Bases d'objets : une infrastructure de représentation de connaissances pour la gestion de données en CAO

Ignacio de Jesus Ania Briseno

THESE

PRESENTEE A

L'Institut National Polytechnique de Grenoble

L'Université Joseph FOURIER

pour obtenir

le titre de Docteur Ingénieur

(Spécialité : Informatique)

par

Ignacio de Jesús ANIA BRISEÑO

* * *

BASES D'OBJETS

UNE INFRASTRUCTURE DE REPRESENTATION DE CONNAISSANCES POUR LA GESTION DE DONNEES EN CAO

* * *

Soutenue le 27 juin 1988 devant la Commission d'Examen.

JURY

Monsieur	**G. MAZARE**	**, PRESIDENT**
Messieurs	**C. JULLIEN**	)
	J. MERMET	) **EXAMINATEURS**
	F. RECHENMANN	)

UNIVERSITE Joseph FOURIER (GRENOBLE I)

Président de l'Université :
M. PAYAN Jean Jacques

Année Universitaire 1987 - 1988

MEMBRES DU CORPS ENSEIGNANT DE SCIENCES ET DE GEOGRAPHIE

PROFESSEURS DE 1ère Classe

ARNAUD Paul	Chimie Organique
ARVIEU ROBERT	Physique Nucléaire I.S.N.
AUBERT Guy	Physique C.N.R.S
AURIAULT Jean-Louis	Mécanique
AYANT Yves	Physique Approfondie
BARBIER Marie-Jeanne	Electrochimie
BARJON Robert	Physique Nucléaire ISN
BARNOUD Fernand	Biochimie Macromoléculaire Végétale
BARRA Jean-René	Statistiques-Mathématiques Appliquées
BECKER Pierre	Physique
BEGUIN Claude	Chimie Organique
BELORISKY Elie	Physique
BENZAKEN Claude	Mathématiques Pures
BERARD Pierre	Mathématiques Pures
BERNARD Alain	Mathématiques Pures
BERTRANDIAS Françoise	Mathématiques Pures
BERTRANDIAS Jean-Paul	Mathématiques Pures
BILLET Jean	Géographie
BOELHER Jean-Paul	Mécanique
BONNIER Jane Marie	Chimie Générale
BOUCHEZ Robert	Physique Nucléaire ISN
BRAVARD Yves	Géographie
CARLIER Georges	Biologie Végétale
CAUQUIS Georges	Chimie Organique
CHARDON Michel	Géographie
CHIBON Pierre	Biologie Animale
COHEN ADDAD Jean-Pierre	Physique
COLIN DE VERDIERE Yves	Mathématiques Pures
CYROT Michel	Physique du Solide
DEBELMAS Jacques	Géologie Générale
DEGRANGE Charles	Zoologie
DEMAILLY Jean-Pierre	Mathématiques Pures
DENEUVILLE Alain	Physique
DEPORTES Charles	Chimie Minérale
DOLIQUE Jean-Michel	Physique des Plasmas
DOUCE Roland	Physiologie Végétale
DUCROS Pierre	Cristallographie
FONTAINE Jean-Marc	Mathématiques Pures
GAGNAIRE Didier	Chimie Physique
GERMAIN Jean-Pierre	Mécanique,
GIRAUD Pierre	Géologie
HICTER Pierre	Chimie
IDELMAN Simon	Physiologie Animale
JANIN Bernard	Géographie
JOLY Jean-René	Mathématiques Pures
KAHANE André, détaché	Physique
KAHANE Josette	Physique
KRAKOWIAK Sacha	Mathématiques Appliquées

LAJZEROWICZ Jeanine	Physique
LAJZEROWICZ Joseph	Physique
LAURENT Pierre-Jean	Mathématiques Appliquées
LEBRETON Alain	Mathématiques Appliquées
DE LEIRIS Joël	Biologie
LHOMME Jean	Chimie
LLIBOUTRY Louis	Géophysique
LOISEAUX Jean-Marie	Sciences Nucléaires I.S.N.
LUNA Domingo	Mathématiques Pures
MACHE Régis	Physiologie Végétale
MASCLE Georges	Géologie
MAYNARD Roger	Physique du Solide
OMONT Alain	Astrophysique
OZENDA Paul	Botanique (Biologie Végétale)
PAYAN Jean-Jacques	Mathématiques Pures
PEBAY-PEYROULA Jean-Claude	Physique
PERRIER Guy	Géophysique
PIERRARD Jean-Marie	Mécanique
PIERRE Jean-Louis	Chimie Organique
RENARD Michel	Thermodynamique
RINAUDO Marguerite	Chimie CERMAV
ROSSI André	Biologie
SAXOD Raymond	Biologie Animale
SENGEL Philippe	Biologie Animale
SERGERAERT Francis	Mathématiques Pures
SOUCHIER Bernard	Biologie
SOUTIF Michel	Physique
STUTZ Pierre	Mécanique
TRILLING Laurent	Mathématiques Appliquées
VALENTIN Jacques	Physique Nucléaire I.S.N.
VAN CUTSEM Bernard	Mathématiques Appliquées
VIALON Pierre	Géologie

PROFESSEURS de 2ème Classe

ADIBA Michel	Mathématiques Pures
ANTOINE Pierre	Géologie
ARMAND Gilbert	Géographie
BARET Paul	Chimie
BLANCHI J.Pierre	STAPS
BLUM Jacques	Mathématiques Appliquées
BOITET Christian	Mathématiques Appliquées
BORNAREL Jean	Physique
BRUANDET J.François	Physique
BRUGAL Gérard	Biologie
BRUN Gilbert	Biologie
CASTAING Bernard	Physique
CERFF Rudiger	Biologie
CHIARAMELLA Yves	Mathématiques Appliquées
COURT Jean	Chimie
DUFRESNOY Alain	Mathématiques Pures
GASPARD François	Physique
GAUTRON René	Chimie
GENIES Eugène	Chimie
GIDON Maurice	Géologie
GIGNOUX Claude	Sciences Nucléaires
GILLARD Roland	Mathématiques Pures
GIORNI Alain	Sciences Nucléaires
GONZALEZ SPRINBERG Gérardo	Mathématiques Pures
GUIGO Maryse	Géographie
GUMUCHAIN Hervé	Géographie
GUITTON Jacques	Chimie

HACQUES Gérard	Mathématiques Appliquées
HERBIN Jacky	Géographie
HERAULT Jeanny	Physique
JARDON Pierre	Chimie
JOSELEAU Jean-Paul	Biochimie
KERCKHOVE Claude	Géologie
LONGEQUEUE Nicole	Sciences Nucléaires I.S.N.
LUCAS Robert	Physique
MANDARON Paul	Biologie
MARTINEZ Francis	Mathématiques Appliquées
NEMOZ Alain	Thermodymamique CNRS - CRTBT
OUDET Bruno	Mathématiques Appliquées
PECHER Arnaud	Géologie
PELMONT Jean	Biochimie
PERRIN Claude	Sciences Nucléaires I.S.N.
PFISTER Jean-Claude	Physique du Solide
PIBOULE Michel	Géologie
RAYNAUD Hervé	Mathématiques Appliquées
RICHARD Jean-Marc	Physique
RIEDTMANN Christine	Mathématiques Pures
ROBERT Gilles	Mathématiques Pures
ROBERT Jean-Bernard	Chimie Physique
SARROT-REYNAULD Jean	Géologie
SAYETAT Françoise	Physique
SERVE Denis	Chimie
STOECKEL Frédéric	Physique
SCHOLL Pierre-Claude	Mathématiques Appliquées
SUBRA Robert	Chimie
VALLADE Marcel	Physique
VIDAL Michel	Chimie Organique
VIVIAN Robert	Géographie
VOTTERO Philippe	Chimie

MEMBRES DU CORPS ENSEIGNANT DE L' IUT 1

PROFESSEURS de 1ère Classe

BUISSON Roger	Physique IUT 1
DODU Jacques	Mécanique Appliquée IUT 1
NEGRE Robert	Génie Civil IUT 1
NOUGARET Marcel	Automatique IUT 1
PERARD Jacques	EEA. IUT 1

PROFESSEURS de 2ème classe

BOUTHINON Michel	EEA. IUT 1
CHAMBON René	Génie Mécanique IUT 1
CHEHIKIAN Alain	EEA. IUT 1
CHENAVAS Jean	Physique IUT 1
CHOUTEAU Gérard	Physique IUT 1
CONTE René	Physique IUT 1
GOSSE Jean-Pierre	EEA.IUT 1
GROS Yves	Physique IUT 1
KUHN Gérard, (Détaché)	Physique IUT 1
MAZUER Jean	Physique IUT 1
MICHOULIER Jean	Physique IUT 1
MONLLOR Christian	EEA.IUT 1
PEFFEN René	Métallurgie IUT 1
PERRAUD Robert	Chimie IUT 1
PIERRE Gérard	Chimie IUT 1
TERRIEZ Jean-Michel	Génie Mécanique IUT 1
TOUZAIN Philippe	Chimie IUT 1
VINCENDON Marc	Chimie IUT 1

PROFESSEURS DE PHARMACIE

AGNIUS-DELORD Claudine	Physique	Faculté La Tronche
ALARY Josette	Chimie Analytique	Faculté La Tronche
BERIEL Hélène	Physiologie et Pharmacologie	Faculté La Tronche
CUSSAC Max	Chimie Therapeutique	Faculté La Tronche
DEMENGE Pierre	Pharmacodynamie	Faculté La Tronche
FAVIER Alain	Biochimie	C.H.R.G.
JEANNIN Charles	Pharmacie Galénique	Faculté Meylan
LATURAZE Jean	Biochimie	Faculté La Tronche
LUU DUC Cuong	Chimie Générale	Faculté La Tronche
MARIOTTE Anne-Marie	Pharmacognosie	Faculté La Tronche
MARZIN Daniel	Toxicologie	Faculté Meylan
RENAUDET Jacqueline	Bactériologie	Faculté La Tronche
ROCHAT Jacques	Hygiéne et Hydrologie	Faculté La Tronche
SEIGLE-MURANDI Françoise	Botanique et Cryptogamie	Faculté Meylan
VERAIN Alice	Pharmacie Galénique	Faculté Meylan

