incorpore une personalité propre avec toutes les caractéristiques "cognitives" et "réactives" (connaissances, croyance, intentions, sentiments et émotions, caractère, capacités

linguistiques, capacités de perception et de réaction, capacités de raisonnement et de comportement, ...) et une mémoire qui serait utilisée par les différents processus cognitifs. Avec une telle application, on pourrait analyser le comportement de la mémoire d'un agent dans un contexte "naturel" similaire à celui d'un humain. Notons que c'est ce type de mémoire qui constitue l'objet principal de notre modèle.

8.2 Originalités du système

8.2.1 Originalité des opérations sur les GC

Nous avons proposé une définition simple et efficace des opérations sur les GC, définition basée sur l'appariement. Nos opérations sont définies sur un type restreint de GC (une forme plus générale de GC fonctionnels), mais cela constitue un compromis nécessaire pour avoir une réalisation efficace des opérations. Le type de GC que nous avons considéré offre une puissance d'expression suffisante pour modéliser plusieurs domaines d'application. Il est par ailleurs plus général que la majorité des structures typées avec attributs, utilisées de plus en plus en intelligence artificielle [Carpenter, 92].

Nous avons par ailleurs étudié l'appariement des GC composés en tenant compte des co-référents. Peu de chercheurs ont considéré cette extension des opérations sur les GC ([Kabbaj, 87], [Chan, 88], [Leishman, 89], [Esch et Levinson, 95]). De plus, nous avons montré le rôle important des co-référents dans la détermination des points d'entrée pour des GC imbriqués. De plus, d'autres moyens pour déterminer les points d'entrée ont été proposés.

Notre algèbre des GC se caractérise par le nombre des opérations offertes et aussi par leur organisation en une hiérarchie qui montre clairement ce qui est commun et ce qui caractérise chaque opération. L'algèbre est intégrée à deux langages de programmation : Prolog+CG et Synergy, l'un avec une interface textuelle et le second avec une interface graphique.

8.2.2 Originalité de Prolog+CG

Prolog+CG apporte une extension conceptuelle et orientée-objet du langage Prolog. L'extension conceptuelle se base sur la théorie des GC et l'extension orienté-objet se base sur l'interprétation logique de la programmation orientée-objet. Prolog+CG intègre également notre algèbre pour les GC et il ne masque pas Prolog.

Prolog+CG constitue le premier langage de programmation logique contextuelle orientée objet pour les GC.

8.2.3 Originalité de Synergy

Synergy est le premier langage multi-paradigme basé sur l'activation des GC. Il intègre en effet différents modèles de programmation séquentielle et/ou parallèle : modèle procédural, modèle fonctionnel et modèle orientée-objet. Dans les chapitres 4 et 5 nous avons montré comment Synergy peut aussi être utilisé pour réaliser d'autres modèles de programmation (par exemple, programmations par accès, par événement, par processus concurrents et par agents).
Synergy est un langage graphique (visuel); son environnement correspond à un éditeur "élaboré" de GC.
Le caractère multi-paradigme du langage se base sur un modèle "simple" et uniforme : *tout est GC, composé de concepts qui peuvent être actifs et de relations qui peuvent propager l'activation* *(note : on ne peut empêcher notre "processus cognitif" de penser aux réseaux connexionistes ...).*

En particulier, Synergy utilise un même formalisme, les GC, pour formuler aussi bien les connaissances déclaratives que les connaissances procédurales. De même, un même mécanisme de traitement (l'activation de GC) est utilisé pour "réaliser" différents modèles de traitement. Aussi, une partie de sa puissance d'expression est basée sur l'interprétation "procédurale" de la notion de contexte (qui est abscente dans plusieurs formalismes de réseaux sémantiques) et de la notion de co-référence.

En représentant les connaissances procédurales en GC et en fournissant un grand nombre d'opérations sur les GC, Synergy offre au programmeur, la possibilité de construire par "composition dynamique", une connaissance procédurale (une tâche par exemple) à partir d'autres connaissances, le résultat de la composition peut être activé afin d'évaluer par exemple l'utilité de la structure construite (par composition).
Par ailleurs, rappelons que les GC sont très utilisés dans le traitement du langage naturel (analyse et génération), en représentant les connaissances procédurales par des GC, on "avance d'un pas" vers une "programmation interactive en langue naturelle" (une première étape serait une interface en langue naturelle associée à Synergy).

8.2.4 Originalité du modèle de la mémoire dynamique

Notre modèle de la mémoire dynamique peut être considéré comme une "réalisation particulière" du cadre général (framework) proposé par Schank [Schank, 82, 91]. En effet, d'autres réalisations ont été proposées en littérature ([Riesbeck et Schank, 89], [Kolodner, 93], [Pazzani, 90]), mais la problématique sous-jacente à notre modèle est différente de celles adressées par ces travaux; nous sommes concernés en particulier par l'intégration de

différentes structures conceptuelles en mémoire, indépendamment d'un domaine et d'un processus particulier qui utiliserait la mémoire. D'autres points importants du modèle sont : utilisation des GC comme formalisme de représentation des connaissances, "explicitation automatique" de la connaissance à intégrer, utilisation conjointe d'un mécanisme de connexion et d'un mécanisme d'intégration "en profondeur", utilisation de la notion de focus pour diriger et guider le mécanisme d'intégration, traitement du rapport entre les structures conceptuelles (en l'occurrence, entre définition, schéma et synonymie de type) et une interface graphique qui supporte tous les éléments du modèle.
Le modèle que nous proposons est complémentaire aux travaux antérieurs ; nous avons focalisé sur des aspects de la mémoire dynamique qui ont reçu peu d'attention auparavant, les autres travaux ont toutefois illustré les utilisations possibles de la mémoire dynamique (ce que nous n'avons pas considéré dans cette thèse).

8.3 Limites et développements futurs du système

8.3.1 Limites et développements futurs des opérations sur les GC

Notre définition des opérations est procédurale et informelle. Il serait approprié de la compléter par une définition formelle. Il en est de même pour notre étude de la famille des algèbres pour les GC ([Kabbaj, 93], [Kabbaj et Frasson, 94, 95]). Dans [Kabbaj et Frasson, 95] nous avons proposé une algèbre pour les GC avec information par défaut. Il serait approprié de l'implanter et la tester sur différents exemples. Il en est de même pour d'autres algèbres de la famille des algèbres.
Dans [Kabbaj et Frasson, 93c], nous avons proposé un tuteur pour les opérations sur les GC. L'implantation du tuteur et son évaluation peut aider la compréhension de ces opérations.
Dans [Kabbaj et Frasson, 93a] nous avons discuté des utilisations possibles des opérations en acquisition des connaissances. Une utilisation effective, dans différentes applications, montrerait l'importance exacte de ces opérations dans un tel domaine. Il serait également approprié de tester et d'utiliser les opérations dans différents domaines d'application comme les systèmes tutoriels intelligents, le traitement du langage naturel, les bases de connaissances, les systèmes d'information, l'exploration de la mémoire, etc.

8.3.2 Limites et développements futurs de Prolog+CG

Nous avons réalisé une première implantation de Prolog+CG en C-Prolog qui est appropriée pour le développement du prototype. Une prochaine étape serait de réaliser directement l'interpréteur de Prolog+CG (en C++ par exemple). La nouvelle implantation serait plus efficace et permettrait de plus un traitement plus approprié des prédicats avec des GC comme

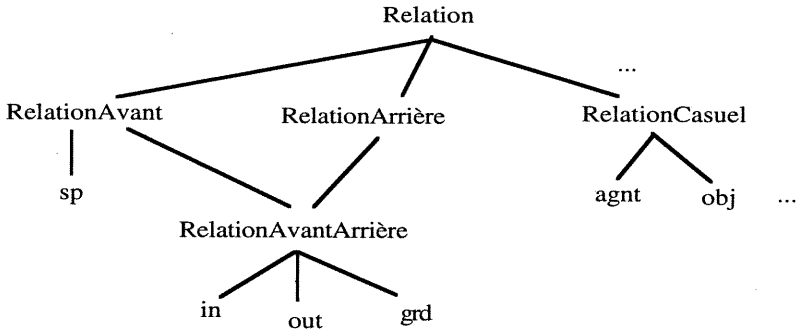
arguments. En effet, l'unification de deux prédicats est effectuée présentement par l'unification de Prolog. Ainsi, si un prédicat contient un GC comme argument, alors ce dernier est considéré selon sa structure interne; une liste (et non comme un GC) et l'unification de deux arguments (qui sont des GC) s'effectuera selon l'unification des prédicats et non selon celle des GC. Dans la nouvelle implantation, l'unification des prédicats serait modifiée afin d'activer l'unification des GC si deux arguments à unifier sont des GC.
Parmi les autres développements futurs, citons :

- Implanter de tous les éléments du langage Prolog+CG et utiliser du langage dans différentes applications.
- Envisager d'autres réalisations de la programmation logique orientée objet (et évaluer les différentes approches). Considérer en l'occurrence le cas de Prolog++ [Moss, 94].
- De part son implantation, Prolog+CG est flexible concernant l'opération d'unification pour les GC : par défaut, l'opération unify(X, Tete) fait appel à unifyCG dans le cas où le message et la tête sont deux GC. Au lieu de considérer par défaut l'opération unifyCG, le langage pourrait demander au programmeur de spécifier une autre opération. De plus, différentes opérations peuvent être adoptées comme différentes "versions d'unification" pour des objets différents. Par exemple, pour un objet, le programmeur peut spécifier une opération d'unification qui est non-déterministe et pour un autre objet il peut spécifier l'opération EstPlusGeneral comme "version de l'unification".
- Développer un "shell" pour les systèmes experts avec un chaînage mixte (arrière et avant), une gestion de l'incertitude et une interface en langue naturelle.
- Traiter le problème, connu dans la programmation logique et les bases de données déductives, du découpage (splitting) d'un message en "sous-messages" lorsque le message initial ne peut être satisfait. Dans la communauté des GC, ce problème a été soulevé par ([Fargues and al., 89], [Fargues, 92]) et a été repris ensuite par [Boksenbaum and al., 93] dans le cadre d'un système de traitement de requêtes pour les bases de données relationnelles.
- Traiter le problème classique de "l'appariement sémantique". Illustrons ce problème par l'exemple suivant : le GC {[refuser]-pat->[personne] obj->[divorce]} ne peut s'unifier (selon la définition classique de cette opération) avec le GC : {[catholique]-poss->[age:45]}, mais si on effectue une jointure de la définition du type Catholique avec le second graphe, l'unification deviendrait possible. Nous prévoyons l'utilisation de notre modèle de la mémoire pour analyser en profondeur ce problème.

8.3.3 Limites et développements futurs de Synergy

- Développer des applications dans différents domaines. On pourrait alors mieux évaluer les aspects conception, implantation et environnement graphique du langage.

- Améliorer l'implantation actuelle de Synergy qui constitue un premier prototype.
- Considérer d'autres données primitives (comme les vecteurs, les tuples, les fichiers, base de données, etc). Aussi, l'ensemble des opérations primitives est actuellement fixé au départ, une solution plus appropriée serait de permettre au programmeur de définir un traitement dans un langage hôte (C++ ou un autre langage) et de pouvoir l'invoquer ensuite à partir d'un programme Synergy, comme opération "externe". Par ailleurs, une solution plus élaborée doit être adoptée concernant le rapport temps-Synergy et temps-cpu (actuellement, le temps d'exécution de toute opération primitive est fixé à 2 unités de temps-Synergy).
- Considérer l'intégration d'objets multi-médias (image, son, texte et vidéos) et le développement d'applications multi-médias. Considérer aussi la possibilité d'associer des icônes aux composantes d'un GC, une telle association permettrait entre autres, le passage d'un GC qui représente la structure d'une machine par exemple, à l'image (ou à la représentation icônique) de la machine et elle permettrait aussi le passage d'une animation d'un GC à l'animation de la représentation icônique de la machine.
- Approfondir l'intégration des modèles de traitement et la consolider avec différentes applications.
- Envisager la possibilité d'une définition locale et contextuelle des types "auxiliaires". Actuellement, la définition d'un type est effectuée au niveau de la mémoire à long terme. Toute définition est donc globale. Dans certaines applications, des types sont définis uniquement pour rendre la formulation plus abstraite et plus concise ; ils sont donc "auxiliaires" et devront être définis localement au(x) contexte(s) qui les utilise(nt). Ceci éviterait une surcharge de la mémoire. Il faudrait donc étudier cette option et considérer en particulier son rapport avec la hiérarchie globale (la mémoire).
- Améliorer (si possible) la formulation et la réalisation de l'interpréteur de Synergy.
- Améliorer et compléter l'environnement graphique.
- Compléter notre définition opérationnelle de Synergy par une définition formelle.
- Considérer la possibilité d'associer des informations aux relations procédurales (comme le poids, le degré de certitude, la fréquence d'occurrence, etc.) et de pouvoir accéder et modifier ces informations. Cette extension permettrait la formulation et l'activation en Synergy de différentes formes de réseaux "pondérés", en l'occurrence les réseaux sémantiques connexionistes.
- Considérer la programmation par satisfaction de contraintes.
- Considérer la possibilité d'étendre la capacité de propagation à des relations autres que les relations procédurales. Comme pour notre distinction entre différentes formes d'activités (ProcedureActivity, ProcessActivity, StrictActivity, LazyActivity), on pourrait distinguer par exemple entre RelationAvant et RelationArrière et avoir la hiérarchie suivante :



Ainsi, selon cette hiérarchie, le lien de spécialisation/généralisation "sp" serait aussi capable d'effectuer une propagation en avant. Une implantation en Synergy de l'héritage par propagation de marqueurs serait alors plus "naturelle" et plus concise.

- Considérer la possibilité d'une implantation sur une machine parallèle. L'implantation actuelle de Synergy simule le parallélisme sur une machine séquentielle. Il serait approprié de considérer une implantation "physiquement" parallèle, avec plusieurs processeurs.
- D'autres limites/développements futurs peuvent être identifiés durant une utilisation continue et à grande échelle de Synergy et de son environnement graphique.

8.3.4 Limites et développements futurs du modèle de la mémoire dynamique

- Utiliser le modèle de la mémoire dans différents domaines et par différents processus "cognitifs" comme le processus de question/réponse, le traitement du langage naturel, le raisonnement, la planification et l'apprentissage (y compris l'apprentissage du langage). Il importe de considérer à cet effet les travaux effectués par le groupe de Schank ([Schank, 82, 86, 88, 91], [Lebowitz, 83, 86, 90], [Kolodner, 84], [Kolodner et Riesbeck, 86], [Schank et Leake, 89]) et ceux proposés dans le cadre des systèmes à base de cas ([Riesbeck et Schank, 89], [Kolodner, 93], [Wess et al., 93], [Veloso et Aamodt, 95]).
- Dans [Kabbaj et Frasson, 93b,c] nous avons proposé un modèle de mémoire basée sur la généralisation, défini pour une représentation des connaissances avec une liste d'attributs-valeurs. Le détail du modèle diffère de celui présenté dans cette thèse. Dans les cas où la représentation avec attributs-valeurs est suffissante, il serait approprié de considérer l'autre modèle. Un développement de ce dernier est donc nécessaire.
- La notion de focus est centrale dans notre modèle, elle permet de guider l'intégration de l'information en mémoire. Présentement, les éléments en focus sont une entrée du processus de formation de la mémoire; nous n'avons pas considéré les mécanismes qui déterminent de tels éléments (par exemple le processus de compréhension du langage naturel). Notons également que l'intégration d'une nouvelle information n'est pas un acte

isolé, il est plutôt situé dans un contexte plus général et souvent des éléments en focus pour une nouvelle information sont déterminés par les intégrations précédentes.
Il importe donc de considérer plus en détail cet axe de recherche.

- Il serait approprié de considérer, conjointement, la formation et l'utilisation de la mémoire. Dans [Kabbaj et Frasson, 93a], nous avons montré comment des mécanismes d'inférence (comme l'induction, l'analogie et l'abduction) peuvent être déclenchés durant l'intégration d'une nouvelle information. Murray [Murray, 90] propose une approche similaire concernant l'impact de l'intégration des connaissances dans une base de connaissances ; une nouvelle connaissance peut déclencher des règles qui pourraient la compléter et susciter des conflits avec les connaissances existantes. Il faudrait approfondir cet axe de recherche et considérer en complément les travaux de Michalski et Tecuci ([Tecuci et Michalski, 1991], [Michalski, 93, 94], [Michalski et Tecuci, 94], [Tecuci, 94, 95]) sur l'apprentissage multi-stratégies où plusieurs stratégies d'apprentissage y sont intégrées (apprentissage par induction, par généralisation, par explication, par abduction et par analogie) afin de traiter des problèmes d'apprentissage "réels". Il importe de considérer également les travaux sur l'intégration de la mémoire basée sur la généralisation et l'apprentissage basé sur l'explication ([Pazzani, 90], [Kodratoff et Tecuci, 87], [Clark, 90], [Fisher et al., 91]).
- Effectuer une implantation parallèle du modèle de la mémoire avec le langage Synergy. Un nœud de la mémoire (définition, instance ou schéma) serait actif, capable de recevoir une donnée (une nouvelle information), de se comparer avec elle et de décider de la suite du traitement selon le résultat de la comparaison (par exemple, envoyer l'information à ses nœuds fils si elle est plus spécifique que lui).
- Un axe de recherche très important consiste à intégrer "fortement" le modèle de la mémoire, le langage Synergy et le langage Prolog+CG. Cette intégration permettrait une *programmation orientée mémoire* (le travail de Hammond et al. [Hammond et al., 90] en est un exemple). L'intégration d'une nouvelle connaissance en (ou la recherche d'une connaissance de la) mémoire peut s'accompagner d'un traitement (activation de graphes ou activation de règles) qui pourrait avoir un impact sur le processus même d'intégration (ou de recherche). Aussi, cette intégration permettrait de mieux formuler les processus cognitifs ainsi que leur rapport avec la mémoire et entre eux (via la mémoire ?). Cette programmation orientée mémoire permettrait la réalisation de systèmes multi-agents de plus en plus "humains" .... *ce rêve qui est toujours, pour demain.*

En conclusion, il y a du travail pour toute une vie et pour toute une communauté !

Conclusion

La représentation et la manipulation des connaissances sont au centre de plusieurs domaines de l'informatique. Face à la diversité des types de connaissances, différents formalismes ont été proposés dans la littérature. La multiplicité des formalismes se trouve non seulement au sein d'un domaine (par exemple, les différents langages fonctionnels dans le domaine "programmation fonctionnelle", les différents formalismes de réseaux sémantiques dans le domaine de "représentation des connaissances" ou les langages agents dans le domaine "programmation par agents"), mais aussi au sein d'un système complexe développé dans le cadre d'un domaine particulier (par exemple, un système tutoriel intelligent ou un système multi-agent). Pour éviter, d'une part, l'apprentissage et l'utilisation de plusieurs formalismes et, d'autre part, les difficultés d'intégration, de communication et de partage des connaissances, il est donc plus simple d'utiliser un seul formalisme.

Nous avons proposé dans cette thèse un système multi-paradigme, basé sur un formalisme uniforme, la théorie des Graphes Conceptuels. Cette dernière a été définie par John Sowa comme un formalisme universel de représentation des connaissances. Elle est présentée aussi comme un fondement logique pour les réseaux sémantiques et comme une théorie du traitement de l'information chez l'humain et la machine.

Suite à une étude des systèmes qui ont été développés pour manipuler les Graphes Conceptuels (GC) et en tenant compte des "besoins" de certains domaines concernant les types de connaissances et leur manipulation, nous avons constaté l'absence de quatre éléments importants : 1) une algèbre pour les GC composés avec liens de co-références, 2) un "Prolog pour les GC", 3) un mécanisme d'activation des GC qui soit assez général et puissant pour tenir compte de différents modèles de programmation et, enfin, 4) l'absence d'un modèle de formation incrémentale d'une base de connaissances selon la théorie des GC.

Le système multi-paradigme que nous avons décrit dans cette thèse incorpore quatre éléments conforment à ceux mentionnés ci-dessus :

1) une algèbre pour les GC composés avec liens de co-références,

   L'algèbre offre un grand nombre d'opérations organisées selon une hiérarchie d'opérations qui montre clairement les rapports entre les opérations,

2) un "Prolog pour les GC" intégrant une extension contextuelle et orientée objet, ceci, sans masquer le langage Prolog,
3) un langage graphique, parallèle et multi-paradigme basé sur l'activation des GC,
   Le langage, appelé Synergy, intègre divers modèles de programmation, en particulier : le modèle procédural, le modèle fonctionnel et le modèle orienté-objet,
4) un modèle de formation incrémentale d'une base de connaissances selon la théorie des GC et en particulier, selon l'organisation de la base d'une application Synergy, assurant ainsi l'intégration de ces deux composantes du système.

Pour chacun des éléments de notre système, nous avons fourni une définition détaillée et des exemples d'utilisation. Nous avons discuté ensuite des applications, originalités, limites et développements futurs pour les quatre éléments. Nous avons considéré également différentes intégrations des éléments (en particulier le langage Synergy et le modèle de la mémoire).
Bien que chaque composante de notre système peut être utilisée séparément, elles peuvent être toutes fortement intégrées en un seul "outil". Ce dernier peut être utilisé, entres autres, pour le développement d'un *environnement général de Systèmes Multi-Agents (SMA).* Cette éventualité constitue pour notre thèse, un axe de recherche très prometteux. En effet, un tel environnement a besoin d'un système multi-paradigme comme celui proposé dans cette thèse, surtout lorsqu'on considère des systèmes multi-agents hétérogènes comprenant des objets "statiques", des processus, des agents cognitifs et/ou réactifs chacun avec sa mémoire dynamique, ses capacités d'apprentissage, de raisonnement, de comportement et de traitement du langage naturel. De tels systèmes intègrent en eux différentes disciplines de l'informatique et de l'intelligence artificielle.
Les GC ont été utilisé pour une modélisation conceptuelle des objets (il en serait ainsi pour les objets dans un SMA) et pour le traitement du langage naturel (capacité importante d'agent). Synergy constitue un langage flexible pour spécifier et simuler différents types de comportement (une procédure "informatique", une machine, un process ou un agent) comme ceux qu'on peut rencontrer dans un SMA (hétérogène). Notre modèle de la mémoire est approprié pour la formation et l'organisation de la mémoire d'un agent et peut constituer un cadre pour l'apprentissage multi-stratégies. Enfin, le langage Prolog+CG peut gérer les inférences sur les croyances d'un agent.

Le souci de synthèse et d'exploration nous a incité à considérer les quatre "éléments" en parallèle, d'autant plus qu'ils sont complémentaires et qu'une approche globale et uniforme, aussi bien pour la conception que la réalisation, est souhaitable.

Pour chacune des composantes de notre système, nous avons proposé une solution pratique qui peut constituer elle-même un point de départ pour différents développements et

applications. De plus, nous avons proposé des combinaisons et intégrations des éléments, intégrations qui peuvent également être développées et explorées davantage. Celles-ci peuvent de plus suggérer d'autres intégrations.

Bibliographie

Ackerman W. B., Data flow languages, *Computer,* 15:2, p. 15-25, 1982.

ACM, Special Issue on Programming Language Paradigms, *ACM Comput. Surv.,* 21:3, 1989.

Agha G. and C. Hewitt, Actors : A Conceptual Foundation for Concurrent Object-Oriented Programming, pp. 49-74, dans B. Shriver B, P. Wegner (eds.): *Research Directions in Object Oriented Programming,* MIT Press, 1987.

Agha G., P. Wegner et A. Yonezawa (Eds.), *Research Directions in Concurrent Object-Oriented Programming,* The MIT Press, 1993.

Aïmeur A., et J-G Ganascia, Elicitation of taxonomies based on the use of conceptual graphs operators, dans Mineau et al. (Eds.), 1993.

Aït-Kaci H., *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures,* Ph. D. Thesis, U. of Pennsylvania, 1984.

Aït-Kaci H. and R. Nasr R. (1986), LOGIN : A logic programming language with built-in inheritance, *Journal Logic Programming,* 3, pp. 185-215.

Aït-Kaci H. and A. Podelski, Logic programming with functions over order-sorted feature terms, dans E. Lamma and P. Mello (Eds.), *Extensions of Logic Programming,* Springer-Verlag, pp. 100-119, 1992.

Aït-Kaci H., B. Dumant, R. Meyer, A. Podelski et P. van Roy, *The wild LIFE Handbook,* (disponible par WWW) 1994.

Amamiya M. et R. Hasegawa, Data flow computing and eager and lazy evaluations, *New generation Computing,* 2, p. 105-129, 1984.

America P., POOL-T: A Parallel object-oriented language, dans Yonezawa et Tokor (eds.), 1987.

America P., Formal Techniques for Parallel OO Languages, dans Tokoro et al. (Eds.), 1992.

Andrews G. R., *Concurrent Programming, Principles and Practice,* The Benjamin/Cumming, 1991.

Arvind et R. S. Nikhil, Executing a Program on the MIT Tagged-Token Dataflow Architecture, dans J. W. de Bakker, A. J. Nijman et P. C. Treleaven (eds.), *PARLE : Parallel Architectures and Languages Europe,* Volume II, Springer-Verlag, p. 1-29, 1987.

Ashcroft E. A., Data and Education: Data-Driven and Demand-Driven Distributed Computation, Lectures Notes in Computer Science, Springer-Verlag, p. 1-50, 1986.

Atzeni P. and D. Stott Parker Jr., Algorithms for containment inference, dans F. Bancilhon and P.Buneman (eds.), *Advances in database programming languages,* ACM Press, 1990.

Bail J. Le et al., Réseaux de Pétri Hybrides, *Technique et Science Informatique,* 11:5, 1992.

Bailly C., J-F. Challine, P. Y. Gloess, H-C. Ferri et B. Marchesin, *Les langages orientés objets,* Cepadues-Editions, 1987.

Balbo G., Performance Issues in Parallel Programming, p. 1-23, dans K. Jensen (ed), *Application and Theory of Petri Nets 1992,* Springer-Verlag, 1992.

Balch T., et al., Io, Ganymede and Callisto, A Multiagent Robot Trash-Collecting Team, dans *AI Magazine,* 16:2, 1995.

Baldassari M. et G. Bruno, An Environment for Object-Oriented Conceptual Programming Based on PROT Nets, dans G. Rozenberg (ed), *Advances in Petri Nets 1988,* Springer-Verlag, p. 1-19, 1988.

Banâtre J. P. and D. Le Métayer (eds.), *Research directions in high-level parallel programming languages,* Springer-Verlag, 1991.

Bareiss R., *Exemplar-Based Knowledge Acquisition, A Unified Approach to Concept Representation, Classification, and Learning,* Academic Press, 1989.

Bergadano F., A. Giordana et L. Saitta, *MACHINE LEARNING, An Integrated Framework and its Applications,* Ellis Horwood, 1991.

Bertino E. et S. Urban (Eds.), *Object-Oriented Methodologies and Systems,* Springer-Verlag, 1994.

Bic L., Processing of Semantic Nets on Dataflow Architectures, *AI,* 27, 219-227, 1985.

Bic L., Data-driven processing of semantic nets, dans Kowalik (Ed), 1988.

Bläsius K. H., Hedtstück U. and Rollinger C. -R., Eds., *Sorts and Types in Artificial Intelligence,* Lecture Notes In Artificial Intelligence, No. 418, Springer-Verlag, 1990.

Boksenbaum C., B. Carbonneill, O. Haemmerlé et T. Libourel, Conceptual graphs for relational databases, dans Mineau et al. (Eds), 1993.

Bond A. H. et L. Gasser (eds.), *Readings in Distributed Artificial Intelligence,* Morgan Kaufmann, 1988.

Bourne J. R. et al., Intelligent CAI in Engineering: Knowledge Representation Strategies to Facilitate Model-Based Reasoning, *Int. J. of Intelligent Systems,* 3, p. 213-228, 1988.

Bournaud I. et J-G Ganascia, Conceptual clustering of complex objects: A Generalization Space based approach, dans Ellis et al. (Eds.), 1995.

Bouron T., J. Ferber et F. Samuel, MAGES : A Multi-agent testbed for heterogeneous agents, dans Demazeau et Müller (eds.), 1991.

Brachman R. et H. Levesque (eds.), *Readings in knowledge representation,* Morgan Kaufmann, 1985.

Brauer W. (ed), *Net Theory and Applications,* Springer-Verlag, 1980.

Brauer W., W. Reisig et G. Rozenberg (eds.), *Petri Nets: Applications and Relationships to Other Models of Concurrency,* Springer-Verlag, 1986.

Brawer S., *Introduction to Parallel Programming,* Academic Press, 1989.

Brown D. C., B. Chandrasekaran: *Design Problem Solving : Knowledge Structures and Control Strategies,* Morgan Kaufmann, 1989.

Brown J. S., R. R. Burton et J. de Kleer, Pedagogical, natural language and knowledge engineering techniques in SOPHIE I, II et III, dans D. Sleeman et J. S. Brown (eds.), *Intelligent Tutoring Systems,* Academic Press, 1982.

Bürckert H-J et J. Müller, RATMAN: Rational Agents Testbed for Multi-agents Networks, dans Demazeau et Müller (eds.), 1991.

Campbell J. A. et M. P. D'Inverno, Knowledge Interchange Protocols, dans Demazeau et Müller (eds.), 1990.

Cardozo E., J. S. Sichman et Y. Demazeau, Using the active object model to implement Multi-Agent Systems, Proc. of the 5th Int. Conf. on Tools with AI, Boston, USA, 1993.

Carriero N., D. Gelernter and L. Zuck, Bauhaus Linda, dans Ciancarini P. (Eds), 1994.