MEMBRES DU CORPS ENSEIGNANT DE MEDECINE

PROFESSEURS CLASSE EXEPTIONNELLE ET 1ère CLASSE

AMBLARD Pierre	Dermatologie	C.H.R.G.
AMBROISE-THOMAS Pierre	Parasitologie	C.H.R.G.
BEAUDOING André	Pédiatrie-Puericulture	C.H.R.G.
BEZEZ Henri	Orthopédie-Traumatologie	Hopital SUD
BONNET Jean-Louis	Ophtalmologie	C.H.R.G.
BOUCHET Yves	Anatomie	Faculté La Merci
	Chirurgie Générale et Digestive	C.H.R.G.
BUTEL Jean	Orthopédie-Traumatologie	C.H.R.G.
CHAMBAZ Edmond	Biochimie	C.H.R.G.
CHAMPETIER Jean	Anatomie-Topographique et Appliquée	C.H.R.G.
CHARACHON Robert	O.R.L.	C.H.R.G.
COLOMB Maurice	Immunologie	Hopital sud
COUDERC Pierre	Anatomie-Pathologique	C.H.R.G.
DELORMAS Pierre	Pneumophtisiologie	C.H.R.G.
DENIS Bernard	Cardiologie	C.H.R.G.
GAVEND Michel	Pharmacologie	Faculté La Merci
HOLLARD Daniel	Hématologie	C.H.R.G.
LATREILLE René	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
LE NOC Pierre	Bactériologie-Virologie	C.H.R.G.
MALINAS Yves	Gynécologie et Obstétrique	C.H.R.G.
MALLION Jean-Michel	Médecine du Travail	C.H.R.G.
MICOUD Max	Clinique Médicale et Maladies Infectieuses	C.H.R.G.
MOURIQUAND Claude	Histologie	Faculté La Merci
PARAMELLE Bernard	Pneumologie	C.H.R.G.
PERRET Jean	Neurologie	C.H.R.G.
RACHAIL Michel	Hépato-Gastro-Entérologie	C.H.R.G.
DE ROUGEMONT Jacques	Neurochirurgie	C.H.R.G.
SARRAZIN Roger	Clinique Chirurgicale	C.H.R.G.
STIEGLITZ Paul	Anestésiologie	C.H.R.G.
TANCHE Maurice	Physiologie	Faculté La Merci
VIGNAIS Pierre	Biochimie	Faculté La Merci

PROFESSEURS 2ème CLASSE

BACHELOT Yvan	Endocrinologie	C.H.R.G.
BARGE Michel	Neurochirurgie	C.H.R.G.
BENABID Alim Louis	Biophysique	Faculté La Merci
BENSA Jean-Claude	Immunologie	Hopital Sud
BERNARD Pierre	Gynécologie-Obstétrique	C.H.R.G.
BESSARD Germain	Pharmacologie	ABIDJAN
BOLLA Michel	Radiothérapie	C.H.R.G.
BOST Michel	Pédiatrie	C.H.R.G.
BOUCHARLAT Jacques	Psychiatrie Adultes	Hopital Sud
BRAMBILLA Christian	Pneumologie	C.H.R.G.
CHIROSSEL Jean-Paul	Anatomie-Neurochirurgie	C.H.R.G.
COMET Michel	Biophysique	Faculté La Merci
CONTAMIN Charles	Chirurgie Thoracique et Cardiovasculaire	C.H.R.G.
CORDONNIER Daniel	Néphrologie	C.H.R.G.
COULOMB Max	Radiologie	C.H.R.G.
CROUZET Guy	Radiologie	C.H.R.G.
DEBRU Jean-Luc	Médecine Interne et Toxicologie	C.H.R.G.
DEMONGEOT Jacques	Biostatistiques et Informatique Médicale	Faculté La Merci
DUPRE Alain	Chirurgie Générale	C.H.R.G.
DYON Jean-François	Chirurgie Infantile	C.H.R.G.
ETERRADOSSI Jacqueline	Physiologie	Faculté La Merci
FAURE Claude	Anatomie et Organogénèse	C.H.R.G.
FAURE Gilbert	Urologie	C.H.R.G.
FOURNET Jacques	Hépato-Gastro-Entérologie	C.H.R.G.
FRANCO Alain	Médecine Interne	C.H.R.G.
GIRARDET Pierre	Anesthésiologie	C.H.R.G.
GUIDICELLI Henri	Chirurgie Générale et Vasculaire	C.H.R.G.
GUIGNIER Michel	Thérapeutique et Réanimation Médicale	C.H.R.G.
HADJIAN Arthur	Biochimie	Faculté La Merci
HALIMI Serge	Endocrinologie et Maladies Métaboliques	C.H.R.G.
HOSTEIN Jean	Hépato-Gastro-Entérologie	C.H.R.G.
HUGONOT Robert	Médecine Interne	C.H.R.G.
JALBERT Pierre	Histologie-Cytogénétique	C.H.R.G.
JUNIEN-LAVILLAUROY Claude	O.R.L.	C.H.R.G.
KOLODIE Lucien	Hématologie Biologique	C.H.R.G.
LETOUBLON Christian	Chirurgie Générale	C.H.R.G.
MACHECOURT Jacques	Cardiologie et Maladies Vasculaires	C.H.R.G.
MAGNIN Robert	Hygiène	C.H.R.G.
MASSOT Christian	Médecine Interne	C.H.R.G.
MOUILLON Michel	Ophtalmologie	C.H.R.G.
PELLAT Jacques	Neurologie	C.H.R.G.
PHELIP Xavier	Rhumatologie	C.H.R.G.
RACINET Claude	Gynécologie-Obstétrique	Hopital Sud
RAMBAUD Pierre	Pédiatrie	C.H.R.G.
RAPHAEL Bernard	Stomatologie	C.H.R.G.
SCHAERER René	Cancérologie	C.H.R.G.
SEIGNEURIN Jean-Marie	Bactériologie-Virologie	Faculté La Merci
SELE Bernard	Cytogénétique	Faculté La Merci
SOTTO Jean-Jacques	Hématologie	C.H.R.G.
STOEBNER Pierre	Anatomie Pathologique	C.H.R.G.
VROUSOS Constantin	Radiothérapie	C.H.R.G.

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Président : Georges LESPINARD **Année 1988**

Professeurs des Universités

BARIBAUD Michel	ENSERG
BARRAUD Alain	ENSIEG
BAUDELET Bernard	ENSPG
BEAUFILS Jean-Pierre	ENSEEG
BLIMAN Samuel	ENSERG
BLOCH Daniel	ENSPG
BOIS Philippe	ENSHMG
BONNETAIN Lucien	ENSEEG
BOUVARD Maurice	ENSHMG
BRISSONNEAU Pierre	ENSIEG
BRUNET Yves	IUFA
CAILLERIE Denis	ENSHMG
CAVAIGNAC Jean-François	ENSPG
CHARTIER Germain	ENSPG
CHENEVIER Pierre	ENSERG
CHERADAME Hervé	UFR PGP
CHOVET Alain	ENSERG
COHEN Joseph	ENSERG
COUMES André	ENSERG
DARVE Félix	ENSHMG
DELLA-DORA Jean	ENSIMAG
DEPORTES Jacques	ENSPG
DOLMAZON Jean-Marc	ENSERG
DURAND Francis	ENSEEG
DURAND Jean-Louis	ENSIEG
FOGGIA Albert	ENSIEG
FONLUPTJean	ENSIMAG
FOULARD Claude	ENSIEG
GANDINI Alessandro	UFR PGP
GAUBERT Claude	ENSPG
GENTIL Pierre	ENSERG
GREVEN Hélène	IUFA
GUERIN Bernard	ENSERG
GUYOT Pierre	ENSEEG
IVANES Marcel	ENSIEG
JAUSSAUD Pierre	ENSIEG
JOUBERT Jean-Claude	ENSPG
JOURDAIN Geneviève	ENSIEG
LACOUME Jean-Louis	ENSIEG
LESIEUR Marcel	ENSHMG
LESPINARD Georges	ENSHMG
LONGEQUEUE Jean-Pierre	ENSPG
LOUCHET François	ENSIEG
MASSE Philippe	ENSIEG
MASSELOT Christian	ENSIEG
MAZARE Guy	ENSIMAG
MOREAU René	ENSHMG
MORET Roger	ENSIEG
MOSSIERE Jacques	ENSIMAG
OBLED Charles	ENSHMG
OZIL Patrick	ENSEEG
PARIAUD Jean-Charles	ENSEEG
PERRET René	ENSIEG
PERRET Robert	ENSIEG
PIAU Jean-Michel	ENSHMG
POUPOT Christian	ENSERG
RAMEAU Jean-Jacques	ENSEEG
RENAUD Maurice	UFR PGP
ROBERT André	UFR PGP
ROBERT François	ENSIMAG
SABONNADIERE Jean-Claude	ENSIEG
SAUCIER Gabrielle	ENSIMAG
SCHLENKER Claire	ENSPG
SCHLENKER Michel	ENSPG
SILVY Jacques	UFR PGP
SIRIEYS Pierre	ENSHMG
SOHM Jean-Claude	ENSEEG
SOLER Jean-Louis	ENSIMAG
SOUQUET Jean-Louis	ENSEEG
TROMPETTE Philippe	ENSHMG
VEILLON Gérard	ENSIMAG
ZADWORNY François	ENSERG

Professeur Université des Sciences Sociales (Grenoble II)

BOLLIET Louis

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LADET Pierre
LATOMBE Claudine
LE GORREC Bernard
MADAR Roland
MULLER Jean
NGUYEN TRONG Bernadette
PASTUREL Alain
PLA Fernand
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri

Chercheurs du C.N.R.S
Directeurs de recherche 1ère Classe

CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
LANDAU Ioan
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ANSARA Ibrahim
ARMAND Michel
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
COURTOIS Bernard
DAVID René
DRIOLE Jean
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GUELIN Pierre
JOUD Jean-Charles
KLEITZ Michel
KOFMAN Walter
KAMARINOS Georges
LEJEUNE Gérard
LE PROVOST Christian
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MUNIER Jacques
PIAU Monique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
WACK Bernard

Personnalités agrées à titre permanent à diriger des travaux de recherche (décision du conseil scientifique)

E.N.S.E.E.G

CHATILLON Christian
HAMMOU Abdelkader
MARTIN GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G

BOREL Joseph

E.N.S.I.E.G

DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.G

ROWE Alain

E.N.S.I.M.A.G

COURTIN Jacques

E.F.P.

CHARUEL Robert

C.E.N.G

CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIB Maurice
VINCENDON Marc

Laboratoires extérieurs

C.N.E.T

DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

Je tiens à remercier

Monsieur Guy MAZARE, Professeur à l'Ecole National Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble (ENSIMAG), qui a bien voulu me faire l'honneur de présider le jury de cette thèse.

Monsieur Jean MERMET, Directeur de Recherche au Centre National de la Recherche Scientifique (CNRS), pour m'avoir accueilli au sein de son équipe de recherche et avoir pris la responsabilité de cette thèse. Qu'il trouve ici l'expression de ma profonde gratitude pour m'avoir fait profiter de sa grande expérience et compétence scientifique et technique.

Monsieur Christian JULLIEN, Ingénieur au Département de Recherche en Conception Assistée, du Centre National d'Etudes des Télécommunications (CNET), et Monsieur François RECHENMANN, Directeur de Recherche à l'Institut National de Recherche en Informatique et en Automatique (INRIA), Responsable du projet SHIRKA, pour avoir bien voulu juger mon travail de recherche et participer à ce jury. Je leur suis très reconnaissant des améliorations que j'ai pu apporter à la rédaction définitive de ce document grâce à leurs précieux commentaires et conseils.

Tous les membres de l'équipe CASCADE et du Laboratoire ARTEMIS en général, ainsi que du groupe Bases de Données du projet AIDA-ESPRIT, pour la chaleur de leur amitié aussi bien à l'intérieur qu'à l'extérieur des environnements de travail.

Le personnel de la Médiathèque de l'Institut IMAG, pour leur aide constante tout au long de mes études en France, qui a représentée sans nul doute un facteur décisif dans l'aboutissement de ce travail. Qu'elles trouvent ici l'expression de ma profonde sympathie.

Les membres du Service de Reprographie de l'Institut IMAG pour le soin et l'efficacité qu'ils ont apporté au tirage de cette thèse.

Il me faut également remercier le Conseil National de Science et Technologie du Mexique (CONACYT), pour son soutien financier sans lequel ce travail n'aurait pu être accompli.

A Diana, a Valeria y a Daniela.

TABLE DES MATIERES

Introduction 1

CHAPITRE 1 : Gestion de données en CAO 17

1.1. Introduction 19
1.2. Définition et manipulation d'objets composites 20
1.3. Coexistence des multiples descriptions d'un objet 22
1.4. Organisation multi-hiérarchisée des objets 24
1.5. Modification du schéma conceptuel 26
1.6. Maintien de l'identité des objets 27
1.7. Représentation d'informations sémantiques 28
1.8. Représentation des propriétés opératoires des objets 30
1.9. Traitement flexible d'erreurs et de cas exceptionnels 30
1.10. Discussion 31

CHAPITRE 2 : Concepts fondamentaux 35

2.1. Introduction 37
2.2. Approche centré-objet 37
2.3. Identification des objets 40
2.4. Mécanismes d'abstraction 40
2.5. Discussion 47

CHAPITRE 3 : Paradigme de modélisation 55

3.1. Introduction 57
3.2. Les propriétés prédéfinies 57
3.3. La création et la destruction de classes 60
3.4. La création et la destruction d'instances 63
3.5. Les caractéristiques filtrées 65
3.6. Les descripteurs 67
3.7. Les caractéristiques préétablies et calculées 69
3.8. Les attributs des instances 73
3.9. Les objets subordonnés 77
3.10. Les assertions 79
3.11. Les objets opérateur 82
3.12. Discussion 84

CHAPITRE 4 : Objets génériques et objets représentatifs 87

4.1. Introduction 89
4.2. L'approche ontologique 89
4.3. Les associations représentatif-de 91
4.4. Une nouvelle forme de généralisation 93
4.5. Exemple de modélisation 98
4.6. Discussion 109

Conclusions 113

Références 121

ANNEXES 133

A. Contraintes d'intégrité sur les associations entre objets 135
B. Structure et opérateurs des objets 139

TABLE DES FIGURES

Figure	Page	Figure	Page
1	4	2.5	44
2	4	2.6	50
3	10	2.7	52
1.1	19	3.1	66
1.2	21	3.2	68
1.3	21	3.3	71
1.4	23	3.4	73
1.5	24	3.5	74
1.6	25		
1.7	25	4.1	93
1.8	26	4.2	93
1.9	33	4.3	99
		4.4	100
2.1	39	4.5	108
2.2	39		
2.3	42	B.1	139
2.4	43		

INTRODUCTION

Introduction

«Il y a naturellement une difficulté lorsqu'une personne a été trop influencée par des sources différentes: les gens y appartenant prétendront que cette personne n'a pas compris ce qu'ils étaient en train de faire, et que le résultat net est une sorte de "pot-pourri". Ils ont probablement raison...!»

J. ABRIAL

1. Motivation

L'évolution des systèmes de *conception assistée par ordinateur* (CAO) a été similaire à celle des autres systèmes qui utilisent les ordinateurs pour le traitement intensif d'informations volumineuses dans les différents domaines scientifiques, sociaux ou commerciaux. Vers la fin des années 50, ils étaient composés d'un ensemble de programmes indépendants les uns des autres, dans ce sens que chaque programme créait, utilisait et gérait ses propres données, en utilisant un format défini par les concepteurs du programme en fonction de leurs besoins particuliers.

Ceci impliquait, évidement, la duplication de plusieurs informations et compliquait le contrôle de leur cohérence, car l'exécution de tout programme mettant à jour une information donnée devait être suivie par l'exécution de tous les programmes "propriétaires" des informations dupliquées.

En vue d'éviter ces duplications, et de coupler les différents programmes de CAO pour constituer un seul système intégré, des nouveaux programmes "traducteurs" ont été développés pour changer le format des données que les programmes de CAO devaient partager. Ainsi, dans le cas le moins favorable, le couplage de N programmes requérait le développement de N^2-N traducteurs (*voir* figure 1). Dans ces conditions, l'évolution du système général de CAO, l'intégration de nouveaux programmes, et la maintenance des programmes existants, sont bientôt devenues problématiques.

A partir des années 70, certains formats standard tels que CIF (Caltech Intermediate Form), SDL (Stanford Design Language) et, plus récemment (1983), EDIF (Electronic Design Interchange Format), ont été définis, et leur utilisation pour stocker dans un réservoir commun les données de l'application a permis de réduire à 2N (dans le cas le moins favorable) le nombre de traducteurs nécessaire pour coupler les différents programmes de CAO (*voir* figure 2).

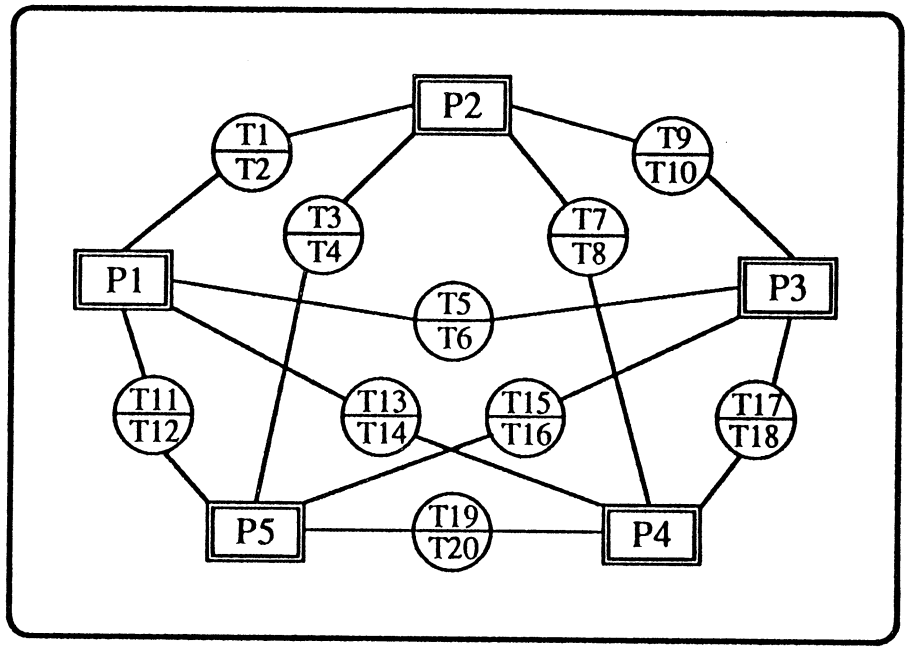


FIGURE 1

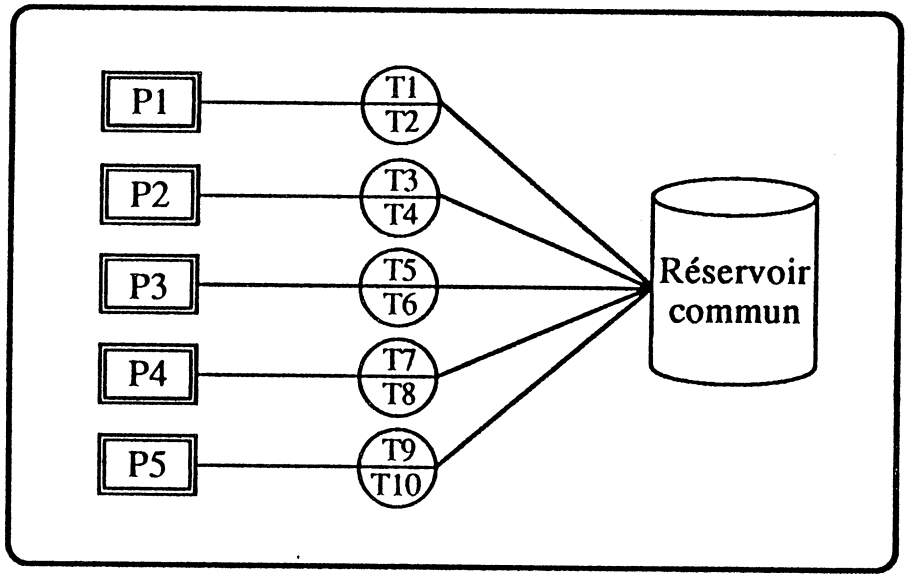


FIGURE 2

Cette solution n'éliminait pourtant pas le besoin de définir, dans chacun des programmes, les différentes routines nécessaires pour accéder aux informations et, en conséquence, toute modification de la structure ou de l'organisation des données requérait la modification des programmes accédant à ces informations.