Carpenter B., *The logic of typed feature structures,* Cambridge Univ. Press, 1992.

Cercone N., R. Goebel, J. de Haan et S. Schaeffer, The ECO Family, dans Lehmann (Ed), p. 95-132, 1992.

Champesme M., Using empirical subsumption to reduce the search space in learning, dans Ellis et al. (Eds.), 1995.

Chan M. C., B. J. Garner et E. Tsui, Recursive modal unification for reasoning with knowledge using a graph representation, *Knowledge-Based Syst.,* 1:2, pp. 94-104, 1988.

Cheikes B. A., GIA: An Agent-Based Architecture for ITS, (disponible par WWW), 1995.

Chen I-M. A. and R. -C. Lee, An approach to deriving object hierarchies from database schema and contents, dans Z. W. Ras et M. Zemankova (eds.), *Mothodologies for intelligent systems,* Springer-Verlag, 1991.

Chosh B. C. and V. Wuwongse, Declarative Semantics of Conceptual Graph Programs, dans R. Levinson et G. Ellis (eds), Proc. of the 2 Inter. Workshop on PEIRCE, 1993.

Christensen S. et N. D. Hansen, Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs, dans M. A. Marsan (ed), *Application and Theory of Petri Nets,* Springer-Verlag, p. 186-205, 1993.

Ciancarini P., Parallel logic Programming using the Linda model of computation, dans Banâtre et Le Metayer (Eds), 1991.

Ciancarini P., O. Nierstrasz et A. Yonezawa (Eds.), *Object-Based Models and Languages for Concurrent Systems,* Springer 1994.

Ciancarini P., K. K. Jensen et D. Yankelevish, On the operational semantics of a Coordination language, dans Ciancarini P. (Eds), 1994.

Clancey W. J., *Knowledge-Based Tutoring : The GUIDON Program,* MIT Press, 1987.

Clark A., Being there : why implementation matters to cognitive science, *Artificial Intelligence Review,* 1, p. 231-244, 1987.

Clark P., Machine learning : Techniques and recent developments, dans A. R. Mirzai (ed), *Artificial intelligence : concepts and applications in engineering,* MIT Press, 1990.

Coad P. et E. Yourdon, *Object Oriented Analysis,* Yourdon Press, Prentice-Hall, 2nd Ed., 1991.

Cohen P. R., J. Morgan et M. E. Pollack (eds.), *Intentions in Communication,* MIT Press, 1990.

Coleman D. et al., Introducing ObjectCharts or How to Use StateCharts in Object-Oriented Design, *IEEE Transaction on Software Ingineering,* 18:1, 1992.

Cogis O. et O. Guinaldo, A linear descriptor for conceptual graphs and a class for polynomial isomorphism test, dans Ellis et al. (Eds), p. 263-277, 1995.

Coombs M. J. et R. T. Hartley, The MGR algorithm and its application to the generation of explanations for novel events, *Int. J. Man-Machine Studies,* 27, p. 679-708, 1987.

Connah D. et P. Wavish, An Experiment in Cooperation, dans Demazeau et Müller (eds.), 1990.

Corbett A. T. et J. R. Anderson, LISP Intelligent Tutoring System: Research in Skill Acquisition, dans Larkin J. H. et R. W. Chabay (eds.), 1992.

Cordingley E. S., Knowledge elicitation techniques for knowledge-based systems, dans Diaper (Ed), 1989.

Cunningham S. et R. J. Hubbold (eds.), *Interactive Learning Through Visualization,* Springer-Verlag, 1992.

Cyre W., Integrating Knowledge in Digital Systems Specifications using Conceptual Graphs, dans Proc. of the Sixth Annual Workshop on Conceptual Structures, 1991.

Danyluk A. P., GEMINI: An integration of analytical and empirical learning, dans Michalski R. et G. Tecuci (eds.), 1994.

Davis A. L. and R. M. Keller, Data Flow Program Graphs, dans Computer, Feb. 1982 et réimprimé dans Thakkar (ed), 1987.

Delugach H. S., Dynamic Assertion and Retraction of Conceptual Graphs, Workshop on Conceptual Structures, 1991.

Delugach H. S., *A Multiple-Viewed Approach to Software Requirements,* Ph. D. Thesis, Computer Science Dept., Univ. of Virginia, 1991.

Delugach H. S., Analysing Multiple Views of Software Requirements, dans Nagle et al. (Eds.), 1992.

Delugach H. S. et T. H. Hinke, AERIE: Database inference modeling and detection using conceptual graphs, dans Pfeiffer et Nagle (Eds.), 1992.

Demazeau Y. et J. -P. Müller (eds.), *Decentralized AI,* North-Holland, 1990a.

Demazeau Y. et J. -P. Müller, Decentralized artificial intelligence, dans Demazeau Y. et J.-P. Müller (eds.), 1990b.

Demazeau Y. et J. -P. Müller (eds.), *Decentralized AI : II,* North-Holland, 1991a.

Demazeau Y. et J. -P. Müller, From reactive to intentional Agents, Demazeau et Müller (eds.), 1991b.

Denti E., E. Lamma, P. Mello, A. Natali and A. Omicini, Techniques for implementing Contexts in Logic Programming, dans E. Lamma and P. Mello (Eds.), *Extensions of Logic Programming,* Springer-Verlag, pp. 100-119, 1992.

Diaper D. (ed), *Knowledge elicitation,* Ellis Horwood, 1989.

Dichev C., Distributed knowledge and data processing, dans ICO'93 Proceeding, pp. 272-282, 1993.

Dichev C., Logic programming with worlds, dans : *Artificial Intelligence : Methodology, Systems, Applications,* North-Holland, 1992, pp. 57-67.

Djamen J-Y, M. Kaltenbach and C. Frasson, The interactive planning with PIF, dans Frasson et al. (Eds.), 1992.

Doran J., Distributed AI and its Applications, in *Advanced Topics in AI,* p. 368-372, Springer-Verlag, 1992.

Ebert J, et G. Engels, Structural and Behavioral Views on OMT-Classes, in Bertino E. et S. Urban (Eds.), 1994.

Ellis G., Sorting Conceptual Graphs, in Nagle et al. (eds.), 1992.

Ellis G., Compiled hierarchical retrieval, in Nagle et al. (Eds.), 1992.

Ellis G., *PEIRCE User Manual,* 1993.

Ellis G. et R. Levinson (Eds.), *The Fourth PEIRCE workshop,* 1994.

Ellis G., Efficient Retrieval from Hierarchies of Objects using Lattice Operations, in Mineau et al. (Eds), 1993.

Ellis G., Object-Oriented Conceptual Graphs, in Ellis et al. (Eds.), 1995.

Ellis G. et R. Levinson, The birth of PEIRCE: A conceptual graphs workbench, in Pfeiffer et Nagle (Eds.), 1992.

Ellis G. et R. Levinson (eds), Proc. of the 1 Inter. Workshop on PEIRCE: A Conceptual Graphs Workbench, 1993.

Ellis G., R. Levinson, W. Rich et J. F. Sowa (Eds.), *Conceptual Structures: Applications, Implementation and Theory,* Proc. of the Third Intern. Conf. on Conceptual Structures, ICCS'95, Santa Cruz, CA, USA, 1995, Springer-Verlag.

Emond B., operating on conceptual structure and Peirce's system of existential graphs, in Mineau et al. (Eds.), 1993.

Engelmore B. et T. Morgan (eds.), *Blackboard Systems,* Addison-Wesley, 1988.

Esch J. et R. Levinson, An implementation model for contexts and negation in conceptual graphs, in Ellis et al. (Eds.), p. 247-262, 1995.

Evans J. B., The Devnet: a Petri Net for Discrete Event Simulation, p. 91-125, in G. Rozenberg (ed.), *Advances in Petri Nets 1993,* Springer-Verlag, 1993.

Fahlman S. E., Parallel processing in Artificial Intelligence, in Kowalik (Ed), Parallel Computation ans Computers for Artificial Intelligence, Kluwer Academic Press, 1988.

Fall A., Spanning Tree representations of graphs and orders in conceptual structures, in Ellis et al. (Eds.), p. 232-247, 1995.

Fan C. -C. , L. C-Y Cheung, T. Holden, Building Knowledge Acquisition Tools for Dynamic Decision Making, *In EXPERSYS-92.* 1992.

Fargues J., Landau M-C, Duguord A. and Catach L., Conceptual graphs for semantics and knowledge processing, *IBM Journal of Research and Development,* v. 30:1, pp. 70-79, 1986.

Fargues J., CG information retrieval using linear resolution, generalization and graph splitting, in the 4 Int. Workshop on CGS, 1989 (voir aussi chapitre de Fargues dans Nagle et al. (eds.), 1992).

Feigenbaum E. A., The simulation of verbal learning behavior, in Feigenbaum E. A. and J. Feldman (Eds.), *Computers and Thought,* McGraw-Hill, 1963.

Feigenbaum E. A. et H. Simon, EPAM-like models of recognition and learning, *Cognitive Science,* 8, p. 305-336, 1984.

Feller A. et R. Rucker, Meta-Modeling Systems Analysis Primitives, in Nagle et al. (eds.), 1992).

Ferber J. and P. Volle, Introduction to an intensional theory of object knowledge representation, in E. Chouraqui (ed), *Modélisation de la connaissance et du raisonnement,* Compte-rendu des Journées d'études des 8 et 9 Février 1988.

Ferguson I. A., Integrated Control and Coordinated Behavior: a Case for Agent Models, in in Wooldridge M. J. et N. R. Jennings (Eds.), 1995.

Fernandez-castro I., A. Diaz-ilarraza et F. Verdejo, Architctural and Planning Issues in ITS, *J. of AI in Education,* 4:4, 1993.

Fields C. A., H. D. Pfeiffer et T. C. Eskridge, Knowledge representation and control in gm1, an automated DNA sequence analysis system based on the MGR architecture, *Intern. J. of Man-Machine Studies,* 34, p. 549-573, 1991.

Fikes R. E. et N. J. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, *Artificial Intelligence,* 2:3-4, p. 189-208, 1971.

Findler N. V. (ed), *Associative Networks : representation and use of knowledge by computers,* Academic Press, 1979.

Fisher D. H., Knowledge acquisition via incremental conceptual clustering, *Machine learning,* v. 2, pp. 139-172, 1987.

Fisher D. H., M. J. Pazzani et P. Langley (eds.), *Concept formation : knowledge and experience in unsupervised learning,* Morgan Kaufmann, 1991.

Frasson C., G. Gauthier et G. I. McCalla (eds.), *Intelligent Tutoring Systems'92,* Springer-Verlag, 1992.

Frederiksen N., R. Glaser, A. Lesgold et M. G. Shafto (eds.), *Diagnostic Monitoring of Skill and Knowledge Acquisition,* Lawrence Erlbaum Associates, 1990.

Fukunaga K. and S. Hirose, An experience with a Prolog-based Object-Oriented Language, In OOPSLA'86 Proceedings, pp. 224-231, 1986.

Gabbay D. M. and U. Reyle, N-Prolog: an extension of prolog with hypothetical implications, *J. Logic Programming,* 2, pp. 251-284, 1985.

Garner B. J. and E. Tsui, An extendible graph processor for knowledge engineering,in J. F. Gilmore (ed), *Applications of AI 3,* 1986.

Garner B. J. and E. Tsui, General purpose inference engine for canonical graph models, *Knowledge-Based Systems,* 1:5, pp. 266-278, 1988.

Garner B. J., E. T. Tsui, D. Lui, D. Lukose and J. Koh, Extendible Graph Processing in Knowledge Acquisition, Planning and Reasoning, in Nagle et al. (eds.), 1992.

Gasser L. et M Huhns (eds.), *Distributed Artificial Intelligence II,* Morgan Kaufmann, 1989.

Gasser L., C. Braganza et N. Herman, MACE: A Flexible Testbed for Distributed Artificial Intelligence, dans Huhns (ed), 1987.

Gaudiot J. L., Dataflow Machines, in Milutinovic et M. Flynn (Eds.), High-Level Language Computer Architecture, Computer Science Press, 1989.

Gehani N. H. et al., Event Specification in an Active Object-Oriented Database, ACM SIGMOD, *International Conference on Management of Data,* 21, Juin 1992.

Genesereth M. R. et S. P. Ketchpel, Software Agents, ACM, 1994.(disponible par WWW).

Gennari J. H., P. Langley and D. Fisher, Models of Incremental Concept Formation, *Artificial Intelligence,* v. 40:1, pp 11-61, 1989.

Genrich H. J., Predicate/Transition nets, dans W. Brauer, W. Reisig et G. Rozenberg (eds.), *Petri Nets: Applications and Relationships to Other Models of Concurrency,* Springer-Verlag, 1986.

Gentner D., Structure Mapping : A Theoretical Framework for Analogy, *Cognitive Science,* 7, p. 155-170, 1983.

Ghosh et V. Wuwongse, Declarative Semantics for CCG, in Levinson et Ellis (eds.), 1993.

Ghosh et V. Wuwongse, Inference systems for conceptual graph programs, in Tepfenhart et al. (Eds.), 1994.

Ghosh et V. Wuwongse, A direct proof procedure for definite conceptual graph programs, in Ellis et al. (Eds.), 1995.

Gick M. and Holyoak K., Schema Induction and Analogical Transfer, *Cognitive Psychology,* 15, p. 1-38, 1983.

Giovanni R. Di , HOOD Nets, dans G. Rozenberg (ed), *Advances in Petri Nets,* Springer-Verlag, p. 140-160, 1991.

Godin R., E. Saunders and J. Gecsei, Lattice model of browsable data spaces, *Information Sciences,* v. 40, pp. 89-116, 1986.

Godin R., G. Mineau et R. Missaoui, Incremental Structuring of Knowledge Bases, *in Proc. of the Intern. KRUSE Symposium: Knowledge Retrieval, Use, and Storage for Efficiency,* Santa Cruz, CA, USA, p. 179-192, Aug. 1995.

Guerraoui R., Les langages concurrents à objets, *Technique et science informatique,* 14:8, p. 945-971, 1995.

Haemmerlé O., implementation of Multi-Agent Systems using conceptual graphs for knowledge and message representation: The CoGITo Platform, *in the Proc. supplement of ICCS'95* ,1995.

Hall R. P., Computational Approaches to Analogical Reasoning: A Comparative Analysis, *Artificial Intelligence,* 1989.

Hammond K., T. Converse and C. Martin, Integrating Planning and Acting in a Case-Based Framework, *AAAI'90,* p. 292-297, 1990.

Hanson S. J. et M. Bauer, Conceptual clustering, categorization, and polymorphy, *Mach. Learning* 3, p. 343-372, 1989.

Harel D., Statemate : a Working Environment for the Development of Complex reactive systems, *IEEE Transaction on Software Ingineering,* 16:4, 1990.

Harvey N. et J. Morris, Parallel Programming Languages: The Visual Advantage, *Aust. Comp. Sci Comms.*, Feb. 1995.

Heaton J. E. et P. Kocura, Presenting a Peirce logic based inference engine and theorem prover for conceptual graphs, in Mineau et al. (Eds.), 1993.

Hee K. M., P. M. P. Rambags et P. A. C. Verkoulen, Specification and Simulation with ExSpect, in Lauer (Ed), *Functional Programming, Concurrency, Simulation and Automated Reasoning,* Springer-Verlag, 1993.

Hendler J. A., Massively-parellel marker-passing, in Lehmann (Ed), p. 277-292, 1992.

Hendrix G. G., Modeling Simultaneous Actions and Continuous Processes, *Artificial Intelligence,*, 4, p. 145-180, 1973.

Hendrix G. G., Encoding knowledge in partitioned networks, in Findler N. V. (Ed), p. 51-92, 1979.

Herzog O. and C. -R. Rollinger (Eds.), *Text Understanding in LILOG,* Springer-Verlag, 1991.

Hickman S. et M. Shiels, Situated Action as a Basis for Cooperation, dans Demazeau et Müller (eds.), 1991.

Hils D. D., Visual Languages and Computing Survey: Data Flow visual programming languages, *J. of Visual Languages nd Computing,* 3:1, 1992.

Hines T. R., J. C. Oh, M. A. Hines, Object-Oriented Conceptual Graphs, in *Proc. of the Fifth Annual Workshop on Conceptual Structures,* 1990.

Hines T. R., M. A. Hines, Estimating Lines-of-Code for Software Development Efforts: A Learning from Examples Approach, in *Proc. of the Fifth Annual Workshop on Conceptual Structures,* 1990.

Hollan J. D., E. L. Hutchins et L. Weitzman, STEAMER: An interactive inspectable simulation-based training system, *AI Magazine,* 5, 15-27, 1984.

Hoover S. V. et R. F. Perry, *Simulation : A Problem-Solving Approach,* Addison-Wesley, 1989.

Hoare C. A. R., *Communicating Sequential Processes,* Prentice-Hall, 1985.

Huang J., N. R. Jennings et J. Fox, An Agent Architecture for Distributed Medical Care, in Wooldridge M. J. et N. R. Jennings (Eds.), 1995.

Hudak P., Conception, Evolution and Application of Functional Programming, *ACM Comput. Surv.,* 21:3, p. 359-411, 1989.

Hui S., A. Goh et J. K. Raphel, CLOG: A Class-Based Logic Language For Object-Oriented Databases, in Nishio S. et A. Yonezawa (eds.), 1993.

Huhns M. N. (ed), *Distributed AI,* Morgan Kaufmann, 1987.

Jackson P., *Introduction to Expert Systems,* Second Edition, Addison-Wesley, 1990.

Jacobson I., *Object-Oriented Software Engineering,* Addison-Wesley, 1992.

Jagannathan V., R. Dodhiawala et L. S. Baum, *Blackboard Architectures and Applications,* Academic Press, 1989.

Janson S. et Haridi S., An Introduction to AKL, A Multi-Paradigm Programming Language, (par WWW), 1993.

Jensen K., Coloured Petri Nets, dans Brauer et al. (Eds), 1986.

Jensen K., coloured Petri Nets: A High level language for system design and analysis. *Advances in Petri Nets 1990,* Springer-Verlag, 1991.

Kabbaj A., *SMGC: un système de manipulation des graphes conceptuels,* Mémoire de M. Sc., Dept. Informatique, Université Laval, 1987, Québec, Canada.

Kabbaj A., *Intelligence Artificielle en Lisp et Prolog*, Masson, 1991a.

Kabbaj A., *Vers un système intégré de représentation et manipulation des connaissances,* Rap. Int. #866, DIRO, Université de Montréal, 56 pages, 1993a.

Kabbaj A., Un système de representation et manipulation des connaissances, *ICO'93,* 1993c.

Kabbaj A., Toward a Conceptual Actor Language, in *the first Int. Conf. on Conceptual Structures,* 1993d.

Kabbaj A., Un modèle général de traitement parallèle orienté-graphe : Le langage CAL, RenPar'6, Lyon, 1994a.

Kabbaj A. *The Prolog+CG Manual,* 1995 (disponible par ftp).

Kabbaj A., Self-Organizing Knowledge Bases: The Integration Based Approach, in *Proc.of the Intern. KRUSE Symposium: Knowledge Retrieval, Use, and Storage for Efficiency,* Santa Cruz, CA, USA, p. 64-68, Aug. 1995.

Kabbaj A., *The Synergy Manual,* à paraître.

Kabbaj A. et C. Frasson, Representation et acquisition des connaissances pour un tuteur intelligent, *ACTI'93,* Limoge, 1993a.

Kabbaj A. et C. Frasson, Un algorithme de formation dynamique de la mémoire, *ICO'93,* 1993b.

Kabbaj A. et C. Frasson, Dynamic integration of knowledge in memory, *ICCI,* Toronto, 1993c.

Kabbaj A. et C. Frasson, An incremental model of memory formation for a multi-strategy learning environment, dans *Proc. of the first Int. Conf. on Conceptual Structures,* Quebec, Canada, p. 30-44, 1993d.

Kabbaj A. et C. Frasson, Towards a tutor for knowledge representation, *Int. Conf. on Computers in Education,* Taiwan, 1993e.

Kabbaj A. et C. Frasson, Operational and Logical components of a Knowledge Manipulation System, Int. Conf. in *AI and Education ; AI-ED'93,* Edinburgh, 1993f.

Kabbaj A., C. Frasson, M. Kaltenbach et J-Y Djamen, A conceptual and contextual object-oriented logic programming : PROLOG++ language, in Tepfenhart et al. (Eds), p. 251-274, 1994

Kabbaj A. et C. Frasson, A new programming language : PROLOG++, Rap. Int. # 919, DIRO, U. de Montréal, 71 pages, 1994b.

Kabbaj A. et C. Frasson, CAL: A general-purpose parallel and symbolic language, Pub. # 924, DIRO, Univ. de Montréal, 1994.

Kabbaj A. et C. Frasson, A general model of behavior for CG Theory: SYNERGY language, Pub. #944, DIRO, Univ. de Montréal, 1995.

Kabbaj A. et C. Frasson, CAL Language: When education influences the design of an AI language, *7th World Conf. on AI in Education (AI-ED'95),* Washington, DC, Aug., 1995.

Kabbaj A. et C. Frasson, Dynamic CG: Toward a General Model of Computation, in *the Proc. Supplement of ICCS'95,* Santa Cruz, 1995.

Kabbaj A. et B. Moulin, SMGC : A tool for conceptual graphs processing, in *The Journal for the integrated study of artificial intelligence, cognitive science and applied epistemology,* 7:1, pp. 23-47, 1990.

Kaiser et al., Multiple Concurrency control policies in an OOP System, in Agha et al. (Eds.),

1993.

Karakoulas G. J., Diagnostic reasoning about an economic system : A model-based approach, in M. G. Singh and L. Travé-Massuyès (eds.), pp. 293-296, 1991.

Karamouzis T. and S. Feyock, An integration of Case-based and Model-based reasoning and its application to physical system faults, in F. Belli and F. J. Radermacher (eds.), *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems,* Springer-Verlag, 1992.

Kauffmann H. and A. Grumbach, MULTILOG : MULTIple worlds in LOGic programming, in *the proceeding of the 7th European Conference on AI,* 1986.

Kettler B. P., J. A. Hendler, W. A. Andersen et M. P. Evett, Massively Parallel Support for Case-Based Planning, IEEE Expert, February 1994.

Kieras D. E., What Mental Model Should Be Taught: Choosing Instructional Content for Complex Engineered Systems, in Psotka and al. (eds.), 1988.

Kieras D. E., The Role of Cognitive Simulation Models in the Development of Advanced Training and Testing Systems, dans N. Frederiksen, R. Glaser, A. Lesgold et M. G. Shafto (eds.), 1990.

Kluge W., Reduction, Data Flow and Control Flow Models of Computation, p. 466-498, in Brauer et al. (Eds), 1986.

Kocura P. and K. Kwong Ho, Aspects of Conceptual Graphs Processor Design, in the 7 Int. Workshop on CGS, 1992.

Kodratoff Y. and G. Tecuci, The central Role of Explanations in DISCIPLE, In K. Morik (ed.), *Knowledge Representation and Organization in Machine Learning,* Springer-Verlag, 1987.

Kodratoff and R. Michalski (eds), *Machine learning,* volume 3, Morgan Kaufmann, 1990.

Kolodner J. L., *Retrieval and organizational strategies in conceptual memory : a computer model,* Lawrence Erlbaum, 1984.

Kolodner J. L., Case-Based Reasoning, in Shapiro S. C. (ed), *The Encyclopedia of Artificial Intelligence,* Wiley, 1265-1279, 1992.

Kolodner J. L., *Case-Based Reasoning,* Morgan Kaufmann, 1993.

Kolodner J. L. and C. K. Riesbeck, *Experience, Memory and Reasoning,* Lawrence Erlbaum, 1986.

Kusakabe S. et M. Amamiya, A Dataflow-Based Massively Parallel Programming Language "V" and its implementation on a Stock Parallel Machine, in Ito T. et A. Yonezawa (Eds.), *Theory and Practice of Parallel Programming,* p. 457-471, Springer, 1994.

Lakos C., From Coloured Petri Nets to Object Petri Nets, in Michelis G. et M. Diaz (Eds.), *Application and Theory of Petri Nets*, Springer, 1995.

Lanusse A. and F. Terrier, Representation orientée "Acteur" de connaissances temporelles : vers des systèmes industriels temp réel à bases de connaissances, *Avignon'92,* pp. 295-307, 1992.

Larkin J. H. et R. W. Chabay (eds.), *Computer-Assisted instruction and intelligent tutoring systems: Shared goals and complementary approaches,* Lawrence Erlbaum Associates, 1992.

Lebowitz M., generalization from Natural Language Text, *Cognitive Science,* 7:1, p. 1-40, 1983.

Lebowitz M., Concept learning in a rich input domain: Generalization-based memory, dans Michalski et al. (eds.), 1986.

Lebowitz M., The utility of similarity-based learning in a world needing explanation, In Y. Kodratoff and R. Michalski (eds), 1990.

Lehmann F., Semantic networks, *International Journal computers & mathematics with applications,* 23:2-9, p. 1-50, 1992a.

Lehmann F. (ed), Special Issue on Semantic networks in artificial intelligence, in *an International Journal computers & mathematics with applications,* 23:2-9, 1992b.

Leishman D., An analogical tool: Based on Evaluation of partial correspondances over conceptual graphs, in Nagle et al. (Eds.), p. 349-360, 1992.

Lesgold A., S. Lajoie, M. Bunzo et G. Eggan, SHERLOCK: A Coached Practice Environment for an Electronics Troubleshooting Job, dans J. H. Larkin et R. W. Chabay (eds.), 1992.

Levinson R., A Self-organizing retrieval system for graphs, in *Proc. AAAI-84*, p. 203-206, 1984.

Levinson R., Pattern associativity and the retrieval of semantic networks, in Lehmann F. (ed), 1992.

Levinson R., Towards domain-independent machine intelligence, in Mineau et al. (Eds.), 1993.

Levinson R., LEARN: exploiting the physics of state-space search with MorphII, in Ellis et Levinson (Eds.), Third PEIRCE Workshop, 1994.

Levinson R. and G. Ellis (eds), Proc. of the 2 Inter. Workshop on PEIRCE: A Conceptual Graphs Workbench, 1993.

Lewis T. G. and H. El-Rewini, *Introduction to Parallel Computing,* Prentice Hall, 1992.

Liddle S. W., D. W. Embley et S. N. Woodfield, A Seamless Model for Object-Oriented System development, Bertino E. et S. Urban (Eds.), 1994.

Liquière M., Graphs and learning, in Proc. supplement of ICCS'93.

Liquière M. et O. Brissac, A class of conceptual graphs with polynomial iso-projection, in Tepfenhart et al. (Eds.), 1994.

Lizotte M. et B. Moulin, SAIRVO: a planning system implementing the ACTEM concept, *Knowledge Based Systems,* 2:4, p. 210-218, 1989.

Luger G. F. and W. A. Stubblefield, *AI and the Design of Expert Systems,* Benjamin/Cummins, 1993.

Lukose D., *Goal interpretation as a knowledge acquisition mechanism,* Ph. D. Thesis, Dept. of computing and Mathematics, Deakin Univ., 1992.

Lukose D., Executable conceptual structures, in Mineau et al. (Eds.), 1993.

Lukose D., T. Cross, C. Munday et F. Sobora, Operational KADS Conceptual Model using conceptual graphs and executable conceptual structures, in *Proc. supplement of ICCS'95.*

Madsen O., B. Moller-Pedersen et K. Nygaard, *Obejct-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993.

Maes P. (ed), *Designing Autonomous Agents : Theory and Practice from Biology to Engineering and Back,* MIT Press, 1991.

Malec J., Process Transition Networks : A Formal Graphical Knowledge Representation Tool, in Z. W. Ras, M. Zemankova (eds.): *Methodologies for Intelligent Systems,* Springer-Verlag, 1991.

Malhotra J. et R. M. Shapiro, Generating an Algorithm for Executing Graphical Models, dans P. E. Lauer (ed), 1993.

Maruichi T., M. Ichikawa et M. Tokoro, Modeling Autonomous agents and their groups, dans Demazeau et Müller (eds.), 1990.

Masini G., A. Napoli, D. Colnet, D. Léonard et K. Tombre, *Les langages à objets,* InterÉditions, 1990.

Matsuoka S. et A. Yonezawa, Analysis of Inheritance Anomaly in OOCP Languages, in Agha et al. (Eds.), 1993.

Mattos N. M., *An Approach to Knowledge Base Management,* Springer-Verlag, 1991.

McAffer J., Meta-Level Programming with CodA, in Olthoff (Ed), 1995.

McCabe F. G., *L&O : Logic and Objects,* Prentice-Hall, 1992.

McGregor 91, The evolution technology of classification-based knowledge representation systems, in Sowa (Ed), 1991.

McHale et al., Scheduling Predicates, in Tokoro et al. (Eds.), 1992.

McSkimin et J. Minker, A predicate calculus based semantic network for deductive searching, in Findler N. V. (Ed), p. 205-237, 1979.

Mellor S. J. et S. Shlaer, *Object Lifecycles,* Yourdon Press, 1992.

Michalski R. M., A Theory and methodology of inductive learning, in Michalski R. M. et al. (Eds.), 1983.

Michalski R. M. et R. Stepp, Learning from observation: conceptual clustering, in Michalski R. M. et al. (Eds.), 1983.

Michalski R. M., J. G. Carbonell, T. M. Mitchell (eds.), *Machine Learning, volume I and II,* Morgan Kaufmann, 1983, 1986.

Michalski R. M., Inferential Theory of Learning: Developing Foundations for Multistrategy Learning, in Michalski R. et G. Tecuci (eds.), 1994.