En même temps, des objectifs et des problèmes similaires dans d'autres domaines de l'informatique avaient déjà donné lieu au développement de

systèmes de gestion de bases de données (SGBD) qui permettaient, entre autres, la structuration et le partage des données, le contrôle de l'accès aux informations, la mise en vigueur de standards, la reprise en cas de panne et, dans certains cas, l'accès aux informations au moyen de langages non procéduraux.

Actuellement la communauté de la CAO considère que l'emploi de bases de données est un moyen adéquat pour intégrer dans un seul système les différents programmes utilisés dans les processus de conception, et que l'un des composants essentiels des systèmes intégrés est un outil pour gérer et pour contrôler l'intégrité sémantique de ces bases (*Cf.* par exemple [Buch 84], [CFHL 86], [Davi 81], [East 81], [Katz 83], [McNB 83], [Merm 85] et [NewS 86]), mais cette communauté considère aussi que les SGBD conventionnels ne sont pas complètement appropriés aux environnements de conception.

En fait, ces SGBD ont été aussi mis en cause dans d'autres domaines de l'informatique, essentiellement parce que les concepts structuraux (*e.g.*, enregistrement, fichier, n-uplet et relation) et opératoires (*e.g.*, ouvrir-fichier, insérer-n-uplet et créer-relation) de leur "modèle de données" (*e.g.*, le modèle hiérarchisé, réseau ou relationnel) ne permettent pas de représenter dans les bases de données certaines informations ou connaissances concernant les propriétés statiques et dynamiques des applications (*e.g.*, l'existence d'objets classés de manières multiples, et des conditions qui doivent être satisfaites avant de modifier la structure d'un objet composite).

Cette situation a été à l'origine d'un grand nombre de recherches visant le développement de nouvelles infrastructures de modélisation, qui de manière générique ont été appelées **modèles sémantiques de données** (*e.g.*, le modèle relationnel binaire de J. Abrial [Abri 74], le modèle Entité-Association [Chen 76], RM/T [Codd 79], SDM [HamM 81] et SHM [SmiS 77]).

Les SGBD qui implémentent ces modèles sémantiques sont encore très peu nombreux, ne sont utilisés que comme véhicules de recherche, et n'abordent pas les problèmes particuliers à la CAO. Néanmoins, l'importance des résultats obtenus a motivé, plus récemment, le développement de nouvelles infrastructures pour la gestion de données en CAO (*e.g.*, [BatB 84], [Katz 82] et [Rieu 85]), et c'est dans cet axe de recherche que se situe le travail décrit dans cette thèse.

En particulier, nous présentons une infrastructure de représentation de connaissances qui intègre, adapte et étend des concepts et des techniques développés dans les domaines des bases de données, de l'intelligence artificielle et des langages de programmation. Une description sommaire des recherches connexes dans ces trois domaines, ainsi que la portée de nos

recherches et l'organisation de cette thèse sont présentées dans la suite de cette introduction.

2. Recherches connexes

La représentation de connaissances est l'un des problèmes fondamentaux des sciences de l'informatique. Bien que dans les domaines des bases de données, de l'intelligence artificielle, et des langages de programmation, des approches différentes aient traditionnellement été utilisées dans les recherches sur ce sujet, l'interaction entre ces domaines n'est pas nouvelle.

Dans leur analyse concernant les similitudes et les différences entre les recherches sur la représentation de connaissances en intelligence artificielle et la modélisation de bases de données, H. Wong et J. Mylopoulos [WonM77] font référence à des recherches qui ont été réalisées, depuis 1963, en intégrant des concepts et des techniques empruntées à ces deux domaines. J. King [King 83] présente aussi environ 30 projets de recherche qui témoignent de l'interaction entre les domaines de l'intelligence artificielle et des bases de données.

D'autres exemples sont la conférence de Salt Lake City [Orga76 et Wegb77], qui a eu comme objectif de favoriser la communication entre les domaines des bases de données et des langages de programmation, de la même manière que le workshop de Kiawah Island [Kers 84] et la conférence de Charleston [Kers 86] ont réuni des chercheurs appartenant aux domaines des bases de données et de l'intelligence artificielle. Enfin, les workshops de Pingree Park [BroZ 80] et d'Intervale [BrMS 84] ont été réalisés dans le but d'intensifier la communication entre les chercheurs de ces trois domaines.

Ces conférences et réunions internationales ont mis en évidence que les recherches relatives à la représentation de connaissances en intelligence artificielle, la modélisation de bases de données, et la spécification de programmes, sont de plus en plus orientées vers le développement de concepts, de techniques et d'outils permettant la **modélisation conceptuelle** des applications, c'est-à-dire leur description en termes plus abstraits, naturels et précis.

Le travail de recherche décrit dans cette thèse a été réalisé avec cette orientation, dans ce sens que nous nous sommes intéressés au développement d'une infrastructure permettant de définir, de manière naturelle, la signification conceptuelle des informations créées et manipulées dans les environnements de CAO, afin que le système qui gère ces informations puisse contrôler leur intégrité sémantique.

En vue de caractériser de manière plus précise le travail que nous avons réalisé, nous décrivons, dans la suite de cette section, un certain nombre de similitudes et de différences importantes entre les infrastructures de modélisation développées dans les trois domaines de recherche mentionnés ci-dessus.

2.1. Bases de données

Plusieurs recherches dans le domaine des bases de données ont traité les problèmes relatifs à la spécification des propriétés logiques des systèmes d'information, en des termes naturels, c'est-à-dire telles qu'elles sont perçues par l'être humain (*e.g.*, des personnes, des outils et des événements), et non en termes de leur éventuelle implémentation dans un ordinateur (*e.g.*, des adresses mémoire, des fichiers et des pointeurs).

Les résultats de ces recherches ont été en général des nouvelles infrastructures qui dans cette domaine sont appelées **modèles de bases de données** (ou tout simplement modèles de données), et qui présentent des concepts et des techniques permettant l'emploi de divers **mécanismes d'abstraction** dans la modélisation des applications.

La première notion d'abstraction dans ce domaine fut l'**indépendance des données**, dont l'objectif est de séparer la spécification des propriétés logiques de l'application, de la spécification concernant leur représentation physique. Pour atteindre cet objectif, le modèle réseau introduit le concept de schéma, qui sert d'intermédiaire entre le niveau d'abstraction correspondant aux spécifications logiques et celui des spécifications physiques. Avec le même objectif, le modèle relationnel présente un formalisme qui permet de «décrire les données en termes de leur structure naturelle -c'est-à-dire sans superposer aucune structure additionnelle relative à leur représentation dans une machine» [Codd 70].

Actuellement, plusieurs modèles sémantiques ont adopté une approche centrée-objet, et intégré d'autres mécanismes d'abstraction tels que la généralisation et l'agrégation[1], en vue de permettre de décrire la sémantique des applications d'une manière encore plus naturelle et plus indépendante des aspects d'implémentation. Dans notre infrastructure de représentation de connaissances, la modélisation d'applications de CAO est aussi un processus centré-objet, basé sur l'emploi extensif et étendu des mécanismes d'abstraction que Diane et John Smith [SmiS 77] ont introduits dans le domaine des bases de données.

[1]Ces deux mécanismes seront décrits au chapitre 2.

Une caractéristique importante des infrastructures développées dans ce domaine est que le terme **base de données** est utilisé, en général, pour dénoter une collection de valeurs auxquelles on peut accéder en utilisant un **schéma conceptuel** défini auparavant. Ce schéma contient une description de l'organisation logique de la base, et des contraintes sur les valeurs qui peuvent y exister. La base de données et son schéma conceptuel constituent, ensemble, un **modèle conceptuel** des propriétés logiques d'une application ou système particulier, mais ce modèle n'est pas manipulé de manière uniforme.

En effet, dans le domaine des bases de données, les **connaissances concrètes**, concernant les propriétés structurelles des objets d'une application quelconque (*e.g.*, «le salaire du concepteur Paul est 50 000 FF»), sont stockées dans la base de données et manipulées à l'aide du *langage de manipulation de données* (LMD), tandis que les informations génériques ou **connaissances abstraites** (*e.g.*, «si x est une instance de la classe Concepteur, son salaire doit être entre 20 000 FF et 60 000 FF») constituent le schéma conceptuel et sont manipulées à l'aide du *langage de définition de données* (LDD).

Par ailleurs, les propriétés opératoires particulières aux objets de chaque application ne sont en général représentées ni dans la base ni dans le schéma conceptuel. Elles doivent être définies de manière implicite dans les programmes qui modifient la base de données, car la plupart des infrastructures n'incluent que des concepts pour modéliser la structure des objets. Quelques exceptions sont le modèle de J. Abrial [Abri 74], SHM+ [Brod 81], le modèle des événements [KinM 84] et l'infrastructure que nous décrivons dans cette thèse.

2.2. Intelligence artificielle

Dans le but de simuler le comportement intelligent, les chercheurs du domaine de l'intelligence artificielle ont eu besoin de représenter dans les ordinateurs aussi bien des connaissances abstraites concernant le monde réel, que les mécanismes nécessaires pour manipuler ces connaissances, afin que l'ordinateur puisse comprendre des situations spécifiques qu'il peut trouver [Borg84 et Mylo80].

L'un des problèmes essentiels dans ces travaux est le développement de notations suffisamment précises pour représenter ces connaissances. Les notations sont appelées **schémas de représentation de connaissances** et sont utilisées pour spécifier une **base de connaissances**, qui est souvent considérée comme un modèle (partiel) d'un univers conceptuel appelé univers du discours ou domaine de l'interprétation.

2.2.1. Les schémas de représentation de connaissances

Les schémas de représentation de connaissances sont souvent regroupés dans trois grandes catégories: les déclaratifs, les procéduraux et ceux basés sur la notion de **schéma** (*en anglais*: frame) proposée par M. Minsky [Mins 75]. Parmi les schémas déclaratifs, l'on trouve ceux qui utilisent la logique (de premier ordre, d'ordre supérieur, multivaluée, modale,...) comme formalisme de modélisation, et ceux basés sur la notion de **réseau sémantique** [Quil 68].

Dans ce dernier cas, l'univers conceptuel peut être représenté, de manière **centrée-objet**[1], par un graphe dont les nœuds correspondent aux objets de cet univers, et les branches correspondent aux associations entre ces objets. Cette approche de représentation de connaissances, qui a la vertu d'être en même temps simple et bien structurée, a eu une influence primordiale dans le développement du paradigme de modélisation que nous présentons dans cette thèse.

La conception des premiers schémas déclaratifs a été guidée par l'intérêt de trouver la manière de représenter des connaissances "statiques" (*e.g.*, la structure des objets et leurs associations), tandis que les schémas procéduraux ont été conçus pour représenter principalement la façon dont ces connaissances statiques doivent être utilisées (pour trouver les faits pertinents, pour faire des inférences, etc.).

D'autre part, dans les représentations basées sur la notion de schéma, l'univers conceptuel est représenté par une collection de structures logiques (les schémas) qui contiennent des connaissances, concrètes ou abstraites, concernant les objets de cet univers dans des situations stéréotypées (*e.g.*, des personnes dans un restaurant, ou des voitures sur une autoroute).

L'une des principales caractéristiques des schémas est qu'ils comportent des aspects aussi bien déclaratifs que procéduraux, ce qui leur confère une puissance expressive très importante. En effet, chaque schéma contient un certain nombre de "slots" avec des "facettes" différentes qui décrivent les objets qui jouent un rôle dans des situations stéréotypées, comme celles mentionnées auparavant, ainsi que les associations entre ces objets (*voir* figure 3), mais aussi la manière dont ce schéma peut être utilisé.

Cette dernière "fonctionnalité" est mise en œuvre au moyen d'une technique de représentation de connaissances connue comme **attachement procédural**,

---

[1]Dans le domaine de l'intelligence artificielle, l'adjectif "centrée-objet" est utilisé pour qualifier des représentations dans lesquelles les faits ou assertions son indexés par des termes qui dénotent des entités ou objets de l'univers du discours [Nils 82]. Cette notion sera décrite, avec plus de détail, dans la section 2 du chapitre 2.

qui consiste à définir des facettes dont la valeur est une fonction (*e.g.*, Calculer-Durée, dans la figure 3) ou une procédure qui peut être utilisée, entre autres, pour inférer des nouvelles connaissances, et pour enchaîner plusieurs opérations à la suite de certains événements tels que l'ajout de nouvelles connaissances dans la base, et la modification des connaissances existantes.

SCHEMA Voyage		/* Contient des connaissances abstraites */
lui-même	un	Evénement;
voyageurs	liste-de	Personnes;
ville-départ	une	Ville
	défaut	Grenoble; /* cette facette définit la valeur par défaut de la ville de départ */
date-départ	une	Date;
date-arrivée	une	Date;
durée	un	Entier
	si-besoin	Calculer-Durée avec (date-départ date-arrivée).
SCHEMA voyage-125		/* Contient des connaissances concrètes */
lui-même	(Voyage)	
voyageurs	(Jacques Annie)	
date-départ	(27.04.87)	
date-arrivée	(30.06.87)	

FIGURE 3

Exemple d'une représentation de connaissances, basée sur la notion de schéma, dans laquelle un voyage est défini comme un cas particulier d'événement, caractérisé par le fait d'avoir un certain nombre de voyageurs qui sont des personnes, une ville de départ qui par défaut est Grenoble, et une durée dont la valeur peut être connue, si elle n'est pas spécifiée, au moyen d'une fonction (Calculer-Durée) dont les paramètres sont la date de départ et la date d'arrivée du voyage. Cette fonction doit être utilisée dans le cas du voyage-125, car sa durée n'a pas été spécifiée.

Dans le domaine de l'intelligence artificielle, les notions de schéma et d'attachement procédural ont inspiré le développement de plusieurs systèmes de modélisation tels que GUS [B&al 77], KRL [BobW 77], KEE [FikK 85], Shirka [Rech 85] et FRL [RobG 77], et cette thèse suggère que l'emploi de ces notions peut améliorer significativement le contrôle de l'intégrité des informations crées et manipulées dans les environnements de CAO.

2.2.2. Les bases de connaissances

A différence des bases de données, les bases de connaissances contiennent, en général, aussi bien des informations concrètes que des informations abstraites, et les informations des deux types peuvent être spécifiées et manipulées de manière uniforme, en utilisant le même schéma de représentation de connaissances.

Cette approche permet d'utiliser les informations abstraites non seulement pour vérifier l'intégrité sémantique de la base, mais aussi pour inférer des nouvelles connaissances (*e.g.*, «si x est une instance de la classe Concepteur, son salaire doit être entre 20 000 FF et 60 000 FF, et par défaut celui-ci peut être supposé égal à 20 000 FF»). Il faut cependant remarquer que la simulation de ces processus cognitifs requiert, en plus, la mise en œuvre de divers **mécanismes de raisonnement** (*e.g.*, raisonnement par circonscription [McCa 80], par défaut [Reit 80], et par abstraction[1]) qui ne sont en général pas présents dans les SGBD.

La manipulation uniforme des informations concrètes et des informations abstraites (respectivement "données" et "meta-données", dans la terminologie des bases de données) permet aussi aux utilisateurs de formuler des requêtes avec une connaissance moins précise de l'organisation de la base (*e.g.*, «{retrouver le salaire de toutes les instances de {retrouver le nom de la classe ayant la propriété "salaire"}}»), et en même temps les protège de certaines modifications du "schéma conceptuel" (*e.g.*, le changement du nom de la classe ayant la propriété "salaire", dans l'exemple précédent).

Cette possibilité de combiner, dans la même requête, des expressions relatives aux informations concrètes, avec celles concernant les informations abstraites, est aussi considérée importante et nécessaire aux applications de bases de données en général [D&al 85], et de CAO en particulier [StaA 86], mais elle n'est pas présente dans les modèles de données. D'après nos connaissances, le seul modèle qui permet la manipulation uniforme de la base de données et du schéma conceptuel est RM/T.

L'intégration de connaissances abstraites et de connaissances concrètes, ainsi que l'emploi de mécanismes de raisonnement dans les processus de gestion et de contrôle de l'intégrité de ces connaissances, sont deux notions fondamentales dans cette thèse, et notre approche est essentiellement basée sur les travaux réalisés en intelligence artificielle.

---

[1] Le mécanisme de raisonnement par abstraction sera décrit au chapitre 3.

2.3. Langages de programmation et génie logiciel

Dans les domaines des langages de programmation et du génie logiciel, la notion d'**abstraction des données** est utilisée pour dénoter un processus qui consiste à spécifier la sémantique des programmes indépendamment de la manière dont ils peuvent être implémentés. Dans ces spécifications, l'univers d'objets d'un programme est fréquemment constitué par des **types abstraits** [Gutt 77]. Chaque type abstrait est une classe d'objets complètement caractérisée par les opérateurs qui peuvent être utilisés pour manipuler les objets de cette classe, et non par les structures de données qui peuvent être utilisées pour les représenter dans un ordinateur. En d'autres termes, la sémantique de ces objets est spécifiée de manière abstraite en termes de leurs propriétés opératoires.

Pour implémenter les types abstraits, certains langages de programmation présentent des concepts tels que «agglomérat» (*en anglais*: cluster [LSAS 77]) et «forme» [ShWL 77], qui sont similaires à (et inspirés de) celui de «classe», du langage Simula [DahH 72]. Chaque agglomérat, par exemple, peut être utilisé pour implémenter un type abstrait, et comporte deux parties que nous appelons l'interface et le corps. L'interface contient les noms des opérateurs qui caractérisent les objets appartenant à ce type abstrait, et le corps définit les algorithmes qui implémentent ces opérateurs, ainsi que les structures de données utilisées dans ces algorithmes.

L'une des propriétés les plus importantes de cette approche de modélisation est la claire distinction et l'indépendance de deux niveaux d'abstraction différents: l'un relatif à l'emploi d'un objet, et l'autre qui concerne son implémentation. Un type abstrait peut être employé sans connaître la manière dont il sera implémenté, et peut être implémenté sans savoir comment il sera utilisé. Tout agglomérat A peut accéder aux objets d'un autre agglomérat B, en invoquant les opérateurs qui caractérisent le type abstrait B, et l'implémentation de ces opérateurs peut toujours être redéfinie (pour améliorer par exemple leurs performances) sans avoir besoin de modifier l'agglomérat A (à condition, bien entendu, que cette nouvelle implémentation satisfasse les spécifications relatifs au type abstrait B).

La corrélation entre ces concepts et la notion d'indépendance des données, dans les bases de données, a été à l'origine de plusieurs recherches (*e.g.*, [BarD 81], [Ston 84], [Wass 78] et [Webe 78]) qui ont influencé nos critères d'implémentation du système expérimental OBMS, présenté dans cette thèse.

La possibilité de pouvoir "stratifier" l'univers d'objets d'un programme, dans des niveaux d'abstraction différents, en vue d'ignorer

(temporairement au moins) certains détails du problème traité et de mettre l'accent sur ceux qui sont les plus significatifs, est aussi un objectif fondamental dans les recherches sur les langages de programmation orientée-objet[1], tels que Smalltalk-80 [GolR 83], Ceyx [Hull 84], Flavors [Moon 86] et Loops [SteB 86].

Dans ces langages, "héritiers" aussi du concept de classe, du langage Simula, mais en même temps influencés par les travaux sur la représentation de connaissances en intelligence artificielle, l'abstraction des données peut être mise en œuvre grâce au paradigme d'**envoi de messages**, qui est une sorte d'appel indirect [SteB 86] aux opérateurs qui peuvent être utilisés pour créer et pour manipuler les objets d'un programme. Grâce à cet appel indirect, l'implémentation de ces opérateurs peut toujours être modifiée sans avoir besoin de changer les programmes qui les utilisent. D'autre part, dans les langages orientés-objet, chaque **classe** représente un type abstrait [SteB 86], et ses **méthodes** décrivent les procédures qui implémentent les opérateurs correspondant aux objets, dits **instances**, de cette classe.