Michalski R. et G. Tecuci (eds.), *Machine learning : A multistrategy approach, Vol IV,* Morgan Kaufmann, 1994.

Milner, *A Calculus of Communicating Systems,* Springer-Verlag, 1980.

Milner, *Communication and Concurrency,* Prentice-Hall, 1989.

Mineau G., *Structuration des bases de connaissances par généralisation,* Thèse de Doctorat, DIRO, U. de Montréal, 1990.

Mineau G., Normalizing conceptual graphs, in Nagle et al., (Eds.), 1992.

Mineau G., B. Moulin and J. Sowa (eds.), *Conceptual Graph for Knowledge Representation,* Springer-Verlag, 1993.

Mineau G., Viewa, Mappings and Functions: Essential Definitions to the Conceptual Graph Theory, in Tepfenhart et al. (Eds.), p. 160-174, 1994.

Minsky M. (Ed), *Semantic Information Processing,* MIT Press, 1968.

Mirenkov N. N. (ed), *Parallel Computing Technologies,* World Scientific, 1991.

Miriyala S., G. Agha et Y. Sami, Visualizing Actor Programs Using Predicate Transition Nets, *J. of Visual Languages and Computing,* p. 195-220, 1992.

Mitchell T. M., Generalization as search, *Artificial Intelligence,* 18, p. 203-226, 1982.

Monteiro L. and A. Porto, Contextual Logic Programming, in G. Levi and M. Martelli (Eds.), *Proc. 6th Int. Conf. and Symposium on Logic Programming,* The MIT Press, 1989.

Moreira A. M. D. et R. G. Clark, Rigorous Object-Oriented Analysis, in Bertino E. et S. Urban (Eds.), 1994.

Moss C., *Prolog++ : The Power of Object-Oriented and Logic Programming,* Addison-Wesley, 1994.

Moulin B., D. Côté et A. Kabbaj, Le système de manipulation de graphes conceptuels SMGC, *ICO,* 2:4, p. 103-115, 1990.

Mugnier M. L. et M. Chein, Specialization: where do the difficulties occur ?, in Pfeiffer et Nagle (Eds.), p. 229-238, 1992.

Mugnier M. L. et M. Chein, Polynomial algorithms for projection and matching, in Pfeiffer et Nagle (Eds.), p. 229-238, 1992.

Mugnier M. L. et M. Chein, Characterization and algorithmic recognition of canonical conceptual graphs, in Mineau et al. (Eds), p. 294-311, 1993.

Munday et al., *CGKEE : User and System Manual,* 1995.

Murray K. S., Learning as Knowledge Integration: A Case Study, *AAAI Symposium on Knowledge Assimilation,* 1990.

Myaeng S. H. and A. Lopez-Lopez, A Flexible Algorithm for Matching Conceptual Graphs in the 6 Int. Workshop on CGS, 1991 (voir aussi Nagle et al. (eds.), 1992).

Mylopoulos J., The PSN Tribe, in Lehmann (Ed), p. 223-242, 1992.

Mylopoulos J., H. Wang et B. Kramer, Knowbel: A Hybrid Tool for building Expert Systems, *IEEE Expert,* February 1993.

Nagle T. E., J. W. Esch et G. Mineau, A Notation for Conceptual Structure Graph Matchers, in the 5 Int. Workshop on CGS, 1990 (voir aussi dans Nagle et al. (eds.), 1992).

Nagle T. E., J. A. Nagle, L. L. Gerholz and P. W. Eklund, *Conceptual Structures : Current research and practice,* Ellis Horwood, 1992.

Nebel B., *Reasoning and Revision in Hybrid Representation Systems,* Springer-Verlag, 1990.

Nishio S. et A. Yonezawa (Eds.), *Object Tehnologies for Advanced Software,* Springer-Verlag, 1993.

Norman D. A. et D. E. Rumelhart et le groupe LNR, *Explorations in Cognition,* W. H. Freeman and Company, 1975.

Olthoff W. (Ed.), *ECOOP'95 - Object-Oriented Programming,* Springer, 1995.

Papathomas M., A Unifying Framework for Process Calculus Semantics of COO Languages, in Tokoro et al. (Eds), 1992.

Pazzani M., Creating High Level Knowledge Structures from Simple Elements, In K. Morik (ed.), *Knowledge Representation and Organization in Machine Learning,* Springer-Verlag, 1987.

Pazzani M., *Creating of Memory of Causal Relationships,* Lawrence Erlbaum Ass., 1990.

Peterson J. L., A Note on Colored Petri Nets, *Information Processing Letters,* 11:1, p. 40-43, 1980.

Peterson J. L., *Petri net theory and the modeling of systems*, Prentice-Hall, 1981.

Pfeiffer H. D. and R. T. Hartley, Semantic Additions to Conceptual Programming, in Workshop on Conceptual Structures, 1989.

Pfeiffer H. D. and R. T. Hartley, The Conceptual Programming Environment, CP, in Nagle et al. (Eds.), 1992.

Pfeiffer H. D. and T. E. Nagle (eds.), *Conceptual Structures : Theory and implementation,* Springer-Verlag, 1992.

Pfeiffer H. D. et B. J. Waltar, Automated Message Analysis Using the Conceptual Programming Environment, CP, in *the Proc. supplement of ICCS'95*, 1995.

Pletat U. and K. von Luck, Knowledge Representation in LILOG, in Bläsius et al. (eds), 1990.

Piaget J., *Mémoire et intelligence,* Presses Universitaires de France, Paris 1968.

Pidd M., *Computer Simulation in Management Science,* Sec. Ed., John Wiley & Sons, 1988.

Poggi A., DAISY: An Object-Oriented System for DAI, in Wooldridge M. J. et N. R. Jennings (Eds.), 1995.

Pratt T. W., *Programming languages, Design and Implementation,* 2nd Ed., Prentice-Hall, 1984.

Pressman R. S., *Software Engineering : A Practitioner's Approach,* 3th ed., McGraw-Hill, 1992.

Psotka J., L. Dan Massey et S. A. Mutter (Eds.), *Intelligent Tutoring Systems, Lessons Learned,* Lawrence Erlbaum Ass. Pub., 1988.

Quinlan, Induction of decision trees, *Mach. Learning,* 1, p. 81-106, 1986.

Rao A. S. and N. Y. Foo, CONGRES: Conceptual Graph Reasoning System, *Proc. IEEE,* 1987.

Reddy K. C., et al., XLAR - Cognitive Architecture for Intelligent Action, *Expert Systems with Applications,* 4, p. 129-140, 1992.

Reiser B. J., P. Friedmann, J. Gevins, D. Y. Kimberg, M. Ranney et A. Romero, A graphical programming language interface for an intelligent tutor, In E. Soloway, D. Frye et S. B. Sheppard (eds.), *Human Factors in Computing Systems,* CHI'88 Conf. Proc., ACM Press, 1988.

Rieger C., An organization of Knowledge for problem solving and language comprehension, *Artificial Intelligence,* 7:2, 1976.

Rieger C. and M. Grinberg, The declarative representation and procedural simulation of causality in physical mechanisms, *IJCAI-77,* p. 250-256, 1977.

Riesbeck C. K. and R. C. Schank, *Inside case-based reasoning,* Lawrence Erlbaum associates, 1989.

Roberts D. D., *The Existential Graphs of Charles S. Peirce,* Mouton, The Hague, 1973.

Roussopoulos N. et J. Mylopoulos, Using semantic networks for database management, *Proc. First Intern. VLDB Conf.,* ACM, 1975.

Rumbaugh et al., *Object-Oriented Modelling and Design,* Prentice-Hall, 1991.

Rumelhart D. E., P. H. Lindsay et D. A. Norman, A Process Model for Long-Term Memory, in Tulving E. et W. Donaldson (Eds.), *Organization of Memory,* Academic Press, p. 197-246, 1972.

Rumelhart D. E. et D. A. Norman, Accretion, tuning, and restructuring: three modes of learning, in Cotton J. W. et R. L. Klatzky (Eds.), *Semantic factors in cognition,* Lawrence Erlbaum Ass., 1978.

Rumelhart D. E., McClelland et PDP Research Group, *Parallel Distributed Processing,* MIT Press, 1986.

Russell S. J., *Analogical and Inductive Reasoning,* PhD thesis, Stanford Univ., 1987.

Sack W et E. Soloway, From PROUST to CHIRON: ITS Design as Iterative Engineering; Intermediate Results are Important !, dans Larkin J. H. et R. W. Chabay (eds.), 1992.

Saleh H. et P. Gautron, A Concurrency Control Mechanism for C++ Objects, Nishio S. et A. Yonezawa (Eds.), 1993.

Salton G. and M. J. McGill, *Introduction to Modern Information Retrieval,* McGraw-Hill, 1983.

Sargeant J., Uniting Functional and Object-Oriented Programming, in Nishio S. et A. Yonezawa (Eds.), 1993.

Schank R. C., *Dynamic Memory : A theory of Learning in Computers and People,* Cambridge University Press, 1982.

Schank R. C., *Explanations Patterns, Understanding Mechanically and Creatively,* Lawrence Erlbaum Associates, 1986.

Schank R. C., *The Creative Attitude, Learning to Ask and Answer the Right Questions,* MacMillan, 1988.

Schank R. C., *Tell me a story : A new look at real and artificial memory,* MacMillan, 1991.

Schank R. C. et K. M. Colby (Eds.), *Computer Models of thought and Languages,* W. H. Freeman, San Francisco, CA, 1973.

Schank R. C. and R. P. Abelson, Scripts, Plans and Knowledge, *IJCAI-75,* p. 151-157, 1975.

Schank R. C. et D. B. Leake, Creativity and Learning in a Case-Based Explainer, *Artificial Intelligence,* 40, p. 353-385, 1989.

Schmolze J. G. et T. A. Lipkis, Classification in the KL-ONE knowledge representation system, in *IJCAI-83,* 1983.

Schreiner W., The ADAM Abstract Dataflow Machine, in B. Fronhöfer et G. Wrighton (Eds.), *Parallelization in Inference Systems,* Springer, 1992.

Schruben L., Simulation Modeling with Event Graphs, *Comm. ACM,* 26:11, p. 957-963, 1983.

Schubert L. K., R. G. Goebel and N. J. Cercone, The structure and organization of a semantic net for comprehension and inference, in Findler N. V. (Ed), p. 122-172, 1979.

Scragg G. W., Answering process questions, *IJCAI-75,* p. 435-442, 1975.

Shapiro S. C., The SNePS semantic network processing system, dans Findler N. V. (Ed), p. 179-203, 1979.

Shapiro S. C. et W. J. Rapaport, The SNePS Family, in Lehmann (Ed), p. 243-275, 1992.

Shapiro E., The family of Concurrent Logic Programming Languages, *ACM Comput. Surv.,* 21:3, p. 412-510, 1989.

Shastri L., Structured connectionist models of semantic networks, in Lehmann (Ed), p. 293-328, 1992.

Shibayama E., Semantic Layers of Object-Based Concurrent Computing, in Tokoro et al. (Eds.), 1992.

Shlaer S. et S. J. Mellor, *Object Lifecycles - Modeling the World in States,* Prentice-Hall, 1992.

Shoham Y., Agent-oriented programming, *Artificial intelligence,* 60, p. 51-92, 1993.

Simmons R. et D. Chester, Inferences in quantified semantic networks, *IJCAI-77,* p. 267-273.

Singh M. P., Group Ability and Structure, dans Demazeau et Müller (eds.), 1991.

Singh M. G. and L. Travé-Massuyès (eds.), *Decision Support Systems and Qualitative Reasoning,* North-Holland, 1991.

Skuce D. et T. C. Lethbridge, CODE4: a unified system for managing conceptual knowledge, *Int. J. Human-Computers Studies,* 42, p. 413-451, 1995.

Slade S., Case-Based Reasoning: A Research Paradigm, *AI Magazine,* 12:1, p. 42-55, 1991.

Smolka G., *An Oz Primer,* (par WWW), 1995.

Sowa J. F., Conceptual graphs for a data base interface, *IBM J. of Research and Development,* 20:4, p. 336-357, 1976.

Sowa J. F., *Conceptual Structures : Information Processing in Mind and Machine,* Addison-Wesley, 1984.

Sowa J. F., Conceptual Graphs as a universal knowledge representation, in Lehmann (ed), 1992a.

Sowa J. F. (Ed), *Principles of Semantic Networks: Explorations in the Representation of Knowledge,* Morgan Kaufmann, San Mateo, CA, 1991.

Sowa J. F., Relating Diagrams to Logic, dans Mineau et al. (Eds), 1993a.

Sowa J. F., Logical foundations for representing object-oriented systems, *J. of Experimental and Theoretical AI,* 5, 1993.

Sowa J. F., Syntax, Semantics, and Pragmatics of Contexts, in Ellis et al. (Eds.), p. 1-15, 1995.

Sowa J. F. et E. C. Way, Implementing a semantic interpreter using conceptual graphs, *IBM J. Res. Develop.,* 30:1, p. 57-69, 1986.

Spinner M. P., *Improving Project Management Skills and Techniques,* Prentice Hall,1989.

Staniford G. et R. Paton, Simulating Animal Societies with Adaptive Communicating Agents, in Wooldridge M. J. et N. R. Jennings (Eds.), 1995.

Suchman L., *Plans and Situated Action*, Cambridge Univ. Press, 1987.

Szymankiewicz J., J. McDonald et K. Turner, *Solving business problems by simulation,* Sec. Ed., McGraw-Hill, 1988.

Tecuci G. et R. S. Michalski, input understanding as a basis for multistrategy task-adaptive

learning, p. 419-428, in Z. W. Ras et M. Zemankova (eds.), *Methodologies for intelligent systems,* Springer-Verlag, 1991.

Tecuci G., An Inference-based Framework for multi-strategy learning, in Michalski et Tecuci (Eds), 1995.

Tecuci G., Building knowledge bases through multistrategy learning and knowledge acquisition, in Tecuci G. et Y. Kodratoff (eds.), *Machine Learning and Knowledge Acquisition, Integrated Approaches,* Academic Press, 1995.

Tepfenhart W. M., J. P. Dick et J. F. Sowa (Eds.), *Conceptual Structures,* Proc. of the second intern. conf. on conceptual structures, ICCS'94, 1994.

Thakkar S. S. (ed.), *Selected Reprints on Dataflow and Reduction Architectures,* IEEE Computer Society Press, 1987.

Tiberghien A. et H. Mandl (eds.), *Intelligent Learning Environments and Knowledge Acquisition in Physics,* Springer-Verlag, 1992.

Tokoro M., O. Nierstrasz et P. Wegner (eds.), *Object Based Concurrent Computing,* Springer-Verlag, 1992.

Törn A. A., Simulation Graphs: A General Tool for Modeling Simulation Designs, *Simulation,* 37:6, p. 187-194, 1981.

Törn A. A., Simulation Nets, a Simulation Modeling and Validation Tool, Simulation, 45:2, p. 71-75, 1985.

Törn A. A., Systems Modelling and Analysis Using Simulation Nets, dans C. A. Kulikowski et al. (eds.), *Artificial Intelligence and Expert Systems Languages in Modelling and Simulation,* Elsevier Science North-Holland, p. 283-288, 1988.

Thomas S. R., The PLACA AgentProgramming Language, in Wooldridge M. J. et N. R. Jennings (Eds.), 1995.

Towne D. M. et A. Munro, The Intelligent Maintenance Training System, dans Psotka et al. (eds), 1988.

Trappl R., AI: Introduction, Paradigms, Applications, in *Advanced Topics in AI,* 1992.

Treleaven P. C., R. P. Hopkins et P. W. Rautenbach, Combining Data Flow and Control Flow Computing, *Computer Journal,* 25:2, 1982 et réimprimé dans Thakkar (ed)., p. 355-365, 1987.

Treleaven P. C. (ed), *PARALLEL COMPUTERS : object-Oriented, Functional, Logic,* John Wiley & Sons, 1990.

Turner D. A., A new implementation technique for applicative languages, *Software: Practice and Experience,* 9:1, p. 31-49, 1979.

Turner D. A., Miranda : A non-strict functional language with polymorphic types, in J.-P. Jouannaud (ed), *Functional Programming languages and Computers Architectures,* Springer-Verlag, 1985.

Turner D. A. (ed), Research Topics in Functional Programming, Addison-Wesley, 1990.

Van Hee K. M., P. M. P. Rambags and P. A. C. Verkoulen, Specification and Simulation with ExSpect, in Lauer (ed.), 1993.

Van Marcke K., KRS : An Object Oriented Representation Language, *Revue d'IA,* 1:4, pp. 43-68, 1987.

Veloso M. and A. Aamodt (Eds.), *Case-Based Reasoning Research and Development,* Springer, 1995.

Vygotsky L., *Thought and Language,* MIT Press, 1962.

Voinov A. V., Netlog - A Concept Oriented Logic Programming Language, dans A. Voronkov (ed.), *Logic Programming and Automated Reasoning,* Springer-Verlag, 1992.

Von Martial F., Interactions among autonomous planning agents, in Demazeau and Müller (eds.), 1990.

Von Martial F., *Coordinating Plans of Autonomous Agents,* Springer-Verlag, 1992.

Vrain C. and Y. Kodratoff, The Use of Analogy in Incremental SBL, In K. Morik (ed.), *Knowledge Representation and Organization in Machine Learning,* Springer-Verlag, 1987.

Wakita K., First Class Messages as First Class Continuations, in Nishio S. et A. Yonezawa (Eds.), 1993.

Waltz D. L., Memory-Based Reasoning, in Arbib M. A. et J. A. Robinson (Eds.), *Natural and Artificial Parallel Computation,* The MIT Press, 1990.

Wasserman K., *Unifying representation and generalization: Understanding hierarchically structured objects,* Doctoral Dissertation, Columbia University, New York, 1985.

Watson I., P. Watson et J. V. Woods, Parallel Data Driven Graph Reduction, dans J. V. Woods (ed), *Fifth Generation Computer Architectures,* Elsevier Science Publishers, p. 203-220, 1986.

Watt D. A., *Programming language,* Concepts and Paradigms, Prentice Hall, 1990.

Wavish P. et M. Graham, Roles, Skills and Behaviors: a Situated Action Approach to Organising Systems of Interacting Agents, in Wooldridge M. J. et N. R. Jennings (Eds.), 1995.

Weerasooriya D., A. Rao et K. Ramamohanarao, Design of a Concurrent Agent-Oriented Language, in Wooldridge M. J. et N. R. Jennings (Eds.), 1995.

Weld D. S. and J. de Kleer (eds.), *Reading in Qualitative reasoning about physical systems,* Morgan Kaufmann, 1990.

Wenger E., *Artificial Intelligence and Tutoring Systems,* Morgan Kaufmann, 1987.

Werner E., Cooperating Agents: A Unified Theory of Communication and Social Structure, dans Gasser et huhns (eds.), 1989.

Werner E., Distributed Cooperation Algorithms, dans Demazeau et Müller (eds.), 1990.

Wess S., K. Althoff et M. M. Richter (Eds.), *Topics in Case-Based Reasoning,* Springer-

Verlag, 1993.

Wetter Th., K. -D. Althoff, J. Boose, B. R. Gaines, M. Linster, F. Schmalhofer (eds.): *Current Developments in Knowledge Acquisition,* Springer-Verlag, 1992.

Whipple W., Expert Humans and Expert Systems: Toward a Unity of Uncertain Reasoning, in Pfeiffer et Nagle (Eds.), 1992.

Widman L. E., K. A. Loparo and N. R. Nielsen (eds.), *AI, Simulation and Modeling,* John Wiley & Sons, 1989.

Willems M., Projection and unification for conceptual graphs, in Ellis et al. (Eds.), 1995.

Winston P. H., Learning structural descriptions from examples, in: Winston P. H. (Ed), *The psychology of computer vision,* McGraw-Hill, 1975.

Winston P. H., Learning and Reasoning by Analogy, *Commun. of the ACM*, 23:12, pp. 689-703, 1980.

Woods W. A., Understanding subsumption and taxonomy: A framework for progress, dans Sowa J. F. (ed), 1992.

Woods W. A. et J. G. Schmolze, The KL-ONE family, in Lehmann (ed), 1992.

Wooldridge M. J. , This is MYWORLD: The Logic of an Agent-Oriented DAI Testbed, in Wooldridge M. J. et N. R. Jennings (Eds.), 1995.

Wooldridge M. J. et N. R. Jennings (Eds.), *Intelligent Agents,* Springer-Verlag, 1995.

Wooldridge M. J. et N. R. Jennings, Agent Theories, Architectures, and Languages: A Survey, in Wooldridge M. J. et N. R. Jennings (Eds.), 1995.

Wuwongse V., B. G. Ghosh, Towards Deductive Objective-Oriented Databases Based on Conceptual Graphs, in Proc. of the 7th Annual Workshop on Conceptual Structures, 1992.

Yang G., Y. Choi et J. Oh, CGMA: A Novel conceptual graph matching algorithm, in Nagle et al. (Eds.), 1992.

Yen Y., J. Lee, Issues in Using Conceptual Graphs for Knowledge Specification, in Proc. of the Sixth Annual Workshop on Conceptual Structures, 1991.

Yonezawa A. et M. Tokoro (eds.), *Object-Oriented Concurrent Programming,* MIT Press, 1987.

Yonezawa A., E. Shibayama, T. Takada et Y. Honda, Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1, dans Yonezawa A. et M. Tokoro (eds.), 1987.

Shibayama E. et A. Yonezawa, Distributed Computing in ABCL/1, dans Yonezawa A. et M. Tokoro (eds.), 1987.

Yonezawa A. (ed), *ABCL : An Object-Oriented Concurrent System,* MIT Press, 1990.

Annexe A

Opérations sur la hiérarchie des types et sur les Graphes Conceptuels (GC)

Nous définissons les opérations sur les types des concepts puis les opérations sur les GC. Une définition algorithmique des opérations est également présentée.

A.1 Opérations sur la hiérarchie des types de concepts

Les types des concepts forment une hiérarchie de généralisation des types. Appelons la relation "de généralisation" qui existe entre un type T1 et un sous-type direct Type2 par "sp" : T1 -sp-> T2.
Trois opérations principales sont définies sur une hiérarchie de types. Un exemple, illustrant l'utilisation de ces opérations, est fourni à la fin de la section.

Définition : *EstSuperType(**in** T1, T2: Type): Boolean*
L'opération vérifie si le type T1 est plus général que T2; il existe dans la hiérarchie un lien "sp" entre T1 et T2 : T1-sp->T2 (T2 est plus **sp**écifique que T1), ou bien il existe un type T3 dans la hiérarchie telque T1-sp->T3 et EstSuperType(T3, T2) est vérifiée.
Le type T1 est appelé super-type de T2 et inversement, T2 est appelé sous-type de T1. ♦

Deux autres opérations, reliées à **EstSuperType**, peuvent être considérées :
SousTypes(**in** T: Type; **out** LTypes: ListOf Type), /*** fournir les sous-types d'un type T ***/
SuperTypes(**in** T: Type; **out** LTypes: ListOf Type). /*** fournir les super-types d'un type T ***/

Définition : *PPSousTypeCom(in T1, T2: Type; out T3: Type)*
L'opération cherche dans la hiérarchie le plus proche sous-type commun T3, des deux types T1 et T2. ♦

Note : "plus proche" signifie qu'il n'existe pas un autre type T4 qui soit aussi un sous-type commun des deux types T1 et T2 et qui soit plus général que T3.

Une variante de cette opération est de donner le type T3 en entrée, il s'agit alors de vérifier si T3 est le plus proche sous-type commun des deux types T1 et T2.

Définition : *PPSuperTypeCom(in T1, T2: Type; out T3: Type)*
L'opération cherche dans la hiérarchie, le plus proche super-type commun T3 des deux types T1 et T2. ♦

Note : Comme pour l'opération précédente, "plus proche" signifie qu'il n'existe pas un autre type T4 qui soit aussi un super-type commun des deux types T1 et T2 et que T3 est plus général que T4.
Une variante de cette opération est de donner le type T3 en entrée, il s'agit alors de vérifier si T3 est le plus proche super-type commun des deux types T1 et T2.

Remarque : La hiérarchie des types peut être un graphe quelconque mais nous supposons qu'il y a au plus un seul PPSousTypeCom et un seul PPSuperTypeCom pour deux types donnés.

Les exemples dans cet annexe ainsi que les algorithmes sont extraits sans traduction du manuel du langage *Prolog+CG* [Kabbaj, 95], rédigé en anglais.

Exemple : L'exemple porte sur la hiérarchie suivante (Figure A.1) :

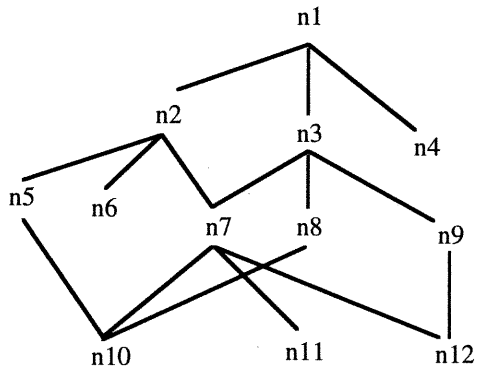


Figure A.1: Hiérarchie de types

Sa description textuelle en Prolog+CG est sauvegardée dans le fichier "hier1" comme suit :

```
n1 > n2, n3, n4.
n2 > n5, n6, n7.
n3 > n7, n8, n9.
n5 > n10.
n6 = (black,white,red).
n7 > n10, n11, n12.
n8 > n10.
n9 > n12.
n10 = integer.
```

.

> Prolog Prolog+CG
...

| ?- compileQuickHier hier1.

hier1 compiled.
...

| ?- quest.
|: typeOperations::{[superTypes]-
|: in->[concType: n7]
|: out->[concTypes: SuperTypes]}. /*** *superTypes("n7", SuperTypes)* ***/
SuperTypes = [n2,n3,n1]

yes
| ?- quest.
|: typeOperations::{[superTypes]- in->[concType: n1] out->[concTypes: SuperTypes]}.
SuperTypes = []

yes
| ?- quest.
|: typeOperations::{[isSuperType]- in1->[concType:n2] in2->[concType: n12]}.

yes
| ?- quest.
|: typeOperations::{[isSuperType]- in1->[concType: n2] in2->[concType: n9]}.

no
| ?- quest.
|: typeOperations::{[subTypes]- in->[concType:n3] out->[concType:SubTypes]}.
SubTypes = [n7,n8,n9,n10,n11,n12]

yes
| ?- quest.
|: typeOperations::{[subTypes]- in->[concType:n10] out->[concType:SubTypes]}.
SubTypes = []

yes
| ?- quest.
|: typeOperations::{[maxComSubType]- in1->[concType:n2] in2->[concType:n3] out->[concType:Type]}.
Type = n7

yes
| ?- quest.
|: typeOperations::{[maxComSubType]- in1->[concType:n2] in2->[concType:n3] in3->[concType:n10]}.

no
| ?- quest.
|: typeOperations::{[maxComSubType]- in1->[concType:n2] in2->[concType:n11] out->[concType:Type]}.
Type = n11

yes
| ?- quest.
|: typeOperations::{[maxComSubType]- in1->[concType:n2] in2->[concType:n4] out->[concType:Type]}.

no
| ?- quest.

```
|: typeOperations::{[minComSuperType]- in1->[concType:n5]  in2->[concType:n12]
|:                                    out->[concType:Type]}.
Type = n2

yes
| ?- quest.
|: typeOperations::{[minComSuperType]- in1->[concType:n5]  in2->[concType:n9]  in3->[concType:n1]}.

yes
| ?- quest.
|: typeOperations::{[minComSuperType]- in1->[concType:n3]  in2->[concType:n10]
|:                                    out->[concType:Type]}.
Type = n3

| ?- quest.
|: typeOperations::{[conform]- in1->[referent:white]  in2->[concType:n6]}.

yes
| ?- quest.
|: typeOperations::{[conform]- in1->[referent:white]  in2->[concType:n1]}.

yes
| ?- quest.
|: typeOperations::{[conform]- in1->[referent:white]  in2->[concType:n3]}.

no
```

A.2 Une algèbre pour les GC

Définition : *Algèbre pour les GC*
Une définition des GC et des opérations associées constitue une *algèbre* pour les GC. ♦

Introduisons en premier le type de GC traités par les opérations définies dans cet annexe.

Définition : *GC fonctionnel*
Un GC g est fonctionnel si tout concept de g n'est pas attaché à des relations de même direction et de même type. En plus, un GC fonctionnel ne peut contenir des concepts avec des référents identiques. ♦

Par exemple, le graphe g1 ci-dessous est non fonctionnel alors que g2 l'est.

```
g1 =                                  g2 =
  [Table] -                             [Table] -
      -attr-> [Couleur]                       -couleurDe-> [Couleur]
      -attr-> [Poids]                         -poidsDe-> [Poids]
```

Remarques :
→ Nous avons introduit les GC fonctionnels dans [Kabbaj, 87] afin d'éviter, lors d'un appariement, l'indéterminisme qui résulterait de l'appariement des relations pour des GC

quelconques. Par exemple, la branche [Table]-attr->[Qualite] peut être appariée à [Table]-attr->[Couleur] de g1 ou à [Table]-attr->[Poids].

Les opérations définies dans [Kabbaj, 87] recoivent deux GC fonctionnels à apparier, ainsi que deux concepts considérés comme points d'entrée. Le traitement est alors déterministe sans retour arrière et l'opération retourne, si elle réussit, un seul GC comme résultat.

Sans la contrainte fonctionnelle sur les GC, les opérations seraient non-déterministes, produisant, par retour-arrière par exemple, plusieurs GC en sortie.

Ainsi, afin d'avoir une définition efficace des opérations, nous avons restreint dans [Kabbaj, 87] la généralité des GC au cas des GC fonctionnels. L'algèbre que nous proposons dans cet annexe tente de conserver le même objectif (opération déterministe sans retour arrière), la restriction sur les GC est toutefois remplacée par une restriction moins stricte : les GC peuvent être non-fonctionnels mais, les opérations sont définies en supposant que l'indéterminisme est résolu immédiatement en considérant l'appariement des concepts attachés aux relations.

→ Rappelons que notre définition des opérations porte sur des GC qui peuvent être imbriqués, permettre plusieurs résultats dans un tel contexte rendrait le traitement très lourd et très problématique. Par ailleurs et dans le cadre de Prolog+CG, avoir une définition non-déterministe de l'unification, allourdirait davantage le mécanisme d'inférence (bien qu'elle peut être utile dans certains cas).

Définition : *GC "quasi"-fonctionnel*

un GC "quasi"-fonctionnel est un GC qui satisfait la contrainte suivante : si une opération d'appariement met en correspondance un concept c1 d'un GC "quasi"-fonctionnel g1 avec un concept c2 d'un GC "quasi"-fonctionnel g2 et si c1 est attaché à une branche : c1 -r1- c11, alors cette branche peut être appariée à au plus une seule branche de c2. ♦

Remarque : Tout GC fonctionnel est un GC "quasi"-fonctionnel. Nous considérons les GC "quasi"-fonctionnels au lieu des GC fonctionnels uniquement car dans certaines applications, on rencontre des GC qui appartiennent à la première catégorie et non à la seconde (par exemple, en descrivant une entité avec ses différents attributs, les relations reliant l'entité aux attributs sont souvent des "attr" mais les attributs sont généralement de types différents, comme le cas précédent de la table avec la couleur et le poids).

Un inconvénient de la définition ci-dessus des GC "quasi"-fonctionnels est qu'elle présente une restriction implicite, contrairement à la définition des GC fonctionnels où la contrainte sur les relations est clairement et explicitement spécifiée. Nos opérations peuvent retourner un "mauvais" résultat si on les applique sur des GC qui ne sont pas "quasi"-fonctionnels.

Dans certains cas et pour une utilisation plus sûre des opérations proposées dans cet annexe, il serait plus approprié de représenter le domaine d'application par des GC fonctionnels.

Considérons maintenant les opérations sur les GC.

Notre définition des opérations sur les GC est basée sur l'opération d'appariement (matching). En effet, comme le montre la Figure A.2, les opérations "principales" sont des *spécialisations* de l'appariement. CopiePartielle et Contraction sont des opérations "secondaires" utilisées par certaines opérations principales. L'ensemble des opérations forme une *hiérarchie de généralisation/composition* (Figure A.2). Nous fournissons une brève description des opérations, description qui correspond à un "parcours" de la hiérarchie ! Une définition plus détaillée des opérations suit.

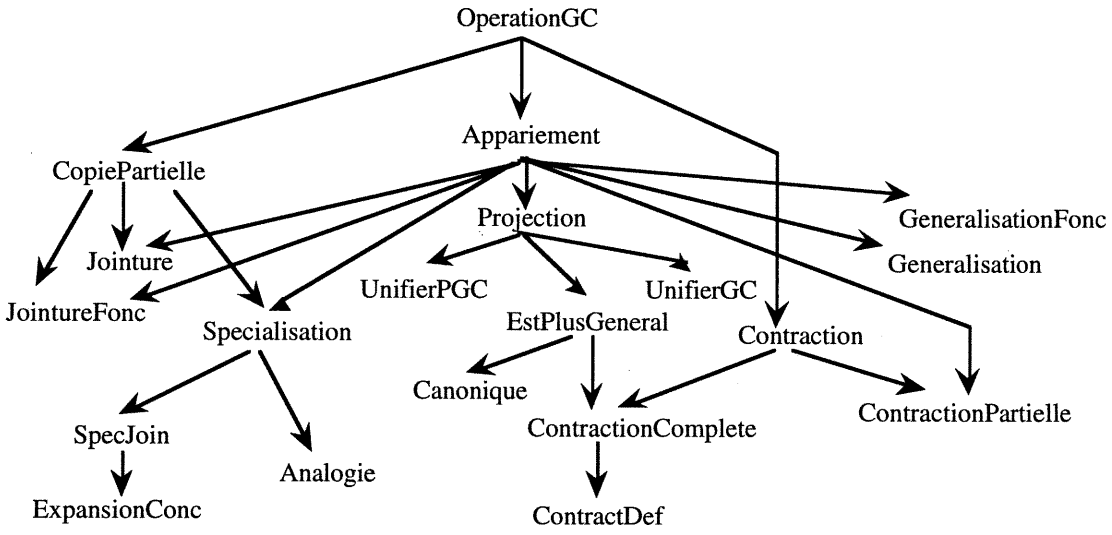


Figure A.2 : La hiérarchie des opérations sur les GC

Les opérations *Jointure* (fusionner deux GC en un GC qui soit leur spécialisation commune), *Généralisation* (déterminer un GC qui soit la généralisation commune des deux GC), *Specialisation* (spécialiser un graphe par un autre) et *Projection* (vérifier que, du point de vue structurel seulement, un GC est un sous-graphe d'un autre GC) sont considérées comme des spécialisations de l'opération générique *Appariement* (déterminer, du point de vue structurel seulement, un sous-graphe commun au deux GC).

Comme l'appariement, la projection et la spécialisation sont des "opérations génériques", spécialisées en "opérations concrètes".

JointureFonc et *GeneralisationFonc* sont des variantes des opérations Jointure et Generalisation. Les deux formes de jointure (Jointure et JointureFonc) impliquent en général la copie de certaines parties de deux graphes dans le graphe résultat (elles utilisent *CopiePartielle*).

L'opération spécialisation, qui est une forme d'appariement, effectue aussi en général, une copie dans le second graphe de certaines parties du premier graphe.
L'opération spécialisation est spécialisée en *SpecJoin* (spécialisation par jointure) et *Analogie* (spécialisation par analogie). SpecJoin est une variante de la Jointure, SpecJoin ne produit pas un nouveau graphe comme la Jointure, mais spécialise le second graphe par le premier graphe. SpecJoin constitue le corps principal de l'opération *ExpansionConc* (spécialiser un graphe par une jointure avec la définition du type d'un de ses concepts).
La projection est spécialisée en*UnifierCG* (unifier un GC avec un autre), *UnifierPCG* (unifier, sans effet de bord sur les deux graphes) et *EstPlusGeneral* (vérifier qu'un GC est plus général qu'un autre).
L'opération *Canonique* (vérifier qu'un graphe respecte les canons des types de concepts et de relations qu'il contient) est basée essentiellement sur l'opération EstPlusGeneral (chaque canon doit être plus général que le graphe en question). L'opération *ContractionComplete* (contracter un GC d'un autre, le premier doit être plus général que le second) est composée de EstPlusGeneral et Contraction.
L'opération *ContractDef* (contracter d'un graphe la définition du type d'un de ses concepts) est basée sur ContractionComplete.
L'opération *ContractionPartielle* (enlever d'un GC les parties communes avec un autre graphe) est une variante de ContractionComplete. L'opération *Contraction* (enlever des parties d'un graphe sans créer une rupture du graphe) est généralement effectuée suite à un appariement (ou suite à EstPlusGeneral).

Nous allons définir à présent chaque opération et fournir quelques exemples d'utilisation. Une définition algorithmique des opérations est fournie dans la prochaine section.

Définition : *Appariement de deux GC*
L'appariement de deux GC g1 et g2 est une opération qui identifie deux sous-graphes de g1 et g2 ayant la même structure. Les concepts des deux sous-graphes sont mis en correspondance. ♦

En terme général, l'opération d'appariement de deux GC g1 et g2 est réalisée comme suit :
Choisir de g1 et g2 deux concepts ce1 et ce2 à apparier et effectuer ensuite leur *appariement local*. Cette dernière commence avec deux concepts c1 et c2 de g1 et g2 respectivement, elle les apparie et cherche ensuite à apparier des relations attachées à c1 et c2. L'opération propage ensuite l'appariement local, de façon récursive, aux autres concepts attachés aux relations appariées. Les concepts/relations déjà appariés ne le seront pas une seconde fois.

Ainsi, l'appariement de deux concepts incite à l'appariement de certaines relations qui incitent à leur tour l'appariement de certains concepts et ainsi de suite jusqu'à ce que l'appariement couvre les deux graphes en entrée.

Définition : *Apparier deux concepts*
Apparier deux concepts consiste à *apparier leurs types et leur référents*. Si ces derniers sont des GC, alors ils sont appariés également, de façon récursive. En général et selon la nature de l'appariement des GC (jointure, généralisation, etc.), l'appariement des types correspond à une des opérations définies sur la hiérarchie des types (PPsousTypeCom, PPsuperTypeCom, etc.). ♦

Définition : *Apparier deux relations*
Soit c1-R-c11 et c2-R-c22 deux relations de g1 et g2 respectivement, avec c1 déjà apparié à c2, ou c11 déjà apparié à c22. Les deux relations peuvent s'apparier si elles ont même direction (c1<-R-c11 et c2<-R-c22, ou c1-R->c11 et c2-R->c22), même type de relation et si leurs "arguments" peuvent s'apparier (apparier c1 à c2 et c11 à c22. Notons qu'au moins un des deux couples a déjà été traité, comme l'indique la première partie de la définition). ♦

Note: Si la hiérarchie des relations était à considérer, il suffirait de vérifier non pas l'identité de leur types, mais leur appariement, comme pour les types de concepts. Aussi, si les relations devaient être n-adiques, il suffirait d'apparier leurs "arguments", le $i^{ème}$ argument de la première relation avec le $i^{ème}$ argument de la seconde.

Définition : *Initiation de l'appariement des deux GC*
L'appariement *peut* recevoir en entrée, en plus des deux graphes g1 et g2, deux concepts à apparier de g1 et g2 respectivement. Si c'est le cas, l'opération commence alors à partir des deux concepts, considérés comme deux "points d'entrée", sinon l'appariement cherche deux relations de g1 et g2 ayant même type : c1 -R-> c11 et c2 -R-> c22 ; l'opération tente ensuite d'apparier les sources des deux relations : c1 avec c2. Ces derniers sont alors considérés comme deux "points d'entrée", inférés par l'opération elle-même. ♦

Toutes les opérations dérivées de l'appariement utilisent les deux moyens cités dans la définition ci-dessus concernant l'initiation de l'appariement. D'autres moyens sont encore considérés, par certaines spécialisations de l'appariement, comme la jointure et l'unification.

Exemple :

Note: rappelons que les exemples sont extraits du manuel de Prolog+CG. On demande à l'objet *cgOperations* d'activer une de ses méthodes qui corespondent aux opérations définies ci-dessus. L'appel est formulée comme un GC.

```
| ?- quest.
|: cgOperations::{[match]-
|:                    in1->[cg: {[work]-agnt->[person:kabbaj]
|:                                        obj->[software]-qlty->[prototype]} ]
|:                    in2->[cg:{[cptTeam:tm34]-chrc->[work]-dur->[week]
|:                                                    agnt->[person]
|:                                                   obj->[software:prologPCG]-attr->[interest:high]}]
|:                        out->[cg:Res]}.

Res = {[match(work,work): match(,)]-
                                obj->[match(software,software): match(,prologPCG)]
                               agnt->[match(person,person): match(kabbaj,)] ,}
```

Comme nous l'avons noté auparavant, l'appariement de deux GC est une opération "abstraite" qui est spécialisée en opérations "concrètes" comme la jointure, l'unification et la généralisation. Dans la définition de ces dernières, nous notons en quoi leur description est une spécialisation de celle de l'appariement.

Définition : *Jointure de deux GC*

La jointure de deux GC g1 et g2 est un appariement modulo la jointure des concepts ("apparier deux concepts" est donc spécialisée en "joindre deux concepts"), le résultat de cet appariement est un GC g3 qui est ensuite augmenté par la copie des "parties propres" à g1 et g2. G3 est donc une fusion de g1 et g2. ♦

Note : Les *parties propres* à un graphe sont ceux qui n'ont pas été appariées ou que leur appariement a échoué. Par exemple, si le graphe g1 contient la relation c1-R1->c11 et le concept c1 (ou c11) est déjà joint avec un concept de g2 et si aucune relation de g2 ne peut se joindre à c1-R1->c11, alors cette dernière relation est ajoutée au graphe résultat g3.

Définition : *Jointure de deux concepts*

La jointure de deux concepts consiste à "joindre" leurs types et leurs référents. Pour les types, cela correspond à trouver le plus proche sous-type commun. Pour les référents, s'ils sont des GC, alors il faut effectuer leur jointure autrement, il faut déterminer le plus spécifique des deux référents. Si l'un des référents est une instance et l'autre une variable alors le résultat est l'instance, si les deux référents sont des instances différentes alors on ne peut déterminer le plus spécifique des deux. Deux concepts ne peuvent se joindre si

leur types ou leurs référents ne peuvent se joindre; les deux types n'ont pas de sous-type commun ou les deux référents sont des instances différentes. ♦

La prochaine définition se base sur la contrainte "qu'un graphe ne peut pas contenir des concepts différents avec des référents identiques". Si deux graphes g1 et g2 sont à joindre par exemple et si g1 contient le concept [Personne: jean34] et g2 le concept [Homme: jean34], alors ces deux concepts doivent se joindre, autrement le graphe résultat contiendrait deux concepts différents avec le même référent "jean34".

Définition : *Identification des points d'entrée pour la jointure*
Si les points d'entrée ne sont pas fournis en entrée, la jointure tente de les identifier par l'identité de leurs référents ou par la "jointure" de leurs co-références. Dans le premier cas, elle cherche respectivement deux concepts de g1 et g2, ayant le même référent (une instance et non une variable). Le second cas peut survenir avec la jointure de graphes imbriqués et avec des concepts en co-référence : soit deux GC imbriqués g1 et g2 (Figure A.3), g1 contient un concept dont le référent est un GC g1_2 et un concept C1 qui est en co-référence avec un concept C1_4 de g1_2; C1 et C1_4 réfèrent donc au même individu. Une description similaire s'applique sur le GC g2 (Figure A.3).

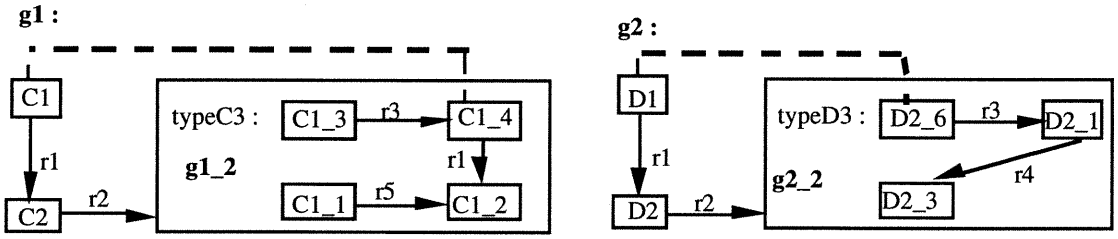


Figure A.3 : GC imbriqués avec liens de coréférence

Si, durant la jointure de g1 et g2, le concept c1 de g1 est joint avec le concept d1 de g2, et le concept [TypeC3: g1_2] est *à joindre* avec le concept [TypeD3: g2_2], alors g1_2 doit se joindre avec g2_2 en considérant comme points d'entrée les concepts c1_4 et d2_6. En effet, si c1_4 n'est pas joint avec d2_6, alors le concept produit par la jointure des concepts c1 et d1 serait en co-référence avec deux concepts différents d'un même graphe.
Si les deux possibilités ci-dessus ne peuvent s'appliquer, alors on considère la jointure de deux relations. Autrement, on considère la jointure de deux concepts des deux GC g1 et g2. ♦

Définition : *Jointure fonctionnelle*

La jointure fonctionnelle, *JointureFonc*, est une jointure qui s'applique sur deux GC fonctionnels. ♦

Définition : *Jointure des relations dans une jointure fonctionnelle*

Le caractère fonctionnel de l'opération *JointureFonc* implique que si le graphe g1 contient la relation c1-R1->c11 et c1 (ou c11) est déjà joint avec un concept c2 (ou c22) de g2 respectivement et si g2 contient une relation : c2-R1->c22, alors la jointure de c1 avec c2, et de c11 avec c22 *doit* être possible autrement, la jointure actuelle des deux graphes est jugée impossible. ♦

Définition : *Identification des points d'entrée pour la jointure fonctionnelle*

La détermination des points d'entrée pour la jointure fonctionnelle est identique à celle de la jointure avec en plus la possibilité suivante : si la jointure fonctionnelle *courante* des deux graphes a été initiée par des points d'entrée déterminés par la jointure de deux relations, alors l'opération tente d'initier une autre jointure des deux graphes, en considérant une nouvelle jointure de relations. Si aucune des jointures n'aboutie, alors l'opération conclut que les deux graphes ne peuvent se joindre. ♦

Remarque : La possibilité ci-dessus ne s'applique pas à la jointure "normale" car cette dernière ne "s'arrête" pas si deux branches ne peuvent se joindre; elles sont ajoutées au graphe résultat. Un retour au point de départ pour une nouvelle jointure avec de nouveaux points d'entrée ne peut donc survenir pour une jointure normale.

Exemples pour la jointure et la jointure fonctionnelle

```
| ?- quest.
|: cgOperations::{[maximalJoin]-
|:                    in1->[cg:{[cptTeam:tm34]-chrc->[work]-
|:                                                    agnt->[person:kabbaj]   dur->[week]
|:                                                    obj->[software:prologCG]-attr->[interest:high]}]
|:                    in2->[cg:{[work]-agnt->[person:P]   obj->[software]}]
|:                     out->[cg:G]}.
P = _V1
G = {[cptTeam: tm34]-chrc->[work]-
                              agnt->[person: kabbaj]
                              obj->[software: prologCG]-attr->[interest: high] ,
                              dur->[week] , ,}

yes
| ?- quest.
|: cgOperations::{[maximalJoin]-
|:                      in1->[cg:{[man:jo]}]
|:                     in2->[cg:{[drink]-agnt->[man]   obj->[beer]}]
|:                      out->[cg:G]}.
G = {[drink]-
```

```
            obj->[beer]
            agnt->[man: jo] ,}

yes
| ?- quest.
|: cgOperations::{[maximalJoin]-
|:                          in1->[cg:{[break]-agnt->[employee]
|:                                        obj->[machine:M]-qlt->[expensive] &
|:                                  [rent]-agnt->[society]
|:                                        obj->[machine:M]
|:                                        dur->[years:4]        }     ]
|:                          in2->[cg:{[situation]-mainAct->[break]-agnt->[elecTrouble]
|:                                                                  obj->[machine:res23]-qlt->[big]  } ]
|:                           out->[cg:JoinRes]}.
M = _V1
JoinRes = {[situation]-mainAct->[break]-
                                  agnt->[elecTrouble]
                                  agnt->[employee]
                                   obj->[machine: res23]-
                                                  qlt->[big]
                                                  qlt->[expensive] , , ,
            &
            [rent]-
               agnt->[society]
               obj->[machine: res23]
               dur->[years: 4] ,}

yes
| ?- quest. /*** Le même exemple mais avec une jointure fonctionnelle ***/
|: cgOperations::{[funcMaximalJoin]-
|:                           in1->[cg: {[break]-agnt->[employee]
|:                                        obj->[machine:M]-qlt->[expensive] &
|:                                  [rent]-agnt->[society]
|:                                        obj->[machine:M]
|:                                        dur->[years:4]}]
|:                          in2->[cg:{[situation]-mainAct->[break]-agnt->[elecTrouble]
|:                                                                  obj->[machine:res23]-qlt->[big]}]
|:                           out->[cg: JoinRes]}.

no /* dans la hiérarchie des types, aucun sous-type commun n'existe pour "employee" et "elecTrouble". */


| ?- quest.  /*** cas de jointure sur des GC composés ***/
|: cgOperations::{[maximalJoin]-
|:                          in1->[cg:{[think]-agnt->[person:P]
|:                                        obj->[prop: {[go]-agnt->[person:P]
|:                                                          dest->[city]}]}]
|:                          in2->[cg:{[think]-
|:                                        obj->[prop: {[go]-agnt->[person:rachid]}]
|:                                        ctxt->[discourse]}]
|:                          out->[cg: JoinRes]}
|:  .
P = _V1
JoinRes = {[think]-
                  obj->[prop: {[go]-
                                 agnt->[person: rachid]
                                 dest->[city] ,}]
                  ctxt->[discourse]
                  agnt->[person: rachid] ,}


yes
```

Le manuel de Prolog+CG contient d'autres exemples illustrant différents points concernant les deux jointures.

Définition : *Généralisation de deux GC*
La généralisation de deux GC g1 et g2 est un appariement modulo la généralisation des concepts ("apparier deux concepts" est donc spécialisée en "généraliser deux concepts"). Cet appariement produit un GC g3 qui est une généralisation commune des deux graphes. ♦

Définition : *Généralisation de deux concepts*
La généralisation de deux concepts consiste à "généraliser" leurs types et leurs référents. Pour les types, cela correspond à trouver le plus proche super-type commun. Pour les référents, s'ils sont des GC, alors il faut effectuer leur généralisation autrement, il faut déterminer le plus général des deux référents. Si les deux référents sont des instances identiques alors leur généralisation est cette instance, sinon c'est une variable. Deux concepts ne peuvent être généralisés si leurs types n'ont pas de super-type commun (ou si c'est Universal, la racine de la hiérarchie des types. Cette contrainte est toutefois inappropriée pour certaines applications). ♦

Définition : *Généralisation fonctionnelle*
La généralisation fonctionnelle, *GeneralisationFonc,* est une généralisation qui s'applique sur deux GC fonctionnels. ♦

Définition : *Généralisation fonctionnelle des relations*
Si le graphe g1 contient la relation c1-R1->c11 et c1 (ou c11) est déjà généralisé avec un concept c2 (ou c22) de g2 respectivement et si g2 contient une relation c2-R1->c22, alors la généralisation de c1 avec c2 et de c11 avec c22 *doit* être possible autrement, la généralisation *actuelle* des deux graphes est impossible. Si la généralisation courante a été initiée par des points d'entrée déterminés par la généralisation de deux relations, alors l'opération tente une autre généralisation de relations. ♦

Exemples pour la généralisation et la généralisation fonctionnelle

```
| ?- quest.
|: cgOperations::{[generalize]-
|:                  in1->[cg:{[cptTeam:tm34]-chrc->[work]-
|:                                                  agnt->[person:kabbaj]
|:                                                  dur->[week]
|:                                                 obj->[software:prologPCG]-attr->[interest:high]}]
|:                  in2->[cg:{[work]-agnt->[person:kabbaj]-spec->[knowledgeRepr] ,
|:                                        obj->[software]}]
|:                   out->[cg:CGres]}
```

```
|:   .
CGres = {[work]-
                obj->[software]
                agnt->[person: kabbaj] ,}

yes


| ?- quest.
|: cgOperations::{[generalize]-
|:                          in1->[cg:{[drink]-agnt->[man:mark]
|:                                    obj->[beer]}]
|:                           in2->[cg:{[man:jo]}]
|:                           out->[cg: G]}.
G = {[man]}

yes
| ?- quest.
|:  cgOperations::{[generalize]-
|:                          in1->[cg:{[break]-agnt->[employee]
|:                                    obj->[machine:M]-qlt->[expensive] &
|:                                [rent]-agnt->[society]
|:                                    obj->[machine:M]
|:                                    dur->[years:4]}]
|:                          in2->[cg:{[situation]-mainAct->[break]-agnt->[elecTrouble]
|:                                                          obj->[machine:res23]-qlt->[big]}]
|:                           out->[cg:Gres]}.
M = _V1
Gres = {[break]-obj->[machine] ,}

yes
| ?- quest.
|:  cgOperations::{[funcGeneralize]-
|:                           in1->[cg:{[break]-agnt->[employee]
|:                                    obj->[machine:M]-qlt->[expensive] &
|:                                [rent]-agnt->[society]
|:                                    obj->[machine:M]
|:                                   dur->[years:4]}]
|:                          in2->[cg:{[situation]-mainAct->[break]-agnt->[elecTrouble]
|:                                                          obj->[machine:res23]-qlt->[big]}]
|:                            out->[cg:G]}.

no
```

Définition : *Spécialisation d'un graphe par un autre*

La spécialisation d'un graphe g1 par un autre g2 est l'appariement, sans produire de graphe résultat, suivie en général d'une copie partielle de g2 dans g1. ♦

La spécialisation de g1 résulte donc de l'ajout, à g1, de certaines parties de g2 mais aussi de l'effet de l'appariement de g1 avec g2. Les définitions de *SpecJoin* et *Analogie* spécialisent cette opération "abstraite" (ou générique).

Définition : *Specialisation par jointure*

L'opération *SpecJoin* spécialise un graphe g1 par un autre g2 en effectuant un appariement des deux graphes modulo la jointure des concepts, puis en ajoutant à g1 les parties spécifiques à g2. ♦

Exemple

```
[SpecJoin]-
        in1->[cg: G = {[work]-agnt->[person:P]
                                    obj->[software]} ]
         in2->[cg: {[cptTeam:tm34]-chrc->[work]-
                                              agnt->[person:kabbaj]
                                               dur->[week]
                                              obj->[software:prologCG]-attr->[interest:high]} ]

G = {[cptTeam: tm34]-chrc->[work]-
                                     agnt->[person: kabbaj]
                                    obj->[software: prologCG]-attr->[interest: high] ,
                                    dur->[week] , ,}
```

Définition : *Expansion d'un concept*

L'opération d'expansion d'un concept C d'un graphe g1, *ExpansionConc* utilise *SpecJoin* afin de spécialiser g1 par la définition du type du concept C. Le type du concept C est remplacé, dans g1, par son super-type. ♦

Exemple:

```
| ?- quest.
|: cgOperations::{[expandConc]-
|:                          in1->[cg:{[eat]-agnt->[man: jack]
|:                                          obj->[apple] }]
|:                           in2->[concept: ! [man: jack] ]
|:                          out->[cg:CGres]}.
CGres = {[eat]-agnt->[person: jack]-has->[sex: masc] ,
                obj->[apple] }

yes
```

Définition : *Analogie*

L'analogie est une *spécialisation* "analogique" d'un graphe cible g2 par un graphe source g1. Le graphe source peut contenir des *dépendances conceptuelles* entre ses concepts. Une dépendance conceptuelle, appelée aussi *règle de détermination*, spécifie qu'un type de concept c1 dépend d'un autre type de concept c2 (ou que le type de c2 détermine celui de c1). Les dépendances conceptuelles incorporent une connaissance du domaine et l'opération d'analogie suppose que cette connaissance s'applique aussi bien au graphe source qu'au graphe cible. Pour homogéneiser la représentation, nous considérons une dépendance conceptuelle comme un type particulier de relation.

Le transfert analogique du graphe source au graphe cible s'applique aussi sur les dépendances conceptuelles, déclenchant des inférences analogiques. En particulier, l'opération *Analogie* est réalisée selon les trois étapes suivantes :

→ spécialiser g2 par g1 avec un appariement "neutre" (les concepts de g2 ne sont pas modifiés suite à l'appariement). La spécialisation effectuera une copie dans g2 des parties non appariées de g1. Plus précisement, la copie dans g2 d'un concept c1 de g1 aura la forme <c1, X>, X étant une variable et la copie dans g2 d'une dépendance c1 -DepConc-> c12 de g1 sera comme suit : c2 -DepConc-> c12 avec c2 un concept de g2 ou une copie d'un concept de g1: <c, Y>. Il en est de même pour le concept c12.

→ vérifier/évaluer les dépendances conceptuelles de g2 : si une dépendance possède un concept/argument de la forme <c, X> alors remplacer le par X. Par exemple, si g2 contient c1 -Dep1-> <c11, X>, alors cette dépendance deviendra c1 -Dep1-> X, cette dernière est ensuite "évaluée", si possible. En effet, les dépendances peuvent partager des concepts/arguments, produisant ainsi des "dépendances" entre elles. Ainsi, on pourrait avoir dans g2 les dépendances c1 -Dep1-> X et X -Dep2-> Y. La seconde dépendance ne peut être évaluée avant la première. L'évaluation de celle-ci produirait une valeur (un concept) pour X et alors, la seconde dépendance peut être évaluée, produisant une valeur pour Y.

→ remplacer les concepts de g2 qui sont de la forme <c, X> par c. Ce sont des concepts qui ont été copiés de g1 et qui ne sont pas des arguments d'une dépendance conceptuelle. ♦

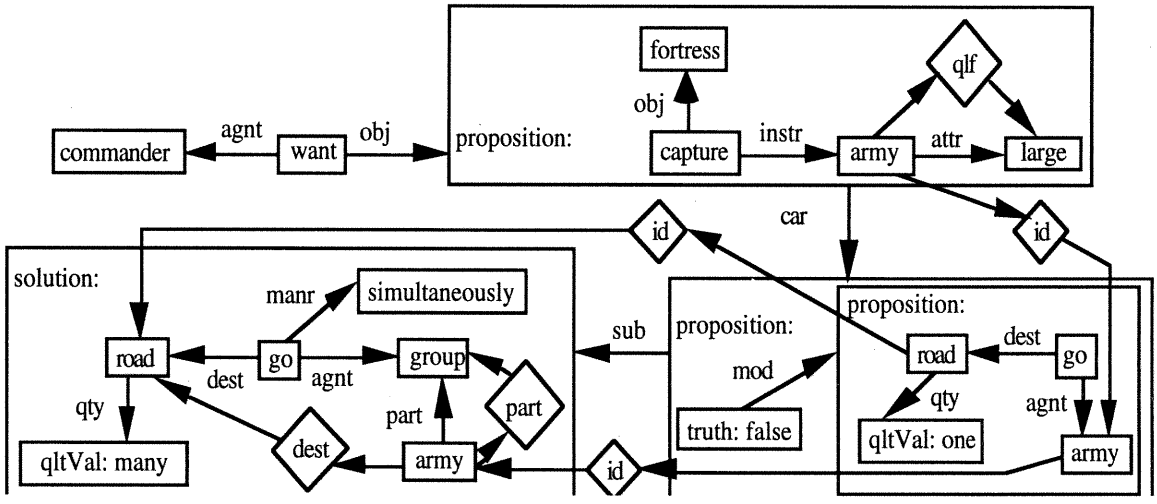
Remarque : Suite à une étude de différents travaux sur l'analogie (comme ceux de Winston, Gentner, Indurkhya, Gick et Holyoak, Greiner, Russell et autres), Leishman [Leishman, 89, 92] a développé un outil pour l'analogie, basé sur la proposition de Gick et Holyoak que "l'induction d'un schéma général à partir d'exemples concrets facilitera le transfert analogique".