Par ailleurs, l'envoi de messages n'est pas la seule manière de stratifier l'univers d'objets d'un programme. Dans la programmation orientée-objet, plusieurs niveaux d'abstraction peuvent exister dans une direction orthogonale à celle définie par l'abstraction des données, car la spécification et l'implémentation des classes peuvent être réalisées "par raffinements successifs". Une classe peut **spécialiser** une autre classe, définie auparavant, et **hériter** de toutes les méthodes de cette classe plus générale (c'est-à-dire plus abstraite), et celle-ci peut à son tour être une spécialisation d'une classe encore plus générale.

Ce processus organise les classes de façon hiérarchisée, car une classe peut avoir plusieurs spécialisations[2], et l'héritage simplifie aussi bien la spécification que l'implémentation des programmes, car à chaque niveau d'abstraction on n'a besoin de définir que les méthodes qui n'ont pas été définies dans les niveaux supérieurs. Cette thèse montre que l'aspect abréviatif est seulement l'une des vertus de la notion d'héritage.

3. L'approche de cette thèse

La gestion de données dans les environnements de CAO est un processus qui requiert la représentation d'une grande quantité de connaissances, concrètes et abstraites, concernant les propriétés statiques et dynamiques des

[1]L'adjectif "orientée-objet", utilisé dans les langages de programmation, a une connotation tout à fait analogue à celle de l'adjectif "centrée-objet", utilisé principalement en intelligence artificielle (*voir* note 1 de la page 9), et dans cette thèse ils sont utilisés de manière indistincte.

[2]Dans certains langages, une classe peut aussi être spécialisation de plusieurs classes.

applications, et non seulement la spécification d'un ensemble de structures de données; néanmoins, ces connaissances ne peuvent être que partiellement représentées dans les SGBD conventionnels.

Cette thèse suggère l'emploi d'une infrastructure de représentation de connaissances qui intègre des concepts et des techniques développés dans les domaines de l'intelligence artificielle, et des langages de programmation, en vue de simplifier de manière significative les problèmes relatifs à la conception, l'implémentation, la manipulation, le contrôle de l'intégrité sémantique, et l'évolution des bases de données des applications de CAO.

Dans cette infrastructure, la gestion de données est vue comme un processus de modélisation dynamique qui consiste à représenter, dans des **bases d'objets**, l'état et l'évolution permanente des environnements de conception. Chaque base est considérée comme un modèle conceptuel d'un environnement précis, à un instant donné; les éléments du modèle décrivent les propriétés structurelles et opératoires des objets constituant cet environnement, et toute modification à ce modèle représente l'évolution de l'environnement en question.

Le modèle conceptuel défini dans la base d'objets est traité comme un réseau sémantique, et non comme une collection de fichiers ou de relations. En effet, la manipulation de ce modèle consiste à ajouter des objets dans la base, enlever des objets existants, et établir, supprimer ou analyser les associations entre ces objets.

Plusieurs types d'associations peuvent être définis par agrégation et par généralisation. L'emploi de ces deux mécanismes d'abstraction organise les objets dans des hiérarchies similaires à celles des classes dans les langages orientés-objet, avec les objets les plus abstraits placés au-dessus des objets les plus concrets. Dans ces hiérarchies, la notion d'héritage est vue non seulement comme un mécanisme abréviatif, mais aussi comme une contrainte d'intégrité et comme une règle d'inférence de connaissances qui ne sont pas représentées de manière explicite dans la base.

D'autre part, la notion de généralisation est étendue pour permettre la définition d'associations caractéristiques des objets des environnements de CAO, telles que les associations entre un objet en cours de conception et ces différentes versions, alternatives, révisions, etc.

La sémantique des objets peut être déterminée en fonction de leurs associations avec d'autres objets, car toute association est caractérisée par un certain nombre de contraintes sur la structure et sur le comportement des objets associés. Ainsi, le contrôle de l'intégrité sémantique de la base est un processus qui consiste à vérifier que les objets satisfont les contraintes inhérentes aux associations auxquelles ils participent. Ces contraintes sont

représentées avec une approche structurelle similaire à celle des schémas en intelligence artificielle.

Bien que nous ayons développé un système expérimental de gestion de bases d'objets, qui implémente cette infrastructure de représentation de connaissances, ce travail a été exclusivement réalisé dans le but de valider les concepts et les techniques que nous présentons dans cette thèse. Autrement dit, la portée de nos recherches n'a pas inclus l'analyse des différents services que tout SGBD doit fournir aux utilisateurs afin d'assurer l'intégrité physique de la base, tels que la sauvegarde des informations et la reprise après panne logicielle ou matérielle. Par ailleurs, nous ne traitons non plus ni les problèmes du contrôle de l'accès concurrent à la base ou à des informations confidentielles, ni les problèmes introduits par la distribution des activités de conception à l'aide d'un serveur et de plusieurs stations de travail.

Enfin, l'infrastructure de représentation de connaissances présentée dans cette thèse à été essentiellement conçue pour proposer des solutions aux problèmes relatifs à la conception, l'implémentation, la manipulation, le contrôle de l'intégrité sémantique, et l'évolution des bases de données en CAO.

4. Organisation de la thèse

Après avoir décrit dans cette introduction les approches de représentation de connaissances qui ont été plus récemment adoptées dans les domaines des bases de données, de l'intelligence artificielle et des langages de programmation, nous allons décrire, dans le chapitre suivant, les fonctionnalités requises dans les environnements de conception, en ce qui concerne la gestion de données, ainsi que les différentes approches qui ont été adoptées pour intégrer ces fonctionnalités dans un SGBD.

Le Chapitre 2 définit et analyse les concepts fondamentaux de nôtre infrastructure de représentation de connaissances, et met en relief leurs principales similitudes et différences avec des concepts analogues utilisés dans les domaines de recherche mentionnés dans cette introduction.

Les chapitres 3 et 4 montrent la façon dont ces concepts fondamentaux peuvent être utilisés pour satisfaire les exigences des environnements de CAO, mentionnées au chapitre 1. Bien que les exemples présentés dans ces chapitres fassent référence à des concepts caractéristiques des environnements de conception de systèmes électroniques (e.g., des additionneurs, des portes logiques, etc.), ces exemples ont été intentionnellement simplifiés afin de permettre leur compréhension sans

avoir besoin de connaître les particularités de tels systèmes, et de les rendre suffisamment généraux donc applicables à d'autres domaines de la CAO.

Le chapitre 3 décrit, en plus, un paradigme de modélisation centré-objet qui permet de définir, de manière simple, la sémantique des applications de CAO, et montre que l'infrastructure de représentation de connaissances permet de développer un système capable de contrôler un grand nombre de contraintes d'intégrité, et de s'adapter à des environnements différents et aux évolutions de ces environnements.

Le chapitre 4 introduit la notion d'objets génériques et décrit une extension à la notion d'«abstraction par généralisation», en vue de permettre la modélisation d'objets qui peuvent avoir plusieurs versions, alternatives, variantes et d'autres classes d'objets représentatifs. Finalement, les conclusions de cette thèse incluent une évaluation du travail réalisé, ainsi que des suggestions pour des recherches futures.

CHAPITRE 1

Gestion de données en CAO

1. Gestion de données en CAO

«La question pertinente concernant l'emploi de base de données en CAO semble ne plus être "Pourra-t-on utiliser des bases de données pour la CAO?" mais plutôt "De quelle manière nouvelle et intelligente peut-on étendre leur utilisation et comment peut-on améliorer les SGBD existants?"»

A. BUCHMANN

1.1. Introduction

Tout processus de *conception assistée par ordinateur* (CAO) implique l'emploi méthodologique d'un ensemble d'outils informatiques pour définir les propriétés d'un objet qui n'existe pas, mais que l'on veut réaliser (*e.g.*, un circuit électronique, un bâtiment, ou un processus chimique), et qui doit satisfaire un certain nombre de contraintes (physiques, thermodynamiques, fonctionnelles,...) spécifiées dans un document connu sous le nom de *cahier des charges.*

L'aspect méthodologique définit essentiellement le nombre d'étapes constituant le processus global de conception, leur ordre, leur type de résultats et les conditions qui doivent être satisfaites pour aller d'une étape à la suivante, ou qui peuvent impliquer des retours aux étapes précédentes; son rôle est de garantir que l'ensemble des résultats finals de toutes les étapes constitue une description complète et correcte (par rapport aux spécifications du cahier des charges) des différentes propriétés de l'objet que l'on veut réaliser (*voir* figure 1.1).

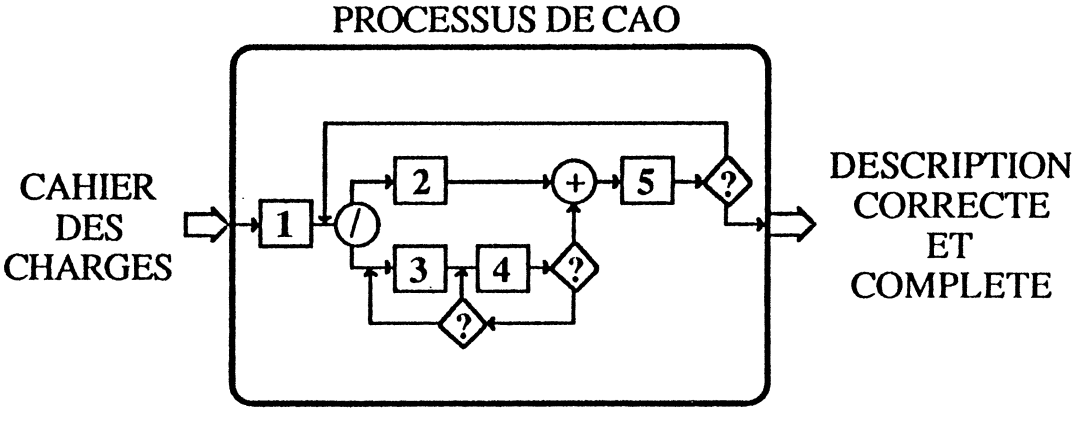


FIGURE 1.1

Cette caractérisation simplificatrice de tels processus montre que tout système de CAO doit comporter, au moins, des outils pour:

a) réaliser chacune des étapes de conception (*e.g.*, des éditeurs, des compilateurs et des simulateurs),
b) mettre en vigueur les contraintes méthodologiques (*e.g.*, un système de gestion de processus de conception),
c) manipuler des informations persistantes, c'est-à-dire celles qui existent avant et/ou après l'exécution des programmes (*e.g.*, un système de gestion de données).