L'outil de Leishman forme une analogie suite à des généralisations d'exemples (appelés "analogues") représentés par des GC. Un transfert analogique est ensuite effectué par inférence analogique, impliquant le transfert de certaines parties du graphe source au graphe cible et puis, une substitution des concepts en correspondance.

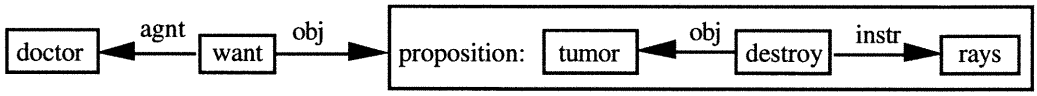
Il n'est toutefois pas clair (du moins dans [Leishman, 89, 92]) comment ce type d'inférence est réalisé (s'il l'a été fait) dans son outil. Dans tous les cas, Leishman note comme limitation de son outil : "une inférence analogique basée uniquement sur la correspondance peut être trompeuse .. et une solution serait de considérer les règles de détermination". A l'atelier sur les GC en 1990, elle a mentionné qu'un projet était en cours afin d'ajouter les règles de détermination à son outil analogique [Leishman, 90].

Exemple : Cet exemple est une adaptation de celui de Leishman [Leishman, 89] qui l'a elle même adapté de Gick et Holyoak [Gick et Holyoak, 83].
Le graphe source spécifie qu'un commandant veut capturer une forteresse avec une grande armée, qu'une mauvaise solution serait d'envoyer toute l'armée par un seul chemin et qu'une bonne solution serait plutôt d'envoyer des groupes (parties de l'armée) simultanément par des chemins différents. Le graphe cible spécifie qu'un docteur veut détruire une tumeur avec des rayons.
Concernant les dépendances conceptuelles, nous considérons que "Large" dépend du concept "Armée" (large est une valeur spéciale pour un attribut important d'une armé; sa "taille"), chemin dépend aussi de "Armée" (une armée suit un chemin) et enfin groupe dépend également de "Armée". Les liens de coréférences sont considérés comme des dépendances conceptuelles; une dépendance d'identité.
Comme connaissance d'arrière-plan associée au graphe cible, nous indiquons que la valeur spéciale pour un attribut important de "Rayon" est "Puissant" (l'attribut est Puissance), qu'un rayon a une direction et qu'il peut être divisé en des "rayons faibles".
Note : dans les diagrammes ci-dessous (Figure A.4), les dépendances conceptuelles sont représentées par des losanges.

gSource :



gCible :



gCible après l'analogie :

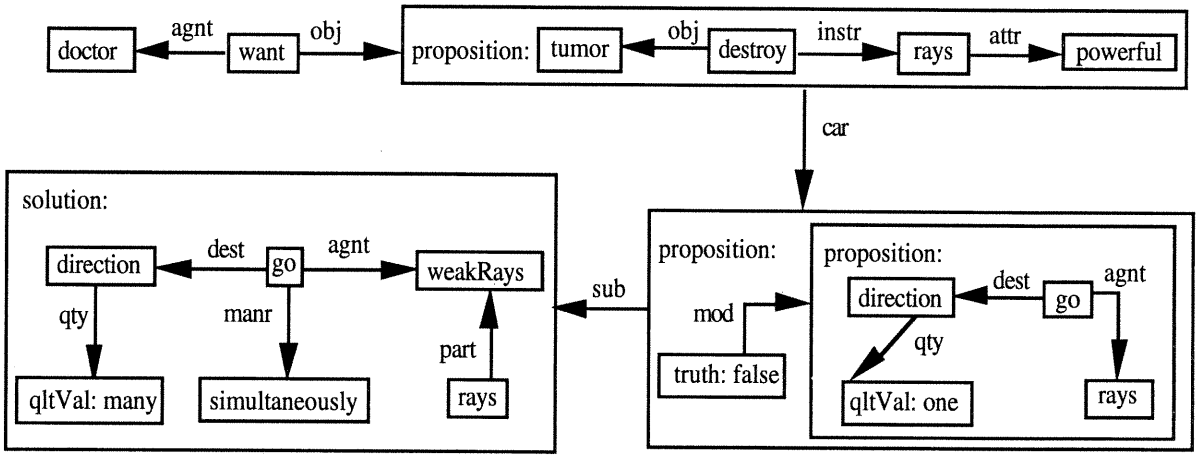


Figure A.4 : Exemple d'analogie

```
| ?- quest
|: cgOperations::
|:    {[analogy]-
|:            in1->[cg:{[want]-
|:                        agnt->[commander]
|:                        obj->[prop: {[capture]- obj->[fortress]
|:                                                inst->[army:A]-attr->[large:L]}]-car->
|:                            [prop: {[truth:false]-mod->[prop: {[go]-
|:                                                              agnt->[army:B]
|:                                                             dest->[road:R]-qty->[one] }] }]-sub->
|:                             [solution: {[army:C]-part->[group:E] &
|:                                         [go]- agnt->[group:E]
|:                                               man->[simultanous]
|:                                              dest->[road:R2]-qty->[many] }]  }]
|:                    -cstr->[constraints:[ [qlf, ![army:A], ![large:L]],  [id, ![army:A], ![army:B]],
|:                                          [id, ![army:B], ![army:C]],   [id, ![road:R], ![road:R2]],
|:                                          [dest, ![army:C], ![road:R2]],   [part, ![army:C], ![group:E]] ] ] ,
|:            in2->[cg:{[want]- agnt->[doctor]
|:                        obj->[prop: {[destroy]-inst->[rays] obj->[tumor]}]  }]
|:             out->[cg:G]}.
A = _V1
L = _V2
B = _V3
R = _V4
C = _V5
E = _V6
R2 = _V7
G = {[want]-
        agnt->[doctor]
        obj->[prop: {[destroy]-
                        obj->[tumor]
                       inst->[rays: _*mr1]-attr->[powerful] , ,}*mr1]-car->
            [prop: {[truth: false]-mod->[prop: {[go]-
                                                agnt->[rays]
                                               dest->[direction: _*mr3]-qty->[one] , ,}] ,}*mr4]-sub->
             [solution: {[go]-
                        agnt->[weakRays: _*mr2]
                         man->[simultanous]
                        dest->[direction: _*mr5]-qty->[many] , ,
```

```
                    &
                   [rays]-part->[weakRays: _*mr2] ,}] , , ,}

yes
| ?-
```

Définition : *Projection d'un GC sur un autre*

La projection d'un GC g1 sur un autre g2 est l'appariement avec la contrainte suivante : g1 doit être isomorphe à un sous-graphe de g2; la structure de g1 doit être une sous-structure de g2. Les concepts de g1 sont mis en correspondance avec ceux du sous-graphe de g2. ♦

Exemple

```
| ?- quest.
|: cgOperations::{[project]-
|:                  in1->[cg:{[work]-agnt->[person:kabbaj]
|:                                    obj->[software]}]
|:                in2->[cg:{[cptTeam:tm34]-chrc->[work]-dur->[week]
|:                                                  agnt->[person]
|:                                                 obj->[software:prologPCG]-attr->[interest:high]}]
|:                  out->[cg:ResProj]}
|:  .
ResProj = {[match(work,work): match(,)]-
                                  obj->[match(software,software): match(,prologPCG)]
                                 agnt->[match(person,person): match(kabbaj,)] ,}

yes
| ?- quest.
|: cgOperations::{[project]-
|:                in1->[cg:{[work]-agnt->[person:kabbaj]
|:                                  obj->[software]-qlty->[prototype]}]
|:                in2->[cg:{[cptTeam:tm34]-chrc->[work]-dur->[week]
|:                                                  agnt->[person]
|:                                                 obj->[software:prologPCG]-attr->[interest:high]}]
|:                   out->[cg:Res]}.

no
| ?- quest.
|: cgOperations::{[project]-
|:                    in1->[cg:{[think]-agnt->[woman:W]
|:                                      obj->[prop:{[person:W]-poss->[car]}]}]
|:                    in2->[cg:{[think]-agnt->[woman:fati]-ageOg->[age:34] ,
|:                                    obj->[prop:{[tired]-pat->[person:fati]-poss->[car]}]}]
|:                     out->[cg:G]}
|:  .
W = _V1
G = {[match(think,think): match(,)]-
          obj->[match(prop,prop): {[match(person,person): match(,fati)]-poss->[match(car,car): match(,)] ,}]
          agnt->[match(woman,woman): match(,fati)] ,}

yes
```

Définition : *EstPlusGeneral*

EstPlusGeneral (subsume) est une projection modulo l'opération *EstPlusGeneralConc* sur les concepts. L'opération EstPlusGeneral vérifie si un GC g1 est plus général qu'un graphe g2. ♦

Définition : *EstPlusGeneralConc*

Le concept c1 est plus général que le concept c2 si le type de c1 est plus général que celui de c2 et le référent de c1 est plus général que celui de c2. Si les deux référents sont des GC, alors il faut y appliquer l'opération *EstPlusGeneral.* ♦

Exemple

```
| ?- quest.
|: cgOperations::{[subsume]-
|:                  in1->[cg:{[work]-agnt->[person]
|:                                        obj->[software]}]
|:                in2->[cg:{[cptTeam:tm34]-chrc->[work]-dur->[week]
|:                                                  agnt->[man:kabbaj]
|:                                                obj->[progLanguage:prologPCG]-attr->[interest:high]}]
|:                      out->[cg:G]}.
G = {[work]-
        obj->[progLanguage: prologPCG]
        agnt->[man: kabbaj] ,}

yes
| ?- quest.
|: cgOperations::{[subsume]-
|:                  in1->[cg:{[work]-agnt->[person:kabbaj]
|:                                        obj->[software]}]
|:                  in2->[cg:{[cptTeam:tm34]-
|:                                  chrc->[work]-dur->[week]
|:                                        agnt->[man]
|:                                        obj->[progLanguage:prologPCG]-attr->[interest:high]}]
|:                       out->[cg:G]}.

no
| ?- quest.
|: cgOperations::{[subsume]-
|:                       in1->[cg:{[think]-agnt->[woman:W]
|:                                        obj->[prop:{[person:W]-poss->[car]}]}]
|:                       in2->[cg:{[think]-agnt->[woman:fati]-ageOg->[age:34] ,
|:                                      obj->[prop:{[tired]-pat->[person:fati]-poss->[car]}]}]
|:                        out->[cg:G]}.
W = _V1
G = {[think]-
        obj->[prop: {[person: fati]-poss->[car] ,}]
        agnt->[woman: fati] ,}

yes

| ?- quest.
|: cgOperations::{[subsume]-
|:                       in1->[cg:{[think]-agnt->[woman:W]
|:                                        obj->[prop:{[person:W]-poss->[car]}]}]
|:                       in2->[cg:{[think]-agnt->[woman:fati]-ageOg->[age:34] ,
|:                                      obj->[prop:{[tired]-pat->[person]-poss->[car]}]}]
```

```
|:                          out->[cg:G]}.

no
| ?- quest.
|: cgOperations::{[subsume]-
|:                          in1->[cg:{[think]-agnt->[woman:W]
|:                                   obj->[prop:{[person:W]-poss->[car]}]}]
|:                          in2->[cg:{[think]-agnt->[woman:fati]-ageOg->[age:34] ,
|:                                   obj->[prop:{[tired]-pat->[person:pratt]-poss->[car]}]}]
|:                          out->[cg:G]}.

no
```

Définition : *Unification de deux graphes*

L'unification de deux graphes g1 et g2, *UnifierGC*, est la projection modulo la jointure des concepts. Des référents-variables de g1 et g2 peuvent être instanciées suite à l'unification. ♦

Remarques :

- Notre définition de l'unification est basée sur celle de Fargues et al., [Fargues et al., 86]. Elle est utilisée par l'interpréteur de Prolog+CG et aussi dans Synergy (en particulier la reconnaissance d'un message).
- L'opération *UnifierPGC* est similaire à *UnifierGC* sauf qu'elle ne produit pas d'effet de bord sur les référents-variables de g1 et g2 et elle produit un GC g3 qui est le résultat de l'unification des deux graphes en entrée.

Exemples

```
| ?- quest.
|: cgOperations::{[unifyCG]-
|:                          in1->[cg:{[speak]-agnt->[person:P]-member->[soccerGroup:[P|Q]]}]
|:                          in2->[cg:{[speak]-agnt->[person:S]-member->[soccerGroup:[martin,jo,najib]]}]}
|:   .
P = martin
Q = [jo,najib]
S = martin

yes
| ?- quest.
|: cgOperations::{[unifyCG]-
|:                          in1->[cg:{[speak]-agnt->[person:P]-member->[soccerGroup:[P|Q]]}]
|:                          in2->[cg:{[speak]-agnt->[person:jo]-member->[soccerGroup:[martin,jo,najib]]}]}
|:  .

no
| ?- quest.
|: cgOperations::{[unifyCG]-  in1->[cg:{[speak]-agnt->[person]}]  in2->[cg: X]}.
X = {[speak]-agnt->[person],}


| ?- quest.
|: cgOperations::{[unifyCG]-
|:                          in1->[cg:{[work]-agnt->[person:P]  obj->[software]}]
```

```
|:                    in2->[cg:{[cptTeam:tm34]-chrc->[work]-
|:                                                    dur->[week]
|:                                                    obj->[software:prologCG]-attr->[interest:high] ,
|:                                                   agnt->[person:kabbaj]}]}.
P = kabbaj


| ?- quest.
|: cgOperations::{[unifyCG]-
|:                    in1->[cg: {[speak]-agnt->[man:Man]-poss->[car:Car]  ageOf->[age:30]}]
|:                      in2->[cg: {[speak]-
|:                                  agnt->[man:karin]-poss->[car:fordLx]  ageOf->[age:45] ,
|:                                  agnt->[man:farene]-poss->[car:volvo]  ageOf->[age:30]}]}
|:   .
Man = farene
Car = volvo

yes
| ?- quest.
|: cgOperations::{[unifyCG]-
|:                    in1->[cg: {[speak]-agnt->[man:Man]-poss->[car:Car]  ageOf->[age:30]}]
|:                      in2->[cg: {[speak]-
|:                                  agnt->[man:farene]-poss->[car:volvo]  ageOf->[age:30] ,
|:                                  agnt->[man:karin]-poss->[car:fordLx]  ageOf->[age:45]}]}
|:   .
Man = farene
Car = volvo

yes
| ?- quest.
|: cgOperations::{[unifyCG]-
|:                  in1->[cg:{[cptTeam:tm34]-chrc->[work]-
|:                                                    dur->[week]    agnt->[person:kabbaj]
|:                                                    obj->[software:prologCG]-attr->[interest:high]}]
|:                   in2->[cg: {[work]-agnt->[person:P]  obj->[software]}] }.

no
| ?- quest.
|: cgOperations::{[unifyCG]-
|:                    in1->[cg:{[think]-agnt->[person:P]
|:                                  obj->[prop: {[go]-agnt->[person:P]  dest->[city:rabat]}]}]
|:                  in2->[cg:{[think]- obj->[prop: {[go]-agnt->[person:rachid]  dest->[city:rabat]}]
|:                                  agnt->[person:rachid]-poss->[car]}] }
|:   .
P = rachid

yes
```

Définition : *Canonique*

L'opération Canonique teste si un GC g respecte les canons des types des concepts et des relations composant le graphe. L'opération vérifie que chacun des canons est plus général que g. ♦

Remarque : Dans leur définition d'une opération similaire à *Canonique*, Mugnier et Chein [Mugnier et Chein, 93] considèrent uniquement les canons des relations. Une définition plus complète de cette opération devrait tenir compte également des canons des types de concepts.

En effet et comme Sowa et Way l'ont formulé, *"canonical graphs show external pattern of relationships that must be attached to concepts of a given type"* [Sowa et Way, 86]. Par exemple, si on considère le graphe [Humain]<-agnt-[Porter]-obj->[Habit] comme étant le canon du type "Porter", alors le graphe [Porter]-obj->[Veste] est non canonique, il est par contre canonique si on considère uniquement le canon de la relation "obj" : [Action]-obj->[Entite].

Définition : *ContractionComplete*

L'opération ContractionComplete teste si un graphe g1 est plus général que g2, si le test réussit, elle contracte g1 de g2 en conservant la connexion de g2. ♦

Définition : *ContractionPartielle*

L'opération ContractionPartielle identifie les parties du graphe g1 qui sont contenues dans g2 et tente ensuite de les contracter de g2. Pour cela, elle effectue l'appariement de g1 à g2 modulo "EstPlusGeneralConc", si l'appariement réussit, elle contracte la partie appariée de g2, en conservant la connexion de g2. ♦

Exemples

```
| ?- quest.
|: cgOperations::{[partialContract]-
|:                    in1->[cg:{[person]-poss->[hair]-attr->[color:red]}]-entryPoint->[concept: ![person]] ,
|:                      in2->[cg:{[sing]-
|:                                  obj->[song:waterlo]
|:                                 agnt->[person]-
|:                                          ageOf->[age:45]
|:                                           poss->[hair]-attr->[color:red] ,
|:                                            nat->[nationality:uk] ,
|:                                 loc->[hotel:malaparta]    }]- entryPoint->[concept: ![person]] ,
|:                      out->[cg:Gres]}
|:  .
Gres = {[sing]-
            obj->[song: waterlo]
            agnt->[person *mr3]-
                        ageOf->[age: 45]
                         nat->[nationality: uk] ,
            loc->[hotel: malaparta] ,}

yes
| ?- quest.
|: cgOperations::{[partialContract]-
|:                          in1->[cg:{[person]-poss->[hair]-
|:                                                    attr->[cutShape:short]
|:                                                   attr->[color:red]      }]-entryC->[concept: ![hair]] ,
|:                           in2->[cg:{[sing]-
|:                                       obj->[song:waterlo]
|:                                       agnt->[person]-
|:                                                  ageOf->[age:45]
|:                                                 poss->[hair]-attr->[color:red] ,
|:                                                  nat->[nationality:uk] ,
|:                                       loc->[hotel:malaparta]       }]-entryPoint->[concept: ![hair]] ,
```

```
|:                              out->[cg:Gres]}.
Gres = {[sing]-
            obj->[song: waterlo]
            agnt->[person  *mr3]-
                              ageOf->[age: 45]
                               nat->[nationality: uk] ,
            loc->[hotel: malaparta] ,}

| ?- quest.
|: cgOperations::{[partialContract]-
|:                              in1->[cg:{[stand]-pat->[person]-poss->[hair]-attr->[color:red] , ,
|:                                        loc->[door]}          ]-entryPoint->[concept: ![hair]] ,
|:                               in2->[cg:{[sing]-
|:                                        obj->[song:waterlo]
|:                                        agnt->[person]-
|:                                                 ageOf->[age:45]
|:                                                 poss->[hair]-
|:                                                          attr->[cutShape:short]
|:                                                          attr->[color:red] ,
|:                                                 nat->[nationality:uk] ,
|:                                        loc->[hotel:malaparta]}      ]-entryPoint->[concept: ![hair]] ,
|:                              out->[cg:Gres]}.
Gres = {[sing]-
            obj->[song: waterlo]
            agnt->[person  *mr3]-
                              ageOf->[age: 45]
                              poss->[hair  *mr5]-attr->[cutShape: short] ,
                               nat->[nationality: uk] ,
            loc->[hotel: malaparta] ,}

yes
| ?- quest.
|: cgOperations::{[completeContract]-
|:                              in1->[cg:{[stand]-pat->[person]-poss->[hair]-
|:                                                          attr->[color]
|:                                                         poss->[density]
|:                                                          attr->[cutShape:short]}]-
|:                                        entryPoint->[concept: ![hair]] ,
|:                               in2->[cg:{[sing]-
|:                                        agnt->[person  *mr1]-
|:                                                 ageOf->[age:45]
|:                                                  poss->[hair]-
|:                                                          attr->[color:red]
|:                                                          attr->[cutShape:short]-attr->[model:m34]
,
|:                                                           poss->[density] , ,
|:                                        loc->[hotel:malaparta] &
|:                                      [stand]- pat->[person *mr1]
|:                                                behind->[girl]}]-entryPoint->[concept: ![hair]],
|:                                 out->[cg:Gres]}
Gres = {[stand]-
            pat->[person  *mr2]-
                              ageOf->[age: 45]
                              poss->[hair  *mr4]-
                                        attr->[color: red]
                                        attr->[cutShape: short*mr6]-attr->[model: m34] , , ,
             behind->[girl] ,
        &
        [sing]-
            agnt->[person  *mr2]
            loc->[hotel: malaparta] ,}
```

Définition : *ContractDef*

L'opération *ContractDef* contracte d'un graphe G la définition D d'un type, D est donc un sous-graphe de G. L'opération utilise pour cela l'opération *ContractionComplete*. Le type défini remplace son super-type dans G. ♦

Exemple

```
| ?- quest.
|: cgOperations::{[contractDef]-
|:                          in1->[cg: {[eat]-agnt->[person: jack]-has->[sex: masc] ,
|:                                                  obj->[apple] }]
|:                           in2->[concept: ! [person: jack] ]
|:                           in3->[type: man]
|:                           out->[cg:CGres]    }.
CGres = {[eat]-agnt->[man: jack]
                  obj->[apple] }

yes
```

A.3 Définition algorithmique des opérations sur les GC

L'algorithme de l'opération d'appariement, décrit ci-dessous, "couvre" les algorithmes des opérations dérivées; il constitue un algorithme "générique", paramétré par le nom de l'opération particulière d'appariement (comme jointure, specialiser, generaliser, etc.). En effet, le "squelette" de l'algorithme est commun à toutes les opérations, les éléments qui distinguent les opérations, comme l'appariement particulier des types et des référents des concepts, sont traités par des structures de cas (cas .. parmi ..).

Structures utilisées dans l'algorithme de l'operation apparier

```
CorefMatchL = ListOf  {  /* utilisée pour enregistrer l'appariement des co-références entre concepts  */
                         Var1, Var2, Var3 : String
                         } ;

CMatchL = ListOf  {  /*** utilisée pour enregistrer des informations sur l'appariement des concepts ***/
                     ConcMatched1, ConcMatched2, ResOfMatch : Concept ;
                     ToBeDone, MatchedLocally : Boolean ;
                    } ;

RMatchL = ListOf  {  /*** utilisée pour enregistrer des informations sur l'appariement des relations ***/
                     RelMatched1, RelMatched2 : Relation ;
                    } ;
```

L'algorithme de l'opération générique Match est composé essentiellement des procédures/fonctions suivantes (Figure A.5) :

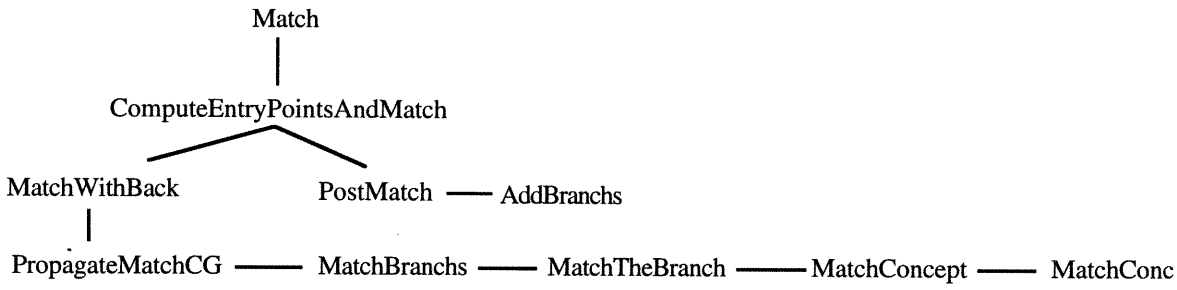


Figure A.5 : Composition de l'algorithme de l'opération Match

match: Si l'opération est membre de l'ensemble {Projection, Subsume, UnifyCG, unifyPGC, CompleteContraction}, alors la contrainte suivante (de base) doit être vérifiée avant de commencer l'appariement : toutes les relations du GC G1 doivent être aussi des relations de G2. En effet, si cette contrainte n'est pas vérifiée, ce n'est pas la peine de considérer l'appariement (spécialisé selon l'une des opérations de l'ensemble précédent).

```
procedure match(inOut OperCG: String, E1:Concept, G1:CG, E2: Concept, G2:CG; G3:CG;
                                                                    BRes:Boolean) is
   if  OprCG in  {Projection, Subsume, UnifyCG, unifyPGC, CompleteContraction}  and
       not bagInclusion(G1.Relations, G2.Relations)  then  BRes := false
              /*** G1.Relations retourne les noms des relations utilisées dans G1 ***/
   else
     {CorefMatchL := nil ;
      BRes := computeEntryPointsAndMatch(OprCG, E1, G1, E2, G2, G3) ;
      FreeSpace(CorefMatchL) }
   endIf
end Appariement.
```

Note:
bagInclusion({agnt, agnt, agnt, obj, instr, obj}, {agnt, obj, instr}) va retourner false.
bagInclusion({agnt, agnt, agnt, obj, instr, obj}, {agnt, obj, agnt, instr, agnt, obj, attr})
va retourner true.

computeEntryPointsAndMatch : cette opération détermine en premier les points d'entrée et initie ensuite l'appariement.

Pour déterminer les points d'entrée, cette opération considère en premier le cas où les points d'entrée sont fournis en argument, ensuite deux cas particuliers où un des graphes contient un seul concept. Sinon, elle considère les deux "heuristiques" pour déterminer les points d'entrée (par identité des référents ou par appariement des co-référents). Sinon, elle considère l'identification des points d'entrée par appariement de deux relations. Pour les opérations dérivées de la projection, puisque toute relation du premier graphe doit s'apparier

avec une relation du second, la procédure prend donc la première relation du premier graphe et cherche une relation du second qui puisse s'y apparier.

Le dernier cas s'applique pour la jointure, la spécialisation et la généralisation : apparier deux concepts des deux graphes.

```
function computeEntryPointsAndMatch(inOut OperCG: String, E1:Concept, G1:CG, E2: Concept,
G2:CG;                                                                    G3:CG): Boolean is
   BRes := false ;
   CMatchL := nil ;
   RMatchL := nil ;
   if E1 <> nil then BRes := matchWithBack(OprCG, E1, E2, nil, nil, G1, G2, G3)
                                                                  /** entry points given **/
   elseIf G1 is a CG that corresponds to one concept with no relation then /** special case **/
      { while BRes = false AND there is a concept C2 not considered yet in G2 do
            BRes := matchConcept(OprCG, G1, C2, G3)
        endWhile ;
        BRes := BRes AND postMatch(OprCG, G1, G2, G3) }
   elseIf G2 is a CG that corresponds to one concept with no relation then /** special case **/
      { while BRes = false AND there is a concept C1 not considered yet in G1 do
            BRes := matchConcept(OprCG, C1, G2, G3)
        endWhile ;
        BRes := BRes AND postMatch(OprCG, G1, G2, G3) }

   elseIf not OprCG in {generalize, funcGeneralize} AND
          [varCoref(C1, G1, C2, G2) OR identInd(C1, G1, C2, G2)] then
                    /** varCoref to determine entry points by co-reference matching ***/
                    /** identInd to determine entry points by identity of referents ***/
       BRes := matchWithBack(OprCG, C1, C2, nil, nil, G1, G2, G3)

   elseIf OprCG in {project, unifyCG, unifyPCG, subsume, cplteContract} then
                                    /** to determine entry points by relation matching **/
       { get the first branch Cs1 -Rel-> Ct1 from G1 ;
         while BRes = false AND there is a branch Cs2 -Rel-> Ct2 not considered yet from G2 do
            BRes := matchWithBack(OprCG, Cs1, Cs2, Ct1, Ct2, G1, G2, G3)
         endWhile }

   else                  /** to determine entry points by relation matching **/
     { while BRes = false AND there is a branch Cs1 -Rel-> Ct1 not considered yet from G1 do
         while BRes = false AND there is a branch Cs2 -Rel-> Ct2 not considered yet from G2 do
            BRes := matchWithBack(OprCG, Cs1, Cs2, Ct1, Ct2, G1, G2, G3)
         endWhile
       endWhile ;
       if BRes = false AND OprCG in {maximalJoin, specialize, generalize} then
                                  /** to determine entry points by concept matching **/
          {while BRes = false AND there is a concept C1 not considered yet in G1 do
             while BRes = false AND there is a concept C2 not considered yet in G2 do
                BRes := matchConcept(OprCG, C1, C2, G3)
             endWhile
           endWhile ;
           if BRes = true And OprCG in {maximalJoin, specialize}
              then addBranchs(OprCG, G1, G2, G3) endIf }
       endIf }
   endIf ;
   FreeSpace(CMatchL) ; FreeSpace(RMatchL) ;
   return BRes ;
end computeEntryPointsAndMatch.
```

matchWithBack : elle initie l'appariement des deux graphes. Si l'appariement échoue elle restitue l'ancien contexte. Ceci est important spécialement pour l'unification qui peut modifier les deux graphes durant le traitement.

```
function matchWithBack(inout OprCG: String; Cs1, Cs2, Ct1, Ct2: Concept; G1, G2, G3: CG): Boolean
                                                                                              is
var BRes : Boolean ; AncCorefL: .. as CorefMatchL ;
    CopyList(AncCorefL, CorefMatchL) ; /** Copy each element of CorefMatchL in AncCorefL **/
    AncConcsG1 , AncConcsG2 : List of Concept ;

     if OprCG = unifyCG then  { CopyList(G1.Concs, AncConcsG1) ;
                                CopyList(G2.Concs, AncConcsG2) }
     endIf ;
     if matchConcept(OprCG, Cs1, Cs2, G3) AND (Ct1 = nil OR matchConcept(OprCG, Ct1, Ct2, G3))
        AND  propagateMatchCG(OprCG, G1, G2, G3)  then  BRes := true
     else /*** restore the old state of the variables ***/
          {FreeSpace(CorefMatchL) ; CopyList(AncCorefL, CorefMatchL) ;
           FreeSpace(CMatchL) ; FreeSpace(RMatchL) ; FreeSpace(G3) ;
           if OprCG = unifyCG then
              { UpdateConcs(G1.Concs, AncConcsG1) ;  UpdateConcs(G2.Concs, AncConcsG2) }
           endIf ;
           Bres := false}
     endIf ;
     FreeSpace(AncCorefL) ;
     if OprCG = unifyCG then {FreeSpace(AncConcsG1) ; FreeSpace(AncConcsG2) } endIf ;
     return  BRes ;
end matchWithBack.
```

propagateMatchCG : Le traitement général de cette opération consiste en une boucle qui vérifie à chaque itération s'il existe un concept c1 de g1 qui a été apparié avec un concept c2 de g2 mais les relations attachées aux deux concepts n'ont pas été encore appariées. Si un tel couple <c1,c2> existe, la fonction tente d'apparier les relations en entrée et ensuite les relations en sortie des deux concepts.

Si l'appariement réussit, l'opération considère le traitement qui vient après (comme l'ajout au graphe résultat des branches spécifiques aux deux graphes en entrée). Ce traitement dépend du type de l'appariement.

L'opération *propagateMatchCG* considère un cas particulier qui peut survenir lors de l'appariement de GC imbriqués. Ce cas suscite une situation de compromis entre le traitement des co-références et l'application récursive de l'appariement : durant l'appariement, si les référents de deux concepts à apparier sont des GC, alors on devrait apparier ces deux GC. Le problème est qu'un appariement immédiat de ces GC peut ne pas bénéficier de l'existence des co-références (rappelons que celles-ci peuvent déterminer les points d'entrée) puisqu'elles peuvent être appariées *après* l'appariement des deux graphes.

Une solution au problème serait de donner priorité à l'appariement des co-références et reporter l'appariement des graphes imbriqués à la fin de l'appariement des graphes courants. Le problème avec cette solution est que l'appariement des "concepts-composés" (qui ont des GC comme référent) n'est alors pas utilisé pour guider et orienter l'appariement des graphes.

La solution que nous proposons à ce "dilemme" est de reporter le plus possible l'appariement des graphes imbriqués : si les référents à apparier sont des GC, alors nous spécifions seulement qu'ils doivent être appariés et dans la procédure *propagateMatchCG* nous considérons en premier les concepts "simples" (qui n'ont pas des GC comme référents). Si dans l'itération courante il n'y a pas de concepts simples dont on peut propager l'appariement, la procédure cherche alors un couple de concepts-composés à apparier et effectue ensuite l'appariement de leur référents.

```
function propagateMatchCG(in OprCG: String;  G1, G2, G3: CG): Boolean     is
    BRes := true ;
    while BRes = true AND
          [ (there is an element E in CMatchL : E. ToBeDone = true And E.MatchedLocally = false)   OR
            (there is an element E in CMatchL : E.ToBeDone = false) ]          do
          { if E.ToBeDone = false  then
                  BRes := matchConceptS(OprCG, E.ConcMatched1, E.ConcMatched2, G3)   endIf ;

            if  BRes = true  then
              { BRes := matchBranchs("outComeBranch", OprCG, E.ConcMatched1, G1,
                                                      E.ConcMatched2, G2, E.ResOfMatch, G3) AND
                        matchBranchs("inComeBranch", OprCG, E.ConcMatched1, G1,
                                                      E.ConcMatched2, G2, E.ResOfMatch, G3) ;
                 if BRes = true  then  E.MatchedLocally := true endIf  }
            endIf }
     endWhile ;
     return  BRes AND postMatch(OprCG, G1, G2, G3) ;
end propagateMatchCG.
```

matchBranchs : Cette opération cherche les branches attachées au concept c1 de g1 qui peuvent s'apparier à des branches attachées au concept c2 de g2, c1 et c2 ont déjà été apparié.

```
function matchBranchs(BranchDirection: String; OprCG: String; C1: Concept; G1: CG; C2: Concept;
                      G2: CG; C3: Concept; G3: CG): Boolean   is
    /*** matchBranchs searchs for the branchs of the concept C1 in G1 that could match with branchs of C2
         in G2.  ***/
    /*** BranchDirection = outComeBranch  (C1, Rel, Ca1)  ===>  look for  C1 -Rel-> Ca1
         BranchDirection = inComeBranch  (C1, Rel, Ca1) ===> look for Ca1 -Rel-> C1  *****/

    BRes := true ;
    while BRes AND there is a branch BranchDirection(C1, Rel, Ca1) in G1 such that :
       there is no element E in RMatchL with E.RelMatched1 = Rel-of-BranchDirection(C1, Rel, Ca1), do :
      /*  Rel-of-BranchDirection(C1, Rel, Ca1) because a CG can have many relations with the same name */
       BRes := matchTheBranch(BranchDirection(C1, Rel, Ca1), OprCG, C2, G2, C3, G3);
    endWhile ;
    return BRes ;
end  matchBranchs.
```

matchTheBranch : Dans les cas des branches en sortie des concepts C1 et C2 respectivement, cette opération cherche une branche C2 -Rel-> Ca2 dans G2 qui puisse s'apparier avec la branche C1 -Rel-> Ca1 fournie en argument. Le même traitement s'applique pour le cas des

branches en entrée à C1 et à C2. Nous commentons le corps de cette opération en considérant le cas des branches en sortie.

La relation Rel dans la branche recherchée C2 -Rel-> Ca2 doit avoir la même direction et le même nom que celle de la branche C1 -Rel-> Ca1 fournie en argument et la branche ne doit pas être déjà appariée; le concept Ca1 ne doit pas être déjà apparié (à moins qu'il le soit avec le concept Ca2) et il en est de même pour le concept Ca2.

Si la branche est trouvée, l'opération tente d'apparier les concepts Ca1 et Ca2 (s'ils ne le sont pas déjà). Si l'appariement réussit, il est sauvegardé dans la liste CMatchL et la branche C3 -Rel-> Ca3, C3 et Ca3 sont respectivement le résultat de l'appariement de C1 avec C2 et de Ca1 avec Ca2, est ajoutée au graphe résultat (à moins que le type d'appariement soit un élément de l'ensemble {unifyCG, specialize, funcSpecialize, cplteContract, partialContract}, le graphe résultat n'est pas construit dans ce cas).

Si la branche n'est pas trouvée, le traitement dépendra alors du type de l'appariement :

→ si l'appariement est funcMaximalJoin, funcSpecialize ou funcGeneralize et il y a une branche C2 -Rel-> Ca2, alors l'appariement des deux graphes est impossible selon les points d'entrée actuels.

→ si l'appariement est project, subsume, unifyCG, unifyPCG ou cplteContract, alors l'appariement des deux graphes est impossible selon les points d'entrée actuels.

→ pour les autres cas, ignorer la branche C1 -Rel-> Ca1 de G1.

```
function matchTheBranch(BranchDirection(C1, Rel, Ca1); OprCG: String; C2: Concept; G2: CG;
                                                        C3: Concept; G3: CG): Boolean is
    BRes := false ;
    while BRes = false AND
          there is a branch BranchDirection(C2, Rel, Ca2) not considered from G2 AND
          there is no element E in RMatchL with E.RelMatched2 = Rel of BranchDirection(C2, Rel, Ca2)
          do :
        BRes := there is <Ca1, Ca2, _, _, _> in CMatchL  OR
                (no there is <Ca1, _, _, _, _> in CMatchL and
                 no there is <_, Ca2, _, _, _> in CMatchL
                 and matchConcept(OprCG, Ca1, Ca2, G3) ) ;
    endWhile ;
    if BRes then
       { insert <Rel-of-BranchDirection(C1, Rel, Ca1), Rel-of-BranchDirection(C2, Rel, Ca2)> in RMatchL ;
         if not OprCG in {unifyCG, specialize, funcSpecialize, cplteContract, partialContract} then
                {Localise <Ca1, Ca2, Ca3, _, _> in CMatchL ;
                 addto G3 the new branch BranchDirection(C3, Rel, Ca3) endIf ;
         return true }
    elseIf not [ (OprCG in {funcMaximalJoin, funcSpecialize, funcGeneralize} and
                           there is a branch BranchDirection(C2, Rel, Ca2) from G2 ) OR
                 OprCG in {project, subsume, unifyCG, unifyPCG, cplteContract} ] then return true
    else return false ;
end matchTheBranch.
```

postMatch : Si l'appariement correspond à match, generalize, funcGeneralize ou partialContract, alors il n'y a pas de traitement après l'appariement. Si l'appariement correspond à maximalJoin ou funcMaximalJoin alors toute branche spécifique à l'un des deux graphes en entrée est ajoutée

au graphe résultat. Si l'opération est specialize ou funcSpecialize alors les branches spécifiques au premier graphe sont ajoutées au second. Si l'opération est un élément de l'ensemble {project, unifyCG, unifyPCG, subsume, cplteContract} alors postMatch vérifie que toutes les branches du premier graphe sont appariées à des branches du second.

```
function postMatch(inout OprCG: String; G1, G2, G3: CG): Boolean is
   BRes := true ;
   case OprCG :
      in {maximalJoin, funcMaximalJoin} : addBranchs(maximalJoin, G1, G2, G3) ;
      in {specialize, funcSpecialize} : addBranchs(specialize, G1, G2, nil) ;
      in {project, unifyCG, unifyPCG, subsume, cplteContract} :
                  BRes := it is true that all the relations of G1 are matched ; /** for any relation of G1
                                                                                  there is an element in RMatchL **/
   endCase ;
   return BRes ;
end postMatch.

function addBranchs(inout OprCG: String; G1, G2, G3: CG): Boolean is
   case OprCG
     = maximalJoin : { addBranchsOfCG(G1, 1, G3) ; /** addBranchsOfCG G1 to G3 **/
                       addBranchsOfCG(G2, 2, G3) /** addBranchsOfCG G2 to G3 **/ }
     = specialize : { for each element <C1, C2, C3, _, _> of CMatchL do :
                       Replace in G1 the concept C1 by the concept C3; /** specialize G2 concepts **/
                    endFor ;
                    addBranchsOfCG(G2, 2, G1) }
   endCase ;
   return true ;
end addBranchs.

function matchConcept(inout OprCG: String; C1, C2: Concept; G3: CG): Boolean is
   if cat(C1.PtVal) = "cg" And cat(C2.PtVal) = "cg" then /*** .. only concept types are matched, referent
                                                                 matching will be done later ***/
      if matchTyp(OprCG, C1.Type, C2.Type, T3) then
         { Create a concept C3 with C3.type := T3 ;
           Insert <C1, C2, C3, false, false> in CMatchL ;
           if not OprCG in {unifyCG, specialize, funcSpecialize, cplteContract, partialContract}
                 then addto G3 the new concept C3 endIf ;
           return true }
      else return false
   elseIf matchConc(OprCG, C1, C2, C3) then
        { Insert <C1, C2, C3, true, false> in CMatchL ;
          if not OprCG in {unifyCG, specialize, funcSpecialize, cplteContract, partialContract}
                 then addto G3 the new concept C3 endIf ;
          return true }
   else return false
   endIf ;
end matchConcept.
```

L'opération **matchConc** apparie les composantes des deux concepts entre elles. Par exemple:

```
function matchTyp(in OprCG: String; Type1, Type2: Type; out Type3:Type): Boolean is
   case OprCG :
      in {maximalJoin, funcMaximalJoin, specialize, funcSpecialize, unifyPCG, unifyCG} :
         return maxComSubType(Type1, Type2, Type3) And Type3 <> "Absurd"
```

```
        in OprCG in {subsume, partialContract, cplteContract} : return isSuperType(Type1, Type2)
        in {generalize, funcGeneralize} :
            return minComSuperType(Type1, Type2, Type3)  And Type3 <> "Universal"
    endCase
end matchTyp.

function  matchRef(in OprCG: String; Ref1, Ref2: Referent; out Ref3:Referent; T3:Type) : Boolean  is
    BRes := true ;
    case OprCG :
        in {maximalJoin, funcMaximalJoin, specialize, funcSpecialize, unifyPCG} :
                                                        {BRes := joinRef(Ref1, Ref2, Ref3) }
        = unifyCG :  {BRes := unifyRef(Ref1, Ref2) }
        in {subsume, partialContract, cplteContract} : BRes := EqRef(Ref1, Ref2)
        in {generalize, funcGeneralize} :  X := generalizeRef(Ref1, Ref2, Ref3)
    endCase ;
    return BRes ;
end matchRef.
```

Le détail de l'appariement des composantes d'un concept (pour le cas de Synergy, nous avons également l'appariement des états et des valeurs du concept, en plus du type et du référent) est fourni dans les manuel de Prolog+CG et de Synergy [Kabbaj, 95, 96].

Les operations ExpandDef et ContractDef

```
procedure ExpandDef(in R: Referent; inout G : CG;  out BRes: Boolean)  is
    Conc := LocalizeConc(R, G) ;        /*** localize in G a concept Conc with a referent R ***/
    GDef := GetDefType(Conc.Type) ;  /** get the definition body of Conc.Type **/
    ConcSuper := LocalizeConc("super", GDef) ; /** Localize the super-concept in the definition **/
    Conc.Type :=   ConcSuper.Type ;              /** Change the type **/
    match(specialize, Conc, G, ConcSuper, GDef, _, _) ;
end ExpandDef.

procedure ContractDef(in T: Type ;  R: Referent; inout G: CG ; out BRes: Boolean) is
    GDef := GetDefType(T) ;
    ConcSuper := LocalizeConc("super", GDef) ;
    Conc := LocalizeConc(R, G) ;
    matchCG(cplteContract, ConcSuper, GDef, Conc, G, _, BRes) ;
    if BRes then { Conc.Type := T; contractG(G) }  endIf ;
    Free(CMatchL)  ;  Free(RMatchL) ;
end contractDef.
```

La prochaine opération utilise la notion de *branche pendante*. Nous allons définir cette dernière en premier.

Définition : *Branche pendante*

Une branche S -rel-> C de G est pendante si C a une seule relation en entrée et aucune relation en sortie ou, si S a une seule relation en sortie et aucune relation en entrée. ♦

contractG(G : CG) : en se basant sur un appariement antérieur de deux graphes, cette opération tente de façon itérative à éliminer des branches-appariées du graphe G tout en

maintenant la connexion de ce dernier. Ainsi à chaque itération, *contractG* cherche en premier à éliminer une branche pendante car cette dernière ne peut causer une rupture du graphe. Le concept qui sera enlevé du graphe suite à l'élimination de la branche doit être identique au concept qui lui est apparié (sinon on aurait une perte d'information). Si l'opération ne trouve pas une branche pendante (avec la condition sur le concept à enlever), elle cherche alors une branche qui ne produirait pas une rupture du graphe si elle en est éliminée.
L'opération cherche en premier les branches pendantes afin d'éviter, le plus possible, l'appel à la procédure qui vérifie la rupture du graphe.

```
procedure contractG(inout G: CG) is
   BranchToEliminate := true ;
   while BranchToEliminate do
      BranchToEliminate := DanglingBranch(G, Branch)  Or  NoRuptureInG(G, Branch) ;
      if BranchToEliminate  then  EliminateBranch(G, Branch) endIf ;
   endWhile ;
end contractG.


function  DanglingBranch(in G: CG, out (S-R->C): Relation) : Boolean is
   return  There is a branch S-R->C in G with :
         the branch has been matched (i.e. there is an element E in RMatchL with E.RelMatched2 = R)  And
         [ (infPreserved(C)  And  C has only one income branch  And  no outcome branch)  Or
           (infPreserved(S)  And  S has only one outcome branch  And  no income branch) ]
end  DanglingBranch.


function  NoRuptureInG(in G: CG; out (S-R->C): Relation) : Boolean is
   Found := There is a branch S-R->C in G that has been matched
   Found := Found  And  There is another branch S -R'- C no matter the direction ;
   if  Found  And  there is no other branch S -R'- C no matter the direction)  then
      { LConcAdjs := AdjacentsOf(S) ; /*** the list of the concepts related to S **/
        Remove(C, LConcAdjs) ;
        Found := IsThereAPath(LConcAdjs, C, (S) ) } /*** (S) represents a list of one element S ***/
   endIf ;
   return Found ;
end NoRuptureInG.

function   IsThereAPath(inout LConcs: ListOf Concept, in C: Concept,
                                                    inout LConcsVisited: ListOf Concept): Boolean is
   Found := false ;
   while Not Found and Not LConcs.IsEmpty  do :
        if member(C, LConcs) then Found := true
        else
          {RemoveHead(LConcs, Cd) ;
           Insert(Cd, LConcsVisited) ;
           for each element Elem of AdjacentsOf(Cd)  do :
                 if Not member(Elem, LConcsVisited) And Not member(Elem, LConcs)
                       then Insert(Elem, LConcs)
                 endIf
           endFor }
        endIf
   endWhile ;
   return Found ;
end  IsThereAPath.
```

Annexe B

Définition algorithmique du cycle de vie d'un concept

Comme nous l'avons souligné dans la section 3.5.3, l'interpréteur de Synergy simule, pour un GC actif, l'exécution parallèle des cycles de vie de ses concepts actifs. Cet annexe complète la section 3.5.3 en fournissant l'interprétation associée à certains états d'un concept.

Nous définissons les procédures/fonctions qui interprétent les états "?", "@prc", ActivateValue (qui est associée à la mise en activation d'un concept), les deux tests de terminaison (pour les primitives et les GC) et autres procédures annexes.

procedure State? :
→ Si un concept C est à l'état "?" (il a donc reçu une demande de "déterminer et évaluer ta valeur") et ne possède pas une valeur et son type n'est ni une opération primitive ni un type défini, alors une propagation-arrière de la demande via les liens "out" sans l'attribut "couper-propagation-arrière ; \" est effectuée et le concept est mis à l'état d'attente "@".
→ Si le concept possède une valeur, on considère alors ses préconditions (s'il n'en a pas, on considère directement son activation) et le concept est mis à l'état "vérifier-préconditions ; @prc". En particulier, une propagation arrière de la demande est initiée pour toute précondition qui n'a pas de valeur, qui n'a pas l'attribut "couper-propagation-arrière ; \" et qui est à l'état repos.

```
procedure State?(in C: Concept, Ctxt: Context) is
   if C.GetVal(Ctxt) = nil And Not (IsDefined(C.Type) Or IsSuperType("PrimitiveOperations", C.Type))
        then
        /*** forward propagation, if possible, through out links ***/
       { for any concept Ci income to C: Ci -R-> C with R.Type = "out without \" do :
             if Ci.State = 'o' then Ci.SetState(Ctxt, '?', "trigger") endIf
        endFor ;
        C.SetState(Ctxt, '@') }
    elseIf Not (there is an income for C : Cj -R-> C with R.Type = "in" Or "grd")
        /*** there is no pre-conditions for C, so its value will be actived immediatly ***/
             then ActivateValue(C, Ctxt)
   else
       {for any concept Ck income to C : Ck -R-> C with Ck.State = 'o' do :
            if (IsPrecond(R, C.Type) And Ck.GetVal(Ctxt) = nil)
                then Ck.SetState(Ctxt, '?', "trigger")
            endIf
        endFor
        C.SetState(Ctxt, '@prc') }
   endIf
end State?.
```

```
function IsPrecond(in R: Relation, T: TypeConc) : Boolean is
   RelType := R.Type ;
   if RelType = "grd without \" then return true
   elseIf RelType = "in without \" And Not IsSuperType("Lazy", T) then return true
   else return false
end IsPrecond.
```

procedure State@prc : Un concept C à l'état "vérifier-préconditions ; @prc" qui a une précondition non satisfaite est mis à l'état "attente-préconditions ; @prc@", autrement (toutes ses préconditions sont vérifiées) on considère son activation. Il importe de noter que l'état "@prc@", aussi bien que "@", sont des états d'attente, aucun traitement ne leur est associé (l'interpréteur ne considère pas les concepts qui sont dans l'un des deux états). C'est l'interprétation des autres concepts qui peut impliquer un changement d'état d'un concept en attente et le rendre (de nouveau) "visible" à l'interpréteur.

```
procedure State@prc(in C: Concept, Ctxt: Context) is
   Found := false ; /** we search for a precondition that isn't satisfied ... ***/
   while concept C has an income (Cj -R-> C) not treated yet And Not Found do
      Found := Not IsCondSatisfied(Cj, R, C.Type, Ctxt) ;
   endWhile ;
   if Found then C.SetState(Ctxt, '@prc@') else ActivateValue(C, Ctxt) endIf ;
end State@prc.

function IsDefined(in T : TypeConc) : Boolean is
   /** It checks for a type T if it is defined, i.e. if there is in the Long-Term Memory a definition for T. */

function IsCondSatisfied(in C: Concept, R: Relation, T: Type, Ctxt: Context) : Boolean
   if R.Type = "grd" then return C.GetVal(Ctxt) = true
   elseIf R.Type = "in" And Not IsSuperType("Lazy", T) then return C.GetVal(Ctxt) <> nil
   else return true ;
end IsCondSatisfied.
```

procedure SetState : elle change l'état d'un concept et considère le cas d'un concept co-référent à l'état "?" sans l'attribut "couper-propagation-arrière ; \" et sans valeur. Si un tel cas se présente et si le concept référé est à l'état "o", ce dernier est mis alors à l'état "?", la demande est ainsi propagée en arrière du concept co-référent au concept référé, via le lien implicite de co-référence.

Si le concept co-référent a été mis à l'état "?" suite à une affectation de valeur, alors le concept référé doit être mis au courant afin qu'il change d'état s'il y a lieu, par exemple, de l'état "o" ou "@" à "?". Il y a toutefois deux exceptions à ce changement :

a) si le concept co-référent C est un paramètre en sortie dans la description d'un concept C1 et si l'argument A correspondant est lié à C1 par "out,/" : C1 -out,/-> A ; alors A est mis à l'état "o" et non à l'état "?".

b) si le concept co-référent C réfère à un concept-fonction, l'état de ce dernier n'est pas alors modifié, le concept est toutefois inséré dans la liste LForward.

```
procedure SetState(in C: Concept, Ctxt: Context, S: State, AffectType: String) is
   C.State := S ;
   if S = '?' And Not C.attr\ then
      { C.GetVal(Ctxt) ;  /** to determine the refered concept if it isn't already **/
        if AffectType = 'trigger' And C.Value.State = 'o' then C.Value.State := '?' /*** C.Value returns a
                                                                    concept, since C is a co-reference. **/
        elseIf AffectType = 'affect' then
           if IsParameter(C) And Out/(C) then C.Value.State := 'o'
           elseIf [ IsSuperType("Procedure", C.Value.Type) Or
                    IsSuperType("PrimitiveOperations", C.Value.Type) ] then
                 Insert(C.Value, LForward)
           elseIf C.Value.State = 'o' or '@' then C.Value.State := '?'
           endIf
        endIf }
   endIf
end SetState.

function IsParameter(in C: Concept, Dir: {"in", "out"} ) : Boolean is
      /*** if Dir isn't specified as argument, is has a null string " " by default ***/
   if (Dir = " " And C.Referent has the prefix "inX/" Or "outX/" where X is nothing or an integer) Or
      (Dir <> " " And C.Referent has the prefix Dir + "X/" ) then return true
   else return false
   endIf;
end IsParameter.
```

procedure ActivateValue : Cette procédure prépare et initie l'activation du concept, selon son type.

→ Si le type correspond à une primitive alors ActivateValue appelle ComposePrimitiveCall pour préparer l'appel de la primitive en déterminant les arguments en entrée et en sortie, ActivateValue élimine ensuite les valeurs des entrées, s'il y a lieu (aussi bien les arguments que les booléens de contrôle) et puis elle initie l'exécution "physique et effective" de la primitive. La durée d'exécution est ensuite enregistrée (dans l'implantation actuelle, c'est toujours 2) dans la liste LPrimitivesInExec avec les résultats et autres informations concernant le concept. La liste LPrimitivesInExec permet à l'interpréteur Synergy d'appliquer une interprétation parallèle à l'exécution séquentielle des primitives.

→ Si le concept n'a pas une valeur mais son type est défini, alors sa valeur est créée par instanciation et est ensuite activée. Par ailleurs, un concept qui a déjà un GC comme valeur est directement activé.

→ Dans le cas où le concept possède une valeur qui n'est pas un GC, son activation est instantanée et on passe donc directement à ses post-conditions.

```
procedure ActivateValue(in C: Concept, Ctxt: Context) is
   CVal := C.GetVal(Ctxt) ;
   if IsSuperType("PrimitiveOperations", C.Type) then
      {ComposePrimitiveCall(C, Ctxt, LValsIn, LArgsOut) ;
       for any concept X income to C with : X -inORgrd,f -> C , do :
          Eliminate the "real" value of X ; /** "real" to note that X can be a co-reference, in this case
                                                we eliminate the value of the refeered concept ***/
       endFor ;
       ExecutePrimitive(<C.Type, LValsIn, LArgsOut>, TimeExec, LComputedValues) ;
```

```
      Insert(<Ctxt, C, LComputedValues, LArgsOut, TimeExec + 1>, LPrimitivesInExec) ;
      C.SetState(Ctxt, '!@') }
   elseIf CVal = nil Or IsSuperType("Procedure", C.Type) then
      {CreateInstance(C, Inst, FormalParam) ;
       ActivateCG(C, Ctxt, Inst, FormalParam)   }
   elseIf CVal is a CG then  ActivateCG(C, Ctxt, CVal, ' ')
   else /** the concept has a value that isn't a CG : its activation takes 0 time**/
      PostConditions(C, Ctxt)
   endIf;
end ActivateValue.

procedure CreateInstance(in C: Concept, out Inst: CG, FormalParam: String) is
   FormalParam := ConceptDef(C.Type).Referent ;
   Inst := CopyCG(ConceptDef(C.Type).Value) ; /** a copy of the definition body of C.Type **/
   .. other technical instructions ..
end CreateInstance.
```

procedure ComposePrimitiveCall : dans sa première partie, cette procédure considère certains cas particuliers qui nécessitent un traitement spécial. A titre d'exemple, nous fournissons le cas SetState et MessageDispatcher. La seconde partie concerne le cas général : les valeurs des arguments en entrée sont déterminées et les arguments en sortie sont identifiés.

```
procedure ComposePrimitiveCall(in C: Concept , Ctxt: Context ;
                               out LValsIn, LArgsOut: ListOf Concept) is
   case C.type
      = "SetState" :
        {Get the income argument ArgIn of C ;
         S := ArgIn.GetVal(Ctxt) ;
         Get the outcome argument ArgOut of C ;
         ArgOut.Value.SetState(Ctxt, S, 'trigger') }
        ...

      = "MessageDispatcher" :
        {Super := LocaliseConc("super", Ctxt.ActiveCG) ;
         if Super <> nil then
              { Declare the concept QSuper ;
                QSuper.Referent := "super.messageQueue" ;
                QSuper.Value := nil ;
                QSuper.GetVal(Ctxt) ;
                QueueSuper := QSuper.Value ;
                Eliminate the concept QSuper }
         else QueueSuper := nil
         endIf ;
         LServices := nil ;
         for any outcome argument ArgOut of C do InsertAtEnd(ArgOut, LServices) endFor ;
         Get the income argument ArgIn of C ;
         MessageDispatcher(Ctxt, LServices, ArgIn.Value, Super, QueueSuper)
         }
         ...

   else
     {LValsIn := nil ;
      for any income argument ArgIn of C do : InsertAtEnd(ArgIn.GetVal(Ctxt), LValsIn) endFor ;
      LArgsOut := nil ;
      if (C has not an outcome argument) And (C has an outcome parameter) then
          InsertAtEnd(C, LArgsOut)
      else for any outcome argument ArgOut of C do
                 InsertAtEnd( DefConcOfVal(ArgOut, Ctxt) , LArgsOut) endFor ;
```

```
      endIf }
    endCase ;
fin PrepareAppelPrimitif.

function LocaliseConc(in R: Referent, G: CG): Concept is
        Find a concept in CG with R as a referent. If find, return the concept else return nil.

function DefConcOfVal(in C: Concept, Ctxt: Context) : Concept is
   if Not IsCoReferent(C.Referent) then return C
   else {C.GetVal(Ctxt) ; return C.Value /** C.Value corresponds in this case to the refered concept **/ }
end DefConcOfVal.

function IsCoReference(in R: Referent ; out CoRef: Referent) : Boolean is
   /*** CoRefForm is "ident.", or "ident.ident. .... .ident" or "." where ident is an instance or a variable
                identifier ***/
   if R has the form Variable=CoRefForm then {CoRef := CoRefForm ; return true}
   elseIf R is a CoRefForm alors {CoRef := R ; return true}
   else return false
end IsCoReference.
```

procedure ActivateCG : elle est composée des parties suivantes

→ Les états des arguments en entrée à C, qui étaient à l'état "?" durant le déclenchement de C, sont transmises aux états des paramètres correspondants. De même, les paramètres en sortie de C qui correspondent à des arguments en sortie à l'état "@", sont mis aussi à l'état "?".