L'existence de tous ces outils doit, en principe, être transparent aux concepteurs, lorsque le système de CAO est un système intégré [Merm 85]. Ces concepteurs doivent plutôt dialoguer avec un Système de Gestion de Processus de Conception (*en anglais:* Design Management System), qui contrôle l'emploi des autres outils en fonction, par exemple, des ressources disponibles, des contraintes méthodologiques, et des droits d'accès de chaque utilisateur. Mais cela n'est malheureusement pas l'état de l'art; en réalité la gestion des processus de conception est généralement effectuée en utilisant des procédures manuelles [BhaK87, East81, NoRR 82].

En tout cas, les processus de conception et sa gestion (manuelle, automatique ou semi-automatique), ainsi que la communication entre les différents outils et le contrôle de l'intégrité sémantique des informations persistantes, requièrent l'emploi intensif de connaissances concernant, entre autres, la ou les méthodologies de conception qui peuvent être utilisées, le rôle des outils dans chacune des étapes du processus global de conception, les associations qui existent ou qui peuvent exister entre les différents objets crées et manipulés par ces outils, et les contraintes sur la structure de ces objets et sur la manière dont ils peuvent être manipulés.

Plusieurs recherches (*e.g.*, [BD3 83], [Buch 84], [East 80], [Katz 83], [L&al 85], et [McNB 83]) ont pourtant montré que les systèmes conventionnels de gestion de données ne permettent pas de représenter et d'utiliser ces connaissances de manière convenant aux systèmes de CAO. Chacune de ces recherches mentionne un certain nombre de fonctionnalités que nous allons décrire dans la suite de ce chapitre, et qui sont considérées comme nécessaires dans les environnements de conception.

1.2. Définition et manipulation d'objets composites

Dans les processus de CAO, les concepteurs ont souvent besoin de représenter dans la base de données le fait qu'un objet est composé de plusieurs dizaines, centaines, voire millions [BD3 83] d'objets plus simples mais peut-être composites aussi, et de types différents.

Un additionneur de huit bits, par exemple, peut être composé de huit additionneurs d'un bit (*voir* figure 1.2). Chaque additionneur d'un bit peut aussi être composé de deux demi-additionneurs et d'une porte NAND (*e.g.*, da-1, da-2 et n3, dans la figure 1.3). Ces demi-additionneurs peuvent être des objets composés d'une porte XOR (x1 et x2, dans la figure 1.3) et d'une porte NAND (n1 et n2, dans la figure 1.3), et ainsi de suite (*e.g.*, des portes logiques composées des transistors, etc.).

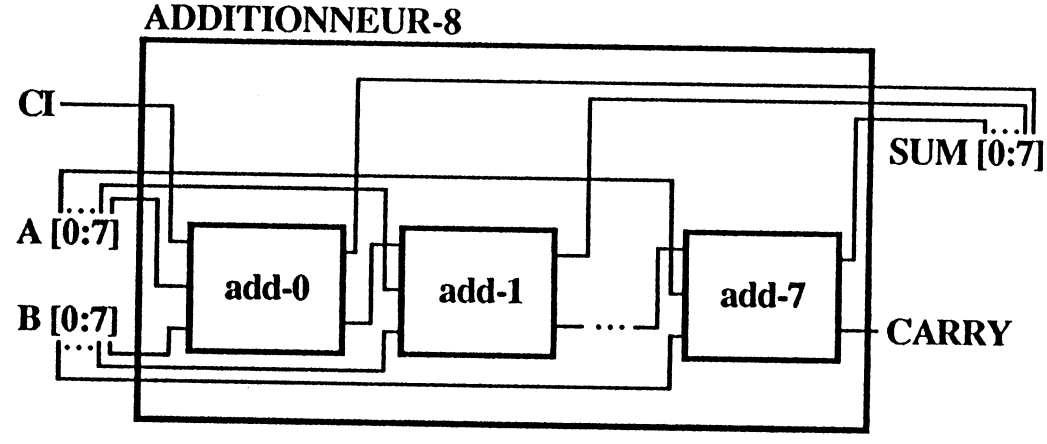


FIGURE 1.2
Exemple d'un objet composite.

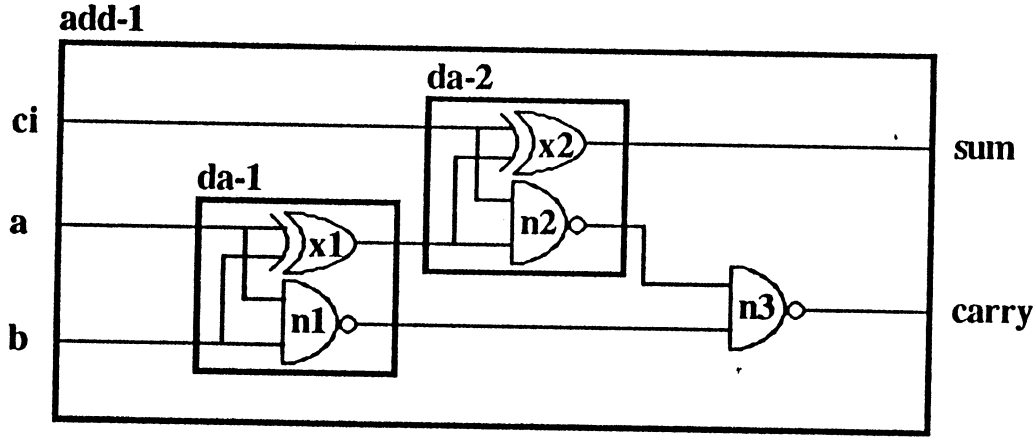


FIGURE 1.3
Structure interne d'un composant de l'objet ADDITIONNEUR-8, illustré dans la figure 1.2.

Dans de telles conditions, les concepteurs ont besoin de compter sur un *Système de Gestion de Bases de Données* (SGBD) permettant de définir ces associations entre les objets composites et leurs composants, mais en termes

naturels, c'est-à-dire tels qu'ils les perçoivent au niveau conceptuel, et non en termes de leur implémentation dans l'ordinateur (*e.g.*, des adresses mémoire, des fichiers, des pointeurs).

D'autre part, accéder à ces objets composites avec un schéma de représentation basé sur l'emploi d'enregistrements, ou de n-uplets, impliquerait l'accès à des instances multiples de fichiers ou relations éventuellement distincts. Il est donc aussi souhaitable que le SGBD soit capable d'identifier et de regrouper ces composants, et de permettre aux concepteurs de référencer et caractériser cette collection de composants comme une seule unité.

1.3. Coexistence de descriptions multiples d'un objet

La CAO implique de longs processus comportant des actions complexes et heuristiques, de nature "provisoire et itérative" [East 80]. Très souvent les concepteurs expérimentent plusieurs alternatives de développement ou "versions" d'un **objet en cours de conception** (OCC).

Dans les grands projets de CAO, plusieurs groupes peuvent même être en train de générer simultanément des versions distinctes d'un même OCC, et une ou plusieurs versions peuvent être transférées à la prochaine étape de développement, lorsqu'elles satisfont les contraintes spécifiées comme des critères de conception. En d'autres termes, il n'y a en général pas une solution unique pour chaque problème de conception. Plusieurs solutions peuvent même être optimales lorsque l'on affecte des poids distincts aux contraintes de conception [StaA 86].

Par ailleurs, l'aspect itératif implique en général des retours aux étapes de conception précédentes, mais en suivant une chemin de développement alternatif. L'évolution d'un OCC n'est pas donc linéaire; elle peut être représentée par une hiérarchie similaire à celle de la figure 1.4, où le numéro de chaque version indique l'ordre de sa création. Ainsi, dans la première étape de conception, deux versions ont été créées, mais seulement la deuxième a été examinée dans les étapes 2 et 3. A ce stade, il y a eu un retour vers la première étape, pour générer la cinquième version qui a été finalement abandonnée. Un nouveau chemin de développement a été alors examiné en générant la sixième version.

D'autre part, cet exemple montre que l'abandon de certaines versions n'implique pas forcément sa destruction. La version 2, par exemple, a été reconsidérée lors de l'abandon de la version 5. Par conséquent, le SGBD doit permettre la coexistence de toutes les versions des OCC dans la base de données.

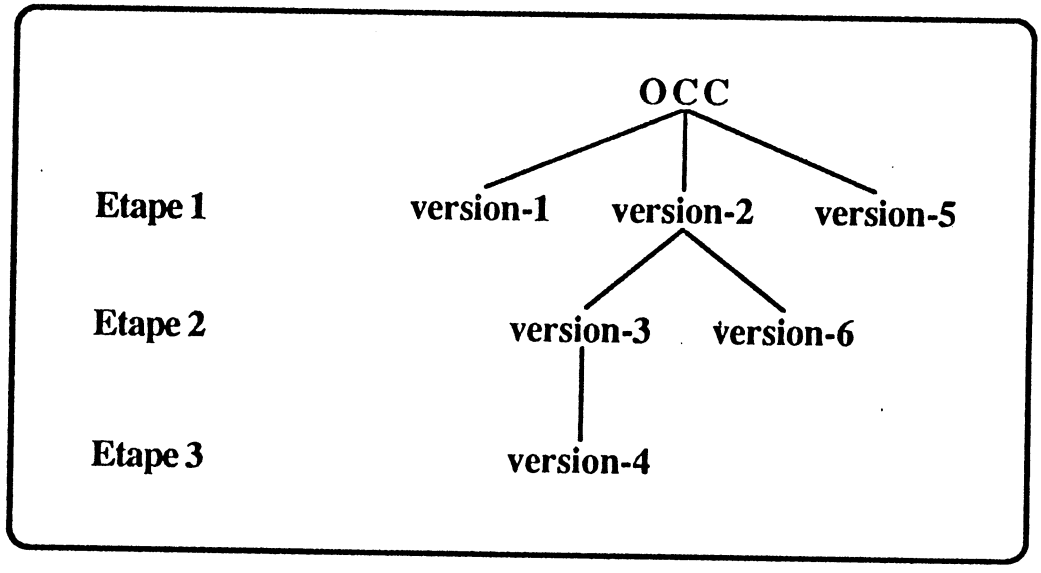


FIGURE 1.4
Evolution possible d'un objet en cours de conception.