→ Transmission des "adresses" : un paramètre réfère à l'argument correspondant à moins que ce dernier ne soit lui-même un co-référent, auquel cas le paramètre réfère au concept référé par l'argument.

→ Les liens conditionnels fonctionnels sont considérés à ce stade; la valeur du booléen est éliminée.

→ Une structure "context" est formée pour gérer l'activation du GC, à moins que ce dernier ait déjà un contexte (à cause des références multiples à une même description).

→ le concept est mis à l'état "!@".

```
procedure ActivateCG(in C: Concept, Ctxt: Context, G: CG, DefParam: String) is

      /** Step 1 : transmission of the states of some in/out arguments of C ***/
   if Find(<C, LRangeArgsIn> , Ctxt.LConceptsWithArgs)  then
      for any element J of LRangeArgsIn do Pj.State := '?' endFor
   endIf ;
   for any outcome argument ArgOut of C with ArgOut.State = '@' do :
       Get from G the parameter ParamOut that corresponds to ArgOut ;
       ParamOut.State := '?'
   endFor ;

      /** Step 2 : transmission of "address" ; the parameters must refer to the correspondant arguments
                   or to the refered concepts (if the argument is itself a refered concept) **/
   for any parameter P in G do :
     {if the argument A that corresponds to P exist And IsCoReference(A.Referent, RefA) then
        { Xr := RefA ;
          if IsParameter(A) then { A.GetVal(Ctxt) ; P.Value := A.Value } endIf }
      else { Xr := " ." ; P.Value := A }
```

```
        endIf;
        if P.Referent <> " "  then  P.Referent := P.Referent + "=" + Xr
        else  P.Referent := P.Referent + Xr
        endIf }
    endFor ;

    if (C has not an outcome argument)  And  (C has one outcome parameter P)  then
       { P.Value := C ;
         P.attr/ := false ;
         P.attr\ := false }
    endIf ;

       /** Step 3 : Eliminate for C the value of the pre-conditions-with-consumption **/
    for any concept X income to C with : X -grd,f -> C   ,  do :
         Eliminate the "real" value of X ;
    endFor ;

                    /** Step 4 : create a context record for CG **/
    if Not  (there is an element Ctxt1 in LActiveCtxts Or LNewActiveCtxts  with  Ctxt1.ActiveCG = G)
      then
       {Create NewContext ;
       with NewContext do
           FormalParam := DefParam ;
           ActiveCG := G ;
           PreviousContext := Ctxt ;
           LConceptsWithArgs := nil ;
       endWith ;
       Insert(NewContext, LNewActiveCtxts)  }
    endIf ;

    C.SetState(Ctxt, '!@') ;
end ActivateCG.
```

Alors que les procédures *ActivateValue* et *ActivateCG* constituent le prologue de l'activation d'un concept, les procédures *TestTerminationCGs* et *TestTerminationPrimitives* constituent l'épilogue.

```
procedure TestTerminationCGs is
   for any element Ctxt of LActiveCtxts other than the two first elements (i.e. LTM & WM)  do :
       C := ConcOf(Ctxt.ActiveCG) ;
       if  Not IsActiveCG(C, Ctxt.ActiveCG)  then
        { PrvCtxt := Ctxt.PreviousContext ;
          Eliminate from PrvCtxt.LConceptsWithArgs the element <C, L> if it is there ;
          if  IsSuperType("Procedure", C.Type)   then
             { Free the space occuped by the graph Ctxt.ActiveCG  ;
               if C.Value = Ctxt.ActiveCG  then  C.Value := nil  endIf  }
          endIf ;
          Free the context record Ctxt ;
          PostConditions(C, PrvCtxt)  }
       endIf ;
   endFor ;
end TestTerminationCGs.
```

function IsActiveCG : Un GC est actif s'il contient au moins un concept qui suscite une activité (un changement) dans le graphe. Les concepts à l'état "?", "@prc", "!@" ou "@nda" vérifient cette condition. Il en est de même pour les concepts dans la liste LForward. Enfin, on

peut considérer qu'un GC G est actif s'il contient un concept à l'état "attente-valeur ; @" qui soit co-référent sans attribut "couper-propagation-avant ; /" (ainsi un changement dans le concept référé est "perçu" par le concept co-référent) et que son concept référé soit dans un contexte actif qui n'est pas imbriqué dans G.

```
function IsActiveCG(in C: Concept; G: CG) : Boolean is
   return
      There is a concept Cj in G with : (Cj.State = ? or @prc or !@ or @nda) Or member(Cj, LForward)
               Or (Cj.State = '@' And Not Cj.attr/ And IsCoReference(Cj.Referent, RefCj) And
                     there is no concept Cc in G with identBase(RefCj, _) <> " " And
                                          testReferent(Cc.Referent, identBase(RefCj, _))   And
                     IsActiveContext(C, Cj.Value) )
end IsActiveCG.

function IsActiveContext(C1, C2 : Concept) : Boolean is
   G := ContextDef(C2) ;
   return
      there is an element Ctxt1 in LActiveCtxts with Ctxt1.ActiveCG = G And
      there is a concept Cj in G with : (Cj.State = ? or @prc or @nda) Or member(Cj, LForward)
                                          Or (Cj.State = '!@' And Not AntConc(Cj, C1))
               Or (Cj.State = '@' And Not Cj.attr/ And IsCoReference(Cj.Referent, RefCj) And
                     there is no concept Cc in G with identBase(RefCj, _) <> " " And
                                          testReferent(Cc.Referent, identBase(RefCj, _))   And
                     IsActiveContext(C, Cj.Value) )
end contexteActif.

function AntConc(in Cj, C1: Concept) : Boolean is
               /** it enables a move from a context to its antecedent */
   if C1 = Cj then return true
   elseIf ContextDef(C1) is Long-Term Memory then return false
   else return AntConc(Cj, ConcOf(ContextDef(C1)) ;
                              /*** ConcOf(CG) returns the concept that has the CG as value ***/
end AntConc.

function testReferent(in R: string, IdentB: string): Boolean is
   if R.Referent = IdentB or "IdentB=..." or "Variable=IdentB" then return true
   else return false
   endIf ;
end testReferent.
```

procedure TestTerminationPrimitives : cette procédure parcourt la liste LPrimitivesInExec à la recherche des primitives qui ont terminé leur exécution ; leur durée est à présent égale à 0. Rappelons qu'à chaque cycle de l'interpréteur, la durée de chaque primitive est réduite de 1 (une unité de temps Synergy). Pour un concept primitif C qui vérifie la condition ci-dessus, la procédure considère le cas où une de ses sorties "out" est fonctionnelle (avec l'attribut "f") et le concept relié possède une valeur, le concept C est alors mis en état "attente-affectation ; @aft". Le cas général est toutefois l'initiation de l'affectation, réalisée par la procédure initieAffect.

```
procedure TestTerminationPrimitives is
   for any element <Ctxt, C, LComputedValues, LArgsOut, 0> in LPrimitivesInExec do :
      if C has an outcome X with : C - out,f -> X And X.GetVal(Ctxt) <> nil
         then C.SetState(Ctxt, '@aft') else InitiateAffect(C, Ctxt) endIf
```

```
   endFor
end TestTerminationPrimitives.

procedure State@aft(in C: Concept, Ctxt: Context) is
   if for any outcome concept Co, of C, with : C -out,f -> Co) And val(Co, Ctxt) = nil
        then InitiateAffect(C, Ctxt) endIf ;
end State@aft.
```

procedure InitiateAffect : InitiateAffect extrait le couple associé à C de la liste d'attente LPrimitivesInExec et, pour chaque valeur-résultat, ajoute dans la liste des affectations LAffectation un élément indiquant l'affectation à effectuer. Le concept C est ensuite mis en état "attente-fin-affectation ; @nda". Si le concept ne possède pas d'arguments en sortie, ses post-conditions sont alors directement considérées.

```
procedure InitiateAffect(in C: Concept, Ctxt: Context) is
   Remove( <Ctxt, C, LComputedValues, LArgsOut, _>, LPrimitivesInExec) ;
   if LComputedValues <> nil then
      {for any I eme element Vi of LComputedValues And I eme element Ai of LArgsOut do
            Insert( <Ctxt, C, Vi, Ai> , LAffectation) ;
       endFor ;
       C.SetState(Ctxt, '@nda') }
   else PostConditions(C, Ctxt) }
end InitiateAffect.
```

procedure AffectationManagment : si plusieurs affectations existent pour un concept donné, une seule est considérée, les autres sont reportées aux prochains cycles de l'interpréteur. Si aucune affectation ne reste en suspens pour un concept C, alors ses post-conditions sont considérées.

```
procedure AffectationManagment is
   LAlreadyOneAffec := nil ;
   for any element <Ctxt, C, Val, Co> of LAffectation And Not member(Co, LAlreadyOneAffec) do :
       { Remove(<Ctxt, C, Val, Co>, LAffectation) ;
         Co.SetVal(Val, Ctxt) ;
         if (C -out,/-> Co) then Co.SetState(Ctxt, 'o') else SetState?(Co, Ctxt) endIf ;
         Insert(Co, LAlreadyOneAffec) ;
         if Not Find( <_, C, _, _> , LAffectation) /** all the affectation related to C are done **/
         then PostConditions(C, Ctxt)
         endIf }
   endFor;
end AffectationManagment.

procedure PostConditions(in C: Concept; Ctxt: Context) is
   C.SetState(Ctxt, 'o') ;
   PropagateForward(C, Ctxt) ;
end PostConditions.

procedure PropagateForward(in C: Concept; Ctxt : Context) is
   for any concept Cj with : C -R-> Cj And (R.Type = "inORgrdORnext without /") do :
    { if Cj.State = 'o' And
            (R.Type = "inORnext" OR (R.Type = "grd" And C.GetVal(Ctxt) = true) )
            then Cj.SetState(Ctxt, '?', 'trigger')
      elseIf Cj.State = '@prc@' And (R.Type = "in" Or
                                     (R.Type = "grd" And C.GetVal(Ctxt) = true) )
```

```
            then  Cj.SetState(Ctxt, '@prc')
        endIf ;
        if Not IsSuperType("PrimitiveOperations", Cj.Type) And R.Type = "in without /" And
                                                          Cj.State <> '!@'   then
            if Find( <Cj, LRangeArgsIn>, Ctxt.LConceptsWithArgs)  then
            /*** the couple <Cj, LRangeArgsIn> is used to enable state transmission from
                  argument to parameter , especially for income argument with "?" state ***/
               Insert(R.Range, LRangeArgsIn)
            else Insert(<Cj, [R.Range]>, Ctxt.LConceptsWithArgs)
            endIf
        endIf }
    endFor
end PropagateForward.
```

procedure SetState? : Si le concept C est une fonction ou si c'est un paramètre qui réfère à un concept-fonction, C est alors inséré dans la liste LForward.

```
procedure SetState?(in C: Concept; Ctxt: Context)  is
    if IsParameter(C)  And  (IsSuperType("Procedure", C.Value.Type) Or
                             IsSuperType("PrimitiveOperations", C.Value.Type) )  then
        { C.State := '?' ;
          Insert(C.Value, LForward) }
    elseIf  IsSuperType("Procedure", C.Type) Or IsSuperType("PrimitiveOperations", C.Type) then
          Insert(C, LForward)
    else C.SetState(Ctxt, '?', 'affect')
    endIf ;
end SetState?.
```

Annexe C

Définition algorithmique de la procédure de résolution de la co-référence

La procédure de résolution de la co-référence est utilisée par l'interpréteur Synergy pour déterminer le concept référé par une co-référence. Nous présentons ici la définition détaillée de cette opération principale dans Synergy.

function GetVal(**in** C: Concept, Ctxt: Context): Object. Si le référent du concept C n'est pas une co-référence alors GetVal retourne la valeur de C. Si c'est une co-référence qui a été résolue, GetVal retourne la valeur du concept référé (rappelons qu'on retient dans le champ Val du concept co-référent le pointeur au concept référé), autrement GetVal initie la résolution de la co-référence. GetVal considère le cas particulier d'un accès direct à la mémoire à long terme.

```
function GetVal(in C: Concept, Ctxt: Context): Object is
    isReference := IsCoReference(C.Referent, CoRef) ;
   if Not isReference then return C.Value
   elseIf C.Value <> nil then return C.Value.Value
   else /** C is a co-reference concept evaluated for the first time **/
     { IdentB := FirstIdent(CoRef, RestCoRef) ;
        /*** LTMemory is the concept that represents the LTMemory ***/
      if IdentB = LTMemory.Referent then
        ConcRef := SearchInEmbddCG(RestCoRef, (), (LTMemory.Value))
      else ConcRef := SearchInContextHier(CoRef, Ctxt.PreviousContext, (Ctxt.ActiveCG)) ;
                                                  /** the search begins from the current context **/
      endIf ;
      if ConcRef = nil then return ErrMessage(CoRef, "can't be resolved.")
      else {C.Value := ConcRef ; /*** ConcRef is the refeered concept ***/
            Insert(<ConcRef, C>, LCoReferences) ; /*** see the remark below ***/
            return ConcRef.Value }
      endIf }
   endIf
end GetVal.

function FirstIdent(in R: Referent ; out RestCoRef: Referent ) : String is
   /*** the function returns the first identifier of the sequence and returns in RestCoRef the rest of the co-
      reference R ***/
```

Remarque : Tout concept peut être référé et s'il est éliminé (par exemple, il se trouve dans un contexte actif qui a terminé son exécution et qui est détruit), on pourrait avoir une référence pendante ! Pour éviter ce problème, GetVal enregistre dans la liste LCoReferences tout lien de

co-référence <ConceptRéféré, ConceptRéférence>. Avant d'éliminer un concept, on consulte la liste LCoReferences pour défaire toute co-référence avec le concept éliminer.

function SearchInContextHier(**in** R: Referent, Ctxt: Context, StackPrevCG: ListOf CG): Concept .

Le rôle de cette fonction est de "monter" dans la hiérarchie des contextes afin de localiser un concept dont le référent est identique au premier identificateur de la référence R (premier paramètre de la procédure SearchInContextHier). Il importe de noter que la hiérarchie est composée des contextes actifs (le contexte courant étant le paramètre Ctxt) et des GC imbriqués (la pile représentée par le paramètre StackPrevCG). Une première "montée" est effectuée dans la pile StackPrevCG et si le concept référé ne se trouve pas dans un des GC de la pile, une seconde "montée" est effectuée dans la hiérarchie des contextes, en commencant avec le contexte courant Ctxt.

La procédure considère deux cas particuliers : le premier élément de la co-référence est "self" ou le paramètre formel d'une définition.

```
function  SearchInContextHier(in R: Referent, Ctxt: Context, StackPrevCG: ListOf CG): Concept  is
   Found := false ;
   while Not Found  And  Not StackPrevCG.IsEmpty  do :
      Conc := SearchInEmbddCG(R, Ctxt, StackPrevCG) ;
      if Conc <> nil  then  Found := true  else  Pop(StackPrevCG)
   endWhile ;
   if Not Found then
     { CurrentContext := Ctxt ;
       while  Not Found  And  CurrentContext <> nil  do :
          IdentBase := FirstIdent(R, RestCoRef) ;
          if  IdentBase = "self"  And  ConcOf(CurrentContext.ActiveCG).Referent = "super"  then
             /*** the current context isn't the refeered by self, so we continue with the "father" ! ***/
                CurrentContext := CurrentContext.PreviousContext
          else
             { if  IdentBase = "self"  Or   IdentBase = CurrentContext.FormalParam  then
                /** the current context is the "good" context, so initiate the search in the embedded CG **/
                    Conc := SearchInEmbddCG(RestCoRef, CurrentContext.PreviousContext,
                                                          (CurrentContext.ActiveCG) )
               else  /*** initiate the search from the current context ***/
                    Conc := SearchInEmbddCG(R, CurrentContext.PreviousContext,
                                                          (CurrentContext.ActiveCG) )
               endIf ;
               if Conc <> nil  Or  IdentBase = CurrentContext.FormalParam  then  Found := true
               else  CurrentContext := CurrentContext.PreviousContext
               endIf }
           endIf
       endWhile }
   endIf ;
   return Conc ;
end  SearchInContextHier.
```

function SearchInEmbddCG(**in** R: Referent, Ctxt: Context, StackPrevCG: ListOf CG): Concept :

Le rôle principal de cette fonction est de "descendre" dans l'imbrication des GC afin de localiser le concept référé. La fonction vérifie en premier si le contexte courant G (le sommet

de StackPrevCG) contient un concept dont le référent est identique au premier élément de la co-référence R (le premier paramètre de la fonction), ou si G contient un concept-super (un concept dont le référent est "super"). Sinon, la fonction retourne "nil" pour indiquer que le concept ne peut être localisé au sein du contexte G.

Si toutefois, la fonction trouve un tel concept dans G alors la recherche dans les GC imbriqués commence, selon la séquence des éléments qui composent le co-référent R. La recherche est réalisée par une boucle qui tente, à chaque cycle, de localiser dans le contexte courant (qui est un GC) un concept dont le référent est identique à l'élément courant de la co-référence.

Rappelons que le contexte courant correspond à la valeur du concept précédemment localisé. Si ce dernier n'a pas de valeur, elle est alors créée par instanciation de la définition de son type.

Durant la recherche, la fonction considère le cas particulier où un des concepts à localiser est en fait une co-référence (un paramètre ou un concept dont le référent est une variable avec une co-référence comme valeur). Dans ce cas, la co-référence est concaténée à la partie (de la co-référence R) qui reste à résoudre.

```
function SearchInEmbddCG(in R: Referent, Ctxt: Context, StackPrevCG: ListOf CG): Concept is
   LocalCtxt := StackPrevCG.Sommet ;
   IdentB := FirstIdent(R, RestCoRef) ;
   if there is a concept C in LocalCtxt with :
         testReferent(C.Referent, IdentB) Or testReferent(C.Referent, "super") then
      /*** we search in the embedded CG, according to the sequence of identifiers in R ***/
      InitialReference := R ;
      if testReferent(C.Referent, IdentB) then R := RestCoRef endIf ;
      while R <> " " do :
         if IsCoReference(C.Referent, CoRefC) then /*** CoRefC = "." ==> C is a parameter ***/
           {if CoRefC = "." And R = " " then C := C.Value
            elseIf CoRefC = "." then C := SearchInEmbddCG(R, Ctxt, (C.Value.Value) )
            else { NewCoRef := AppendCoRef(CoRefC, R) ;
                   C := SearchInContextHier(NewCoRef, Ctxt, StackPrevCG) }
            endIf ;
            R := " " }
         else
           {if C.Value = nil then
               { CreateInstance(C, Inst, Param) ;
                 C.Value := Inst }
            elseIf C.Value is not a CG then ErrMessage(R, ".. a CG is expected as a value. ")
            endIf ;
            LocalCtxt := C.Value ;
            StackPrevCG.Push(LocalCtxt) ;
            IdentB := FirstIdent(R, RestCoRef) ;
            if there is a concept C in LocalCtxt with : testReferent(C.Referent, IdentB) then
               R := RestCoRef
            elseIf there is a concept C in LocalCtxt with : testReferent(C.Referent, "super") then true
            elseIf InitialReferent <> R then ErrMessage(R, "can't be resolved.")
            else {C := nil ; R := " " }
            endIf }
         endIf
      endWhile ;
   else C := nil
```

```
    endIf ;
    return  C ;
end SearchInEmbddCG.

function AppendCoRef(in Ref1, Ref2: Referent) : Referent  is
    /** the function appends two coreferent. If we have for instance : ident1.ident2  &  ident3.ident4 then =>
                                                                        ident1.ident2.ident3.ident4
        If we have  ident1.  &  ident2.ident3  => ident1.ident2.ident3  ****/
```

Remarque : l'implantation en Visual C++ des algorithmes présentées (en partie) dans cette thèse ainsi que de l'environnement graphique a été réalisée par Mr. Rouane Khalid.

Annexe D

Utilisation du langage Synergy dans le projet SAFARI

Cet annexe présente une publication concernant l'utilisation de Synergy dans le cadre du projet SAFARI ; un projet de développement de Systèmes Tutoriels Intelligents. Nous avons utilisé Synergy pour une modélisation et une simulation visuelle orientée agent de l'unité des soins intensifs (un module de SAFARI) ainsi que pour la formulation du processus de génération de cours à partir du curriculum. Les deux applications sont totalement différentes, illustrant ainsi le caractère général de notre langage. En effet, la première application est formulée selon une approche concurrente orientée objet : des agents, en l'occurrence le patient et l'infirmière, communiquent par envoi de messages et l'infirmière a pour tâche d'évaluer l'état du patient, la seconde application est réalisée selon une approche par propagation de l'activation dans un graphe.

La publication paraîtra dans :

ITS'96 : Third Intern. Conf. on Intelligent Tutoring Systems, Montréal, June 12-14, 1996, Springer-Verlag.

The use of a semantic network activation language in an ITS project

Adil KABBAJ, Khalid ROUANE, Claude FRASSON

Université de Montréal, Département d'informatique
et de recherche opérationnelle
C.P. 6128, Succ. Centre-Ville, Montréal, H3C 3J7

fax : 514 343 5834
E-mail : kabbajad@iro.umontreal.ca

Abstract We report the use of a graphical multi-paradigm language in SAFARI ; an environment for the development of ITSs. The language, called *Synergy* uses "active semantic networks" to manipulate both declarative and procedural knowledge.
After an overview of the language, we present its use in some modules of SAFARI, especially for a visual agent-oriented modeling of the Intensive Care Unit and for course generation from a curriculum. The first application illustrates the use of Synergy as a tool for task cognitive analysis.
Synergy could be used in other modules of SAFARI, in other ITS' projects and in other artificial intelligence and computer domains.

Key words Programming models integration, semantic networks activation, intelligent tutoring systems, intensive care unit, course generation.

1 Introduction

Several kinds of knowledge are used in intelligent tutoring system (ITS) and to deal with, different processing paradigms are often used in one intelligent tutoring system (ITS). Communication and share of knowledge, intra and inter ITSs become difficult.
In an attempt to solve this problem, we have developed a graphical multi-paradigm language, called *Synergy*. The language uses semantic network formalism to represent both declarative and procedural knowledge and to integrate many programming models (sequential/parallel procedural, functional, object-oriented and graph-activation models).
Synergy and its environment have been implemented in Visual C++ and used in SAFARI [Gecsei and Frasson, 94], an ITS' development project underway at the University of Montreal.
The language, i.e. the PC executable code, the examples' directory and the User' manual, will be available in summer 1996 at the following address :

ftp.iro.umontreal.ca /pub/its/pub_safari/kabbaj/Synergy .

The paper is organized as follows : after a brief overview of Synergy (§ 2), we present two applications, visual agent-oriented modeling of the Intensive Care Unit (§ 3) and

course generation from a curriculum, taking into account the student model (§ 4). The two applications are taken from SAFARI project. We then report some related and future works (§ 5 and 6) and terminate with a conclusion (§ 7).

2 An overview of Synergy

This section introduces basic elements of Synergy: the conceptual environment of the language, the semantic network (SN) structure and the activation of a SN.

2.1 The conceptual environment of Synergy

The conceptual environment of Synergy is composed of a *long-term memory* (LTM) and of a *working memory* (WM). Synergy' WM provides the working space where the user specifies his requests. Requests can be evaluated in parallel.
Synergy' LTM represents the base where the knowledge about a domain (or some related domains) is stored. As in object-oriented languages, Synergy' LTM is represented as a generalization graph. When a new application is created, Synergy provides its built-in generalization graph composed of primitive types (types not defined in Synergy) and some predefined types (defined in Synergy but provided to the user as predefined). A user can define new types and add them to the generalization graph. He can also specify instances of types.

Primitive/predefined types "Data" primitive types are Number, String, Boolean, List, Window and Semantic network. Among "operation" primitive types, Synergy provides affectation (:=), arithmetic, relational, boolean, I/O, list, and SN operations (unification, fusion, generalization, subsumption, contraction, etc.).
Among "operation" predefined types, Synergy provides communication operations (AcceptMessage, Send, SendAck, Receive, WaitMessage, MethodMgr). Communication operations are used in sequential/concurrent object-oriented and/or agent-oriented applications.

Defined types A user can define a new type as a specialization of existing types (primitive, predefined or defined types). Of interest here the four primitive types : ProcedureActivity, ProcessActivity, StrictActivity and LazyActivity. They are sub-types of the primitive type Activity. ProcedureActivity and ProcessActivity reflect the difference in the life-time of an activity while StrictActivity and LazyActivity reflect the difference in the evaluation mode of the arguments of an activity (if it has). A new type can have some of those primitives as super-types and so, it will be interpreted according to their semantics.
Figure 1.a gives the definition of a new type IsAdult. "sp" link represents the specialization relation. Type "IsAdult" is a procedure with a strict evaluation. In the definition of "IsAdult", the concepts [Age :in/] and [Boolean :out/] represent input and output parameters respectively. Figure 1.b illustrates how the user can specify instances of types, either by giving a specific description of the instance (as for MyTable in figure 1.b) or just by specifying that the individual is an instance of a type (as for Tble34). For the later case, Synergy will create, when necessary, the description of the instance by an instanciation of its type's definition.

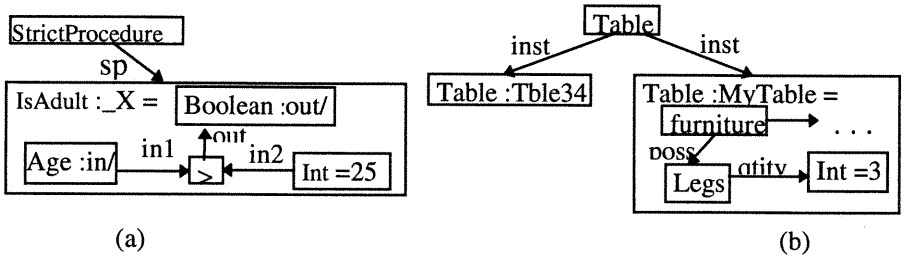


Figure 1: Definition of a type and description of instances

2.2 Synergy's semantic network structure

Semantic network (SN) is the basic structure in Synergy : the LTM, a request in the WM, the body of a concept type definition and of an instance description are SN.
Our SN structure is derived from conceptual graphs formalism [Sowa, 84] which is proposed as a synthesis of many SNs formalisms.
In Synergy, a *semantic network* is a labeled directed graph where nodes represent *concepts* and arcs *relations*.

Concept A concept specifies a referent with its type, value (or description) and state. A general form of a concept is : [Type :Referent =Value #State]. In general, when one concept' component isn't specified, a default is taken.
Concept' type can be primitive or defined. *Concept' referent* can be an instance identifier, a variable (that could have a referent as a value) or a co-reference, i.e. a reference to another concept. For instance, the concept [Man :Daniel.eyes.canOpen] is a co-reference to a concept with referent "canOpen" which is in the description of "eyes" which is itself in the description of "Daniel" (Figure 2).

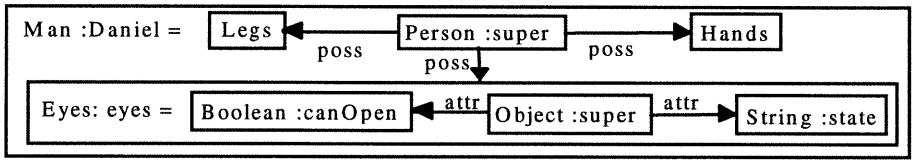


Figure 2: Co-reference to an embeeded concept

Concept' value can be a number, a boolean, a string, a list or a SN. *Concept' state* specifies the state of the concept. Basic states of a concept are "steady", "trigger", "wait-for-value", "wait-for-preconditions", "in-execution" and "wait-for-postconditions". The activation (execution) of a SN depends on the state of its concepts. Also, the state of a concept could change during the activity of the SN.

Relation Synergy provides a finite set of "procedural" relations subdivided in : 1) data relations (in/out) which connect concepts that are input/output arguments to a concept that represents a parametric activity (as a function, a procedure, ...), 2) control

relations (condition "cond" and succession "succ"), and 3) memory relations (specializationOf "sp", instanceOf "inst" and schemaOf "scm").
The above set of relations is special because each relation of the set has a procedural meaning and so, can have an effect on the activation of a SN.
Beside this set, a Synergy programmer can use any identifier as the name of a relation between two concepts, the relation will have, however, no "procedural" effect on Synergy interpreter.

2.3 Semantic network's activation

An execution of a Synergy "program" will correspond, in general, to a parallel activation of many SNs. SN' activation begins with the parallel activation of some concepts, then procedural relations spread, in parallel, the activity through the network. Procedural relations serve also to express conditions and constraints on concepts' activation and so, on the spreading of the activation.
Concept' activation corresponds to the determination and then the activation (execution) of its value. If the concept' type corresponds to a primitive operation, the operation will be activated, if the concept has a value which is a SN then it will be activated, if the concept has no value but its type is defined, the value is then created (by instanciation of the type' definition) and activated.
The graphical environment of Synergy helps the user to navigate in (and explore) such a dynamic space, composed of active SNs.
Due to space limitation, we can't go further in the description of Synergy, but the documentation and the language (reachable via the ftp address provided in the introduction) can be consulted for a complete presentation.

Graphical environment of Synergy The graphical environment of Synergy is basically a SN editor, it enables the management of : files of SNs, SN (creation, destruction, modification of concepts and relations, etc.), contexts (like putting in focus the LTM or other active contexts), execution and windows. A browser is provided to locate a concept into the LTM. Semantic network' activation is expressed as SN animation due to visual change of concept' state and of concept value. Each state has a graphical representation.

3 Application 1: Visual agent-oriented modeling of the Intensive Care Unit

Synergy has been used for a visual agent-oriented modeling of the Intensive Care Unit (ICU). ICU modelization is an important module in the SAFARI project. The goal of the ICU' module is to support tutorial sessions for nurses and physicians trainees. We focused on patient' assessment task performed by a nurse, and on the interaction between the two agents. During the modeling process, we became more and more aware of the potential role of Synergy as a tool for task cognitive analysis. Indeed, during the formulation and the simulation with Synergy, corrections, refinements and extensions have been added to the initial model.
Figure 3 shows a part of the LTM for the ICU application : the definitions of Agent, Nurse, Patient and some related types. The definition of the type Agent is part of the

built-in generalization graph provided by Synergy upon a creation of a new application (i.e. a new LTM). An Agent (Figure 3, window Agent:_Agt) has a messageQueue, a name, a dialog' window and an agent' manager . The types Nurse and Patient are specializations of the type Agent. Nurse (Figure 3, window Nurse:_n) is an agent with a spacial position and is responsible of patient' assessment which corresponds to five system evaluations (nervous, respiratory, circulatory, digestive and urinary). Two instances of Nurse have been specified : Magy and Susan.

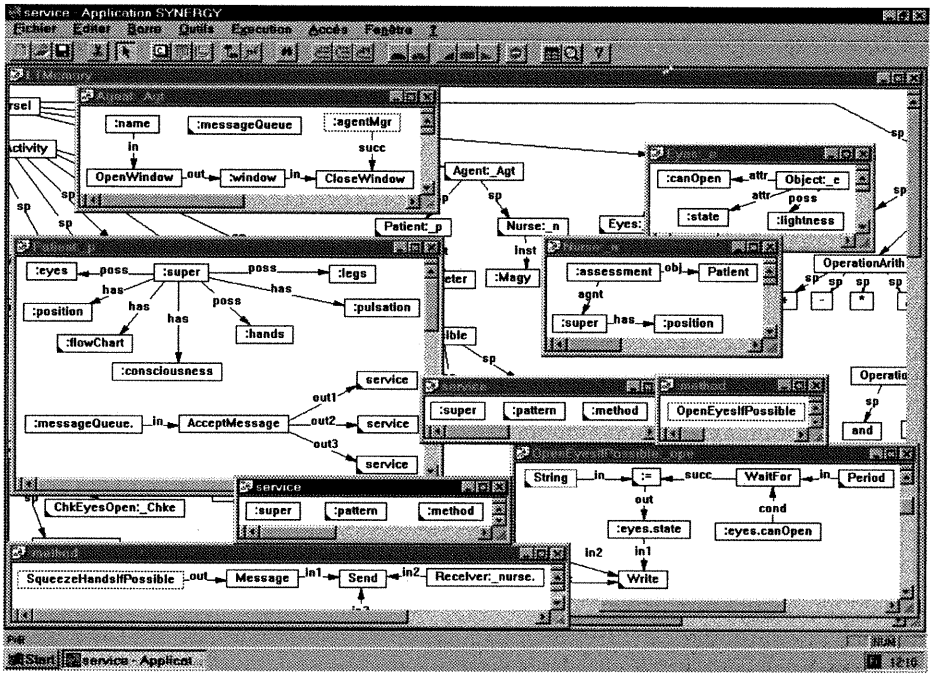


Figure 3 : Definitions of some types of the ICU' application

Patient (Figure 3, window Patient:_p) is an agent with attributes relevants to our modelization. Patient' behavior is to respond to three types of message' patterns : "open eyes", "move legs" and "squeeze hands". A method is associated to each pattern (Figure 3, window service). When the patient receives a message that can be matched with a pattern, the associated method will be activated. For instance, if the patient receives the message "open eyes", the method "OpenEyesIfPossible" will be activated. The definition of OpenEyesIfPossible (Figure 3) specifies that if the patient can open his eyes, a delay will occur that simulates the time for the patient to open his eyes, and then the state of his eyes will be set to "open".

Once the ICU' model has been defined, many scenarios (or situations) can be specified and simulated, depending on the states of the patient and the nurse.

Let us consider now the scenario sc_icu1 (Figure 4, window sc_icu1) where a nurse has to evaluate the patient' nervous system. In another scenario (not specified in this

paper), the nurse has to evaluate both the nervous and the circulatory systems. For this later, she has to use the Multi Channel Monitor to check several parameters.

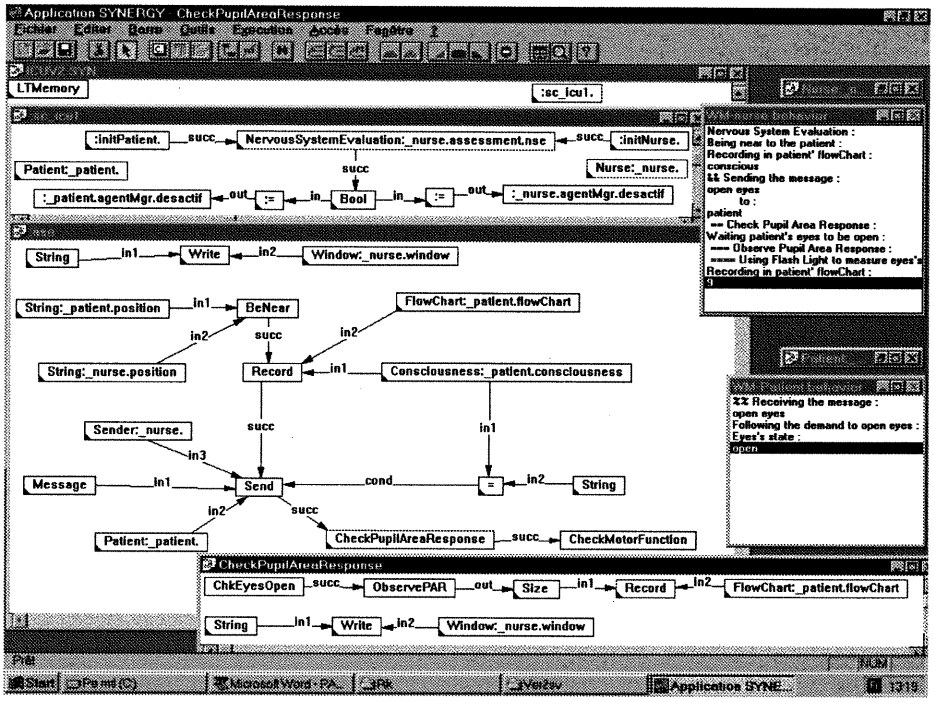


Figure 4 : Execution of the specific scenario sc_icu1 (part 1)

Scenario sc_icu1 (Figure 4, window sc_icu1) specifies the activation of a patient, a nurse and a request to the nurse to evaluate the patient' nervous system. The two concepts "initPatient" and "initNurse" (Figure 4, window sc_icu1) put the two agents in particular states (with specific values to some of their attributes). For instance, initPatient specifies that patient' eyes can be opened (canOpen = true) and their state is "close", patient' consciousness is "conscious", he can move legs, he can squeeze hands, etc.

The activation of the scenario sc_icu1 is done by putting the concept [Scenario :sc_icu1. #trigger] in the working memory (Figure 4, window ICUV2.SYN). Once the scenario sc_icu1 is activated, the patient Daniel (i.e. _patient = Daniel) and the nurse Magy (i.e. _nurse = Magy) will be created, initialized to a specific state and activated. For the two agents, the first action is to create a dialog' window (Figure 4, windows "WM-patient behavior" and "WM-nurse behavior") where comments are written as the agent' behavior proceed. Next, the nurse is asked to evaluate the patient' nervous system (NSE): the concept [NervousSystemEvaluation :_nurse.assessment.nse] in sc_icu1 is now "in-execution" state. Figure 4 shows the definition of this concept and the two dialog windows that give a "trace" of the concept' activation.

The simulation can be done step-by-step under the control of the user, so he can navigate in the active semantic networks and follow the execution in deeper detail.

According to the definition of the nervous system evaluation (Figure 4, window :nse), the nurse has to stand near the patient and record patient' consciousness in patient' flowChart. Hence, if the nurse' position is different from the patient' position, the nurse must move toward the patient (this is basically the definition of BeNear procedure, used in NSE). In our case, the nurse and the patient have the same position "center", and the patient is conscious.
Next, the nurse has to ask the patient to open his eyes in order to check the patient' pupil area response. Note that the nurse will ask the patient only if he is conscious, otherwise the nurse won't send a message to him and so, the next two checking tasks won't be executed.
The task "CheckPupilAreaResponse" is now in activation (Figure 4) : in order to observe patient' pupil area response, the nurse has to wait until the patient opens his eyes (this is done by the sub-task "ChkEyesOpen" in "CheckPupilAreaResponse"). Next, the nurse has to observe the patient' pupil area with a flash light and then record the result in the patient' flowChart. After the termination of CheckPupilAreaResponse, the nurse has to check patient' motor function. To do that, she first asks the patient to move his legs, waits for the response from the patient and then, records the result in patient' flowChart. The nurse will repeat the same behavior for the message "squeeze hands". With the termination of the task "CheckMotorFunction", the nervous system evaluation, as well as the activation of the whole scenario are terminated.

4 Application 2: Course generation from a curriculum

We have used Synergy to simulate the generation of a course from a curriculum, taking into account the student model. Our Synergy' formulation of this application is based on [Nkambou et al., 96]. Course generation process operates on the Curriculum Knowledge Transition Network (CKTN), a network of capabilities, instructional objectives and learning resources. If many objectives contribute to the acquisition of a capability, an heuristic is used to select a subset of objectives from the initial set. One such an heuristic is to consider "one strong objective", or "two moderate objectives " or "one moderate objective and one weak objective" or "three weak objectives".
The generation of a course takes into account the student model: the first step in the generation process is to mark each capability of the CKTN by "known", "partially known" or "unknown" (by the student). The definition of heuristics like the above one considers in fact the mark of the capability. For instance, the above heuristic is for an unknown capability, a different definition of the heuristic is given for a partially known capability. The generation process is activated by expressing the objective of the course : the set of the capabilities to be acquired by the student. With this initial set, a *selective* backward activation is done to determine for each unknown or partially known capability, the objectives that must be satisfied to acquire the capability. The backward activation is selective because the determination of the objectives to activate is done by an heuristic which selects a subset of objectives from the initial set. Next, for each activated objective, a selective backward activation is done to determine the capabilities that are mandatory for the objective. Each of these capabilities must be acquired too.
Figure 5 shows our reformulation of this application in Synergy : Capabilities and Objectives are "active" concepts ; when a capability is activated, the function ComputeObjectiveContr (Figure 5, window Capability) will identify all the objectives

that contribute to the acquisition of the capability. The result will be taken by the function HeuristicObjective which returns a list of selected objectives. The primitive operation "Map" will then apply the operation "TriggerConc" on each element of the list ("TriggerConc" is the value of the second argument of Map). TriggerConc puts its argument (which is a concept' referent) in a trigger state.
The type Objective is defined in the same way but in addition, we specify that the referent of the objective (once it is active, and so required !) will be added to the course structure.

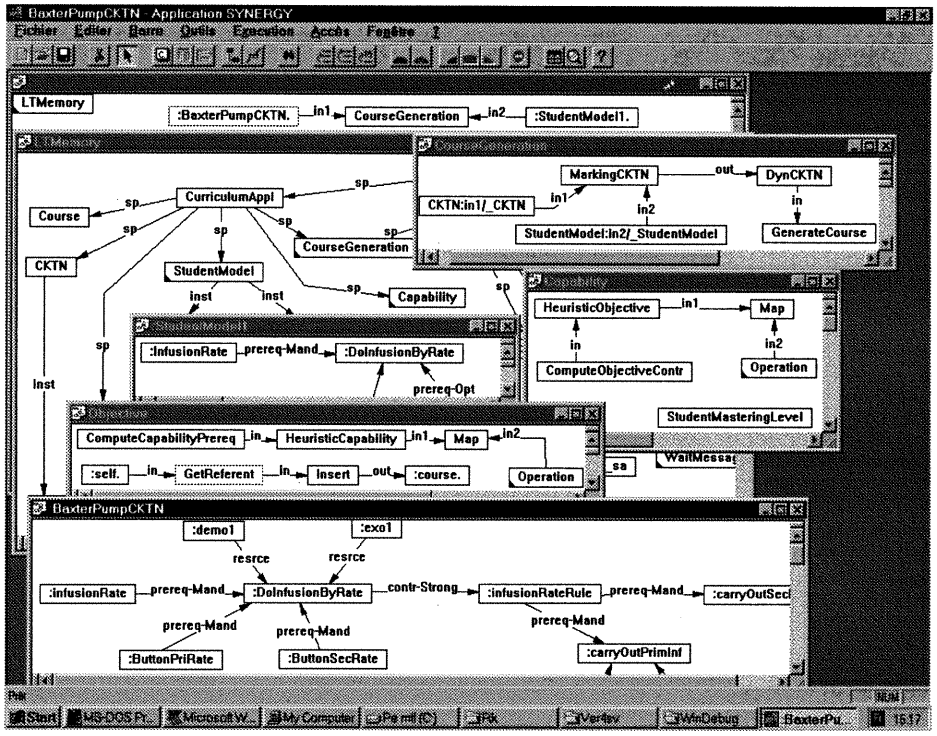


Figure 5 : Generation of a course from a curriculum

5 Related works

Graph-oriented formulation of procedural knowledge is used in programming languages (for instance, Petri nets and data flow graphs), in software engineering and modelization methods where data flow diagrams and transition diagrams are used and in ITS and learning environments ([Psotka et al., 88], [Larkin and Chabay, 92]) where some of the above graphs are used as well as other graphs : task graphs, action graphs, genetic graphs, structural graphs, etc.
Graph-based formalism was used also for declarative knowledge, as the case for semantic networks families [Lehmann, 92].
We attempt to integrate in one graph-based language many graph-oriented models.

6 Future works

Some extensions to our visual agent-oriented modeling of ICU are in order : to consider for instance the physician role, the simulation of some devices used in the ICU and the addition, to the simulation/demonstration mode, of a critic system.
We plan to use Synergy in the student modeling module and for a multi-agents modeling of SAFARI as whole. Also, a visual-multi-media general-purpose simulation environment for Synergy is in progress [Kabbaj and Frasson, 95].

7 Conclusion

We introduced a graphical multi-paradigm language, based on active semantic networks. The language, called Synergy is useful for many complex systems, as ITS. Indeed and as this paper attempts to illustrate, an integration-based language can be used in many modules of an ITS, where different types of knowledge are treated by different processes.

In the User' manual of Synergy we illustrate its use with many examples taken from procedural, functional, object-oriented and agent oriented programmings. Other examples are taken from specific domains as causal networks, project management, real-time temporal knowledge based systems, etc.

Acknowledgment
This research is part of SAFARI project which is supported by the MICST (Ministère de l'Industrie, du Commerce, des Sciences et de la Technologie).

Bibliography

Gecsei J., and Frasson C., SAFARI: an Environment for Creating Tutoring Systems in Industrial Training, EdMedia, World Conference on Educational Multimedia and Hypermedia, Vancouver, June 1994.

Kabbaj A. et C. Frasson, CAL : When education influences the design of an AI language, in Proc. of AI-ED'95, 7th World Conference on AI in Education, pp. 343-350, Washington, Aug. 1995.

Larkin J. H. et R. W. Chabay (eds.), Computer-Assisted instruction and intelligent tutoring systems: Shared goals and complementary approaches, Lawrence Erlbaum Associates, 1992.

Lehmann F. (ed), Special Issue on Semantic Networks in AI, in Computers and Mathematics with Applications, 23:2-9, 1992.

Nkambou R., M. C. Frasson and C. Frasson, Generating Courses in an Intelligent Tutoring System, aie-eai'96, 9th Intern. Conf. on Industrial & Engineering applications of AI & Expert Systems, Japon, June 1996.

Psotka J., L. Dan Massey et S. A. Mutter (Eds.), Intelligent Tutoring Systems, Lessons Learned, Lawrence Erlbaum Ass. Pub., 1988.

Sowa J. F., Conceptual Structures : Information Processing in Mind and Machine, Addison-Wesley, 1984.

Annexe E

Utilisation de la mémoire dans la "reconstruction" d'un concept

Cette exemple correspond à une "note" (datée du 31 août 1995) adressée à Peter Clark à propos d'une utilisation éventuelle de notre modèle de la mémoire dans le cadre de son approche building concept from components. Dans une note de travail, Clark et Porter montrent comment, à partir d'un ensemble de descriptions (les composantes), un concept peut être "construit" par unification successive des composantes. La construction est effectuée en six étapes.
Le sujet de notre "note" est de montrer comment cette approche peut être envisagée comme une "exploitation" de la mémoire construite selon notre modèle.
Bien qu'elle ne constitue qu'une note à une "note de travail", l'exemple présente toutefois une utilisation intéressante de la mémoire (raison principale de son inclusion dans cette thèse).
Nous gardons la formulation en anglais ainsi que le "style" de la note.

Let's assume that we have the memory as described in figure 1 and that all we know about BUS is that it is a roadVehicle. Note first that this memory could be the result of the integration process which receive a stream of conceptual structures (not simply CG) : definitions, schemas, etc.
So, we know that Bus is a roadVehicle. Next, the integration process will receive as input 1 :

[PetrolEngine] <-engine- [Bus] -mass-> [Average] **with** Bus as a focus.

It will be integrated as shown in the figure. Now, if we are asked to *elaborate* on the "bus with petrolEngine", we will explore the memory, taking into account the two "principles" :
- first, a concept can be expanded by its definition. This an elaboration is "sure". Moreover, to have a more precise description, the expansion should not replace the concept by its genus.
- schema behind a concept C **can be** joined to a subConcept of C, if there is no conflict with the current description of the concept.

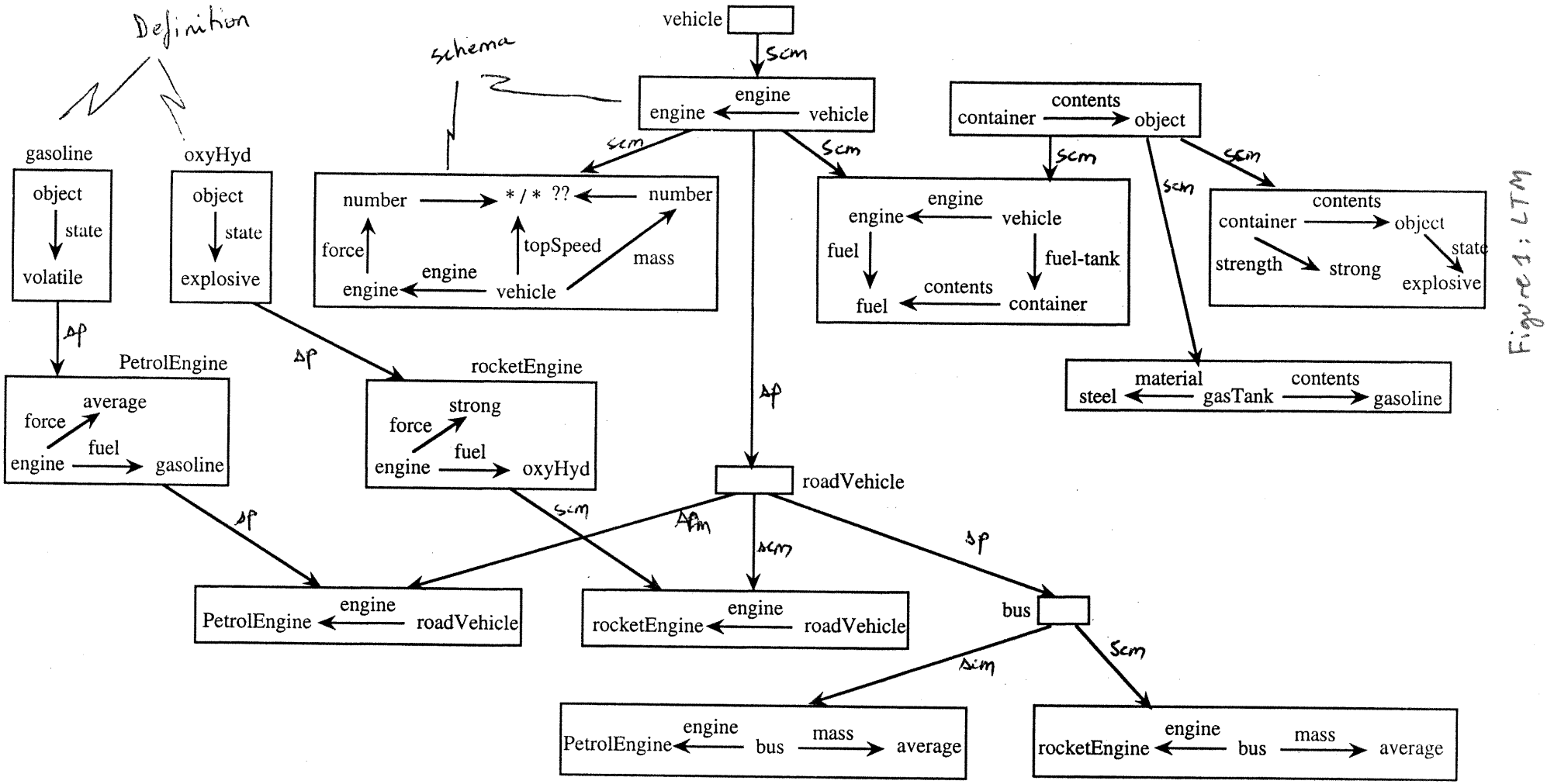
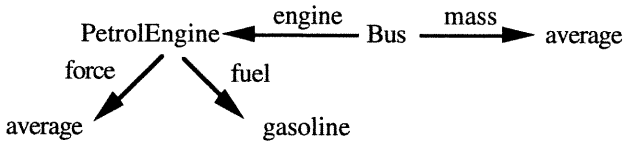


Figure 1

Figure 1 : LTM

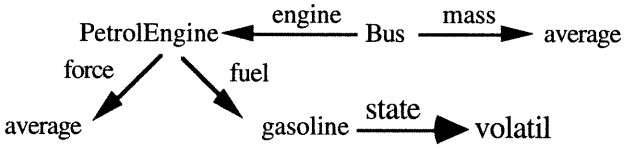
"brother" schemas (i.e. schemas behind the same concept) can be joined, if there is no conflict. In fact, schemas can represent complementary situations (as well as distinct and incompatible situations were the same concept(s) is (are) used).

If we consider the schemas of roadVehicle, only PetrolEngine <-engine- roadVehicle could unify. This means that RocketEngine' definition and OxyHyd' definition are not available in this case. Next, we can elaborate Bus' description (which is not changed yet) by following the two paths of the schema' node PetrolEngine <-engine- roadVehicle :
- expand the concept PetrolEngine in Bus' description :

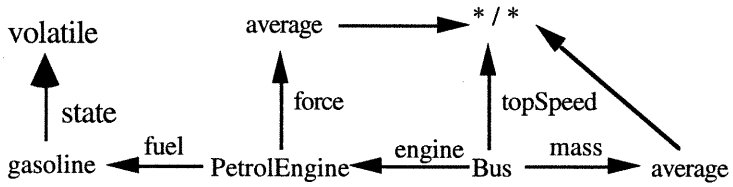


Note : In the expansion, i have not replaced PetrolEngine by Engine, to have a more precise description.

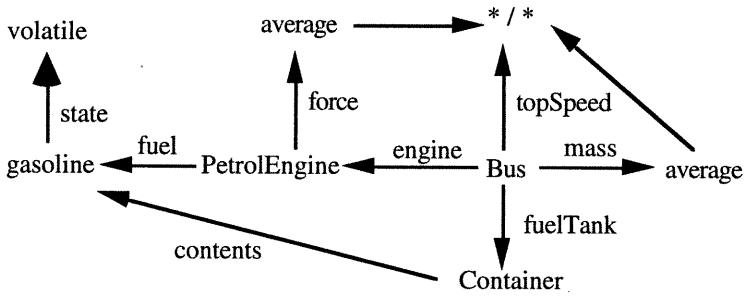
- expand the concept gasoline :



- roadVehicle is a vehicle. We consider the schemas related to it. So, we can join to Bus' description the following schema :

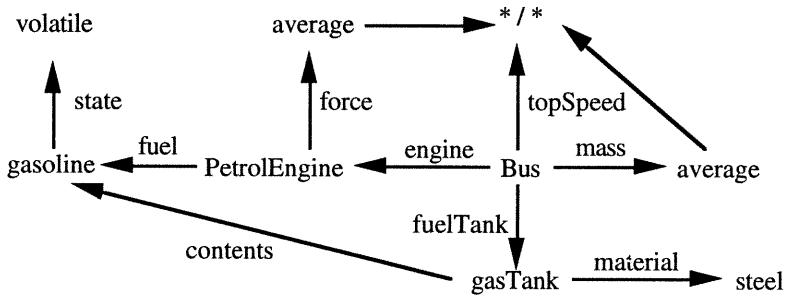


- We can also join the other schema, since there is still no contradiction:



- Now, we can consider the inheritance of the two schemas : the first schema has no inheritance but the second is the specialization of [container]-contents->[object] and is "brother" of two schemas. So we try to join them, one at a time, to the current Bus' description. We begin by the schema :

[gasTank]-contents-> ... :



- Now, we try to join the other schema to Bus' description :

[container]-strength->[strong] ...

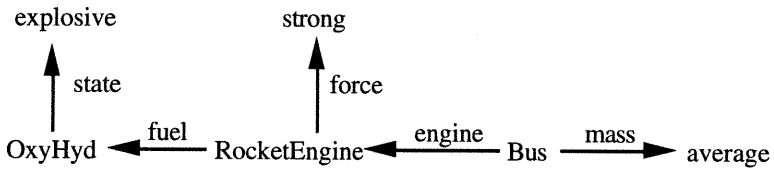
but the join will fail since "state is volatile" (and this is surely true; by definition-expansion) and "state is explosive" are incompatible.

=======> Hence, we have an *elaborate description* of "PetrolEngine Bus".

Suppose now that the integration process will receive as input 2 :

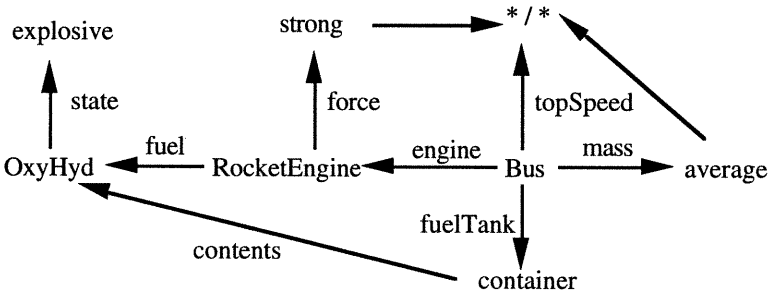
[RocketEngine] <-engine- [Bus] -mass-> [Average] **with** Bus as a focus.

The same elaboration process (= "building process" according to your terminology) is done :
- only schema : RocketEngine <-engine- roadVehicle can be joined to the description (there is no change in this case).
- Next, we can join RocketEngine and OxyHyd definitions. The result is :



Note : Of course, the information about PetrolEngine and gasoline is no more considered since the schema RocketEngine <-engine- roadVehicle , which is the "key" to access to them, is incompatible with the input.

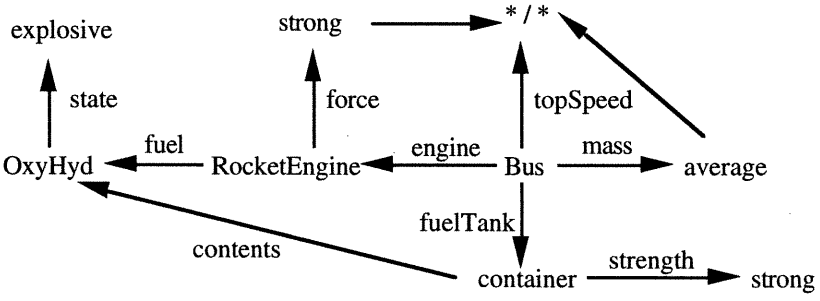
- Next, we consider the other path, which pass trough roadVehicle' node, and again we can join the two schemas about vehicle. The result will be :



- Again, we can consider the inheritance of the schema: [vehicle]-fuelTank->[container] .. , and we consider its two "brothers" :

- The schema [gasTank]-contents->[gasoline] .... can't be joined because gasoline and OxyHyd are incompatible.

- The schema [container]-contents->[object] ... can be joined to the current Bus' description, and the result is :



And so, we have an *elaborate description* of "RocketEngine Bus".

Conclusion

Here, we use the product of the integration process which is an heterogenous generalization graph, the graph represents the system' memory. The special use here is "concept' elaboration" or "building concept" which is a successive join of information taken in the inheritance of the initial description (input 1 and input 2 in our case).

One problem of course is how to select information if many are possibles : the problem is avoided here because the content of memory is small.

The idea of concept' elaboration is however very interesting since the content of memory change over time, and what we know about a concept (Bus for instance) at one time is different from what we know and another time.