Peter Klahold et ses collègues mentionnent dans [KlSW 86] que la notion de versions est de grande importance dans les environnements de CAO, car elle est utilisée aussi bien pour illustrer le cycle de vie des OCC, que pour documenter le processus de développement. D'autre part, dans [Katz 83] l'auteur montre que l'emploi de versions peut simplifier grandement l'implémentation des mécanismes de reprise en cas de panne et de contrôle de l'accès concurrent.

Suivant la méthodologie utilisée, les différentes "versions" d'un OCC peuvent être regroupées et dénommées de manières distinctes, en accord avec des critères divers tels que le type de description qu'elles contient, les propriétés qu'elles décrivent, et le niveau de détail avec lequel elles le font.

Dans la plupart des méthodologies utilisées pour la CAO de systèmes VLSI, par exemple, chaque OCC peut avoir plusieurs "représentations" (au niveau système, au niveau portes logiques, au niveau électrique, etc.). Chaque représentation peut avoir un certain nombre de "variantes" décrivant les possibles implémentations de l'OCC, et ces variantes peuvent avoir de multiples "révisions" (*voir* figure 1.5).

En réalité, l'organisation et la dénomination d'objets illustrée dans la figure 1.5 n'est que l'une des multiples possibilités. D'autres organisations plus ou moins complexes ont été proposées, en utilisant des noms tels que "alternatives", "descriptions", "vues", "perspectives", "versions paramétrées", et "répliques" (*Cf.* par exemple [BatK 85], [JuLL 86], [KaAC 86] et [RieN 86]).

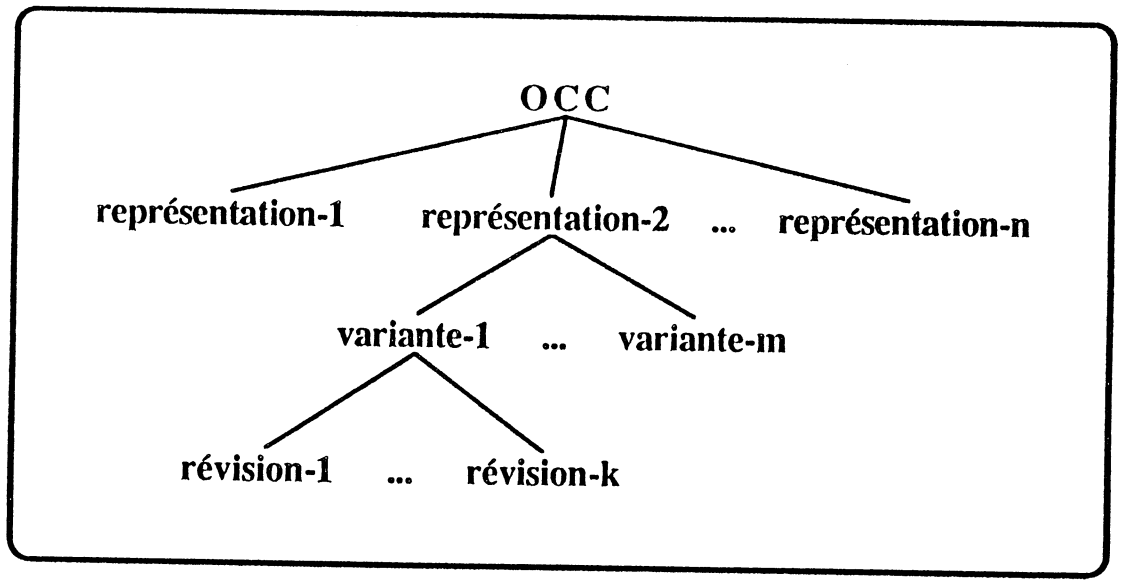


FIGURE 1.5
Exemple d'organisation et de dénomination des différentes descriptions d'un objet en cours de conception.

Dans tous les cas, la possibilité de pouvoir faire coexister dans la même base de données les différentes versions, alternatives, vues, révisions, etc., de chaque objet, et représenter toutes leurs associations, est l'une des principales fonctionnalités nécessaires en CAO, car elle est cruciale pour la gestion des processus de conception [KatL82 et McNB 83] .

1.4. Organisation multi-hiérarchisée des objets

L'organisation d'objets illustrée dans la figure 1.5 n'est pas le seul type d'organisation hiérarchisée dont on a besoin dans les environnements de CAO. En effet, les associations entre les objets composites et leurs composants organisent aussi les objets de façon hiérarchisée. Par exemple, les associations entre les objets illustrés dans les figures 1.2 et 1.3 constituent la hiérarchie montré dans la figure 1.6.

De plus, les concepteurs ont aussi besoin de regrouper les objets en fonction de leurs similitudes structurelles et/ou opératoires (*e.g.*, les additionneurs, les portes logiques, les multiplieurs, etc.), et ce processus peut générer des nouvelles hiérarchies d'objets, car il peut être appliqué de façon récursive aux groupes d'objets définis auparavant (*voir* figure 1.7).

Ces exemples mettent en évidence le fait que les SGBD des environnements de conception doivent permettre la définition et la coexistence de diverses formes d'organisation hiérarchisée des objets de la base de données.

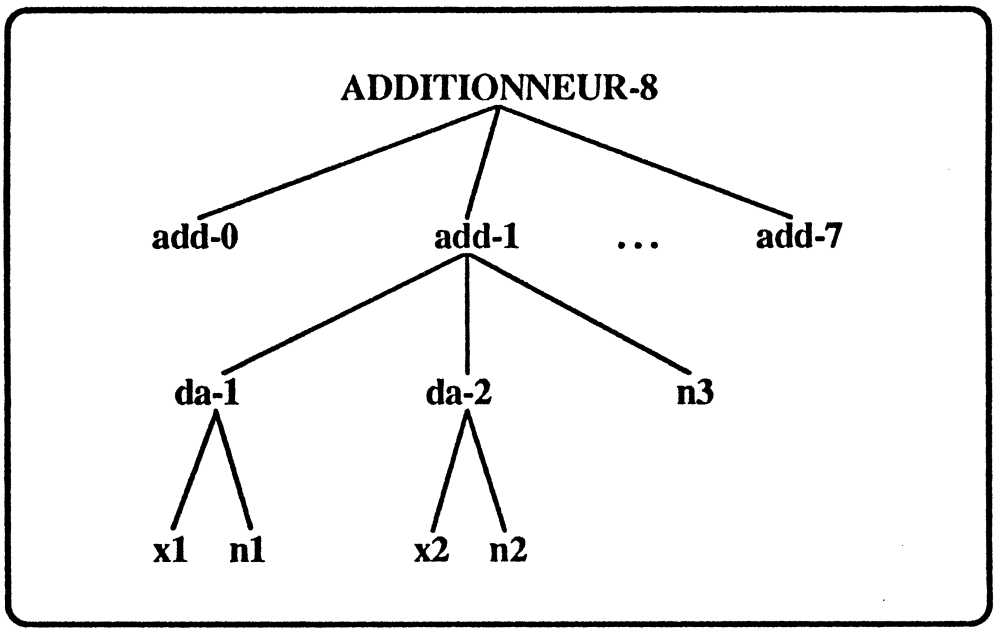


FIGURE 1.6
Organisation hiérarchisée établie par les associations
de composition illustrées dans les figures 1.2 et 1.3.

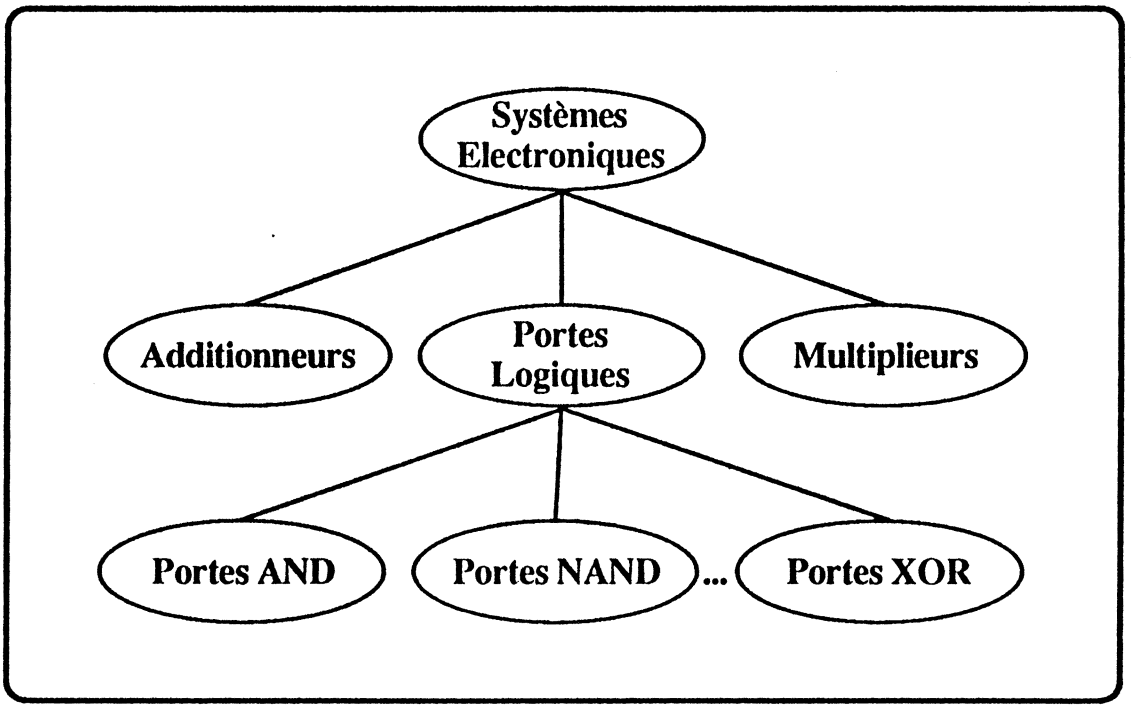


FIGURE 1.7
Exemple d'une organisation hiérarchisée des objets basée
sur leurs similitudes structurelles et/ou opératoires.

1.5. Modification du schéma conceptuel

Les SGBD conventionnels présentent un *langage de définition de données* (LDD) qui doit être utilisé pour spécifier le type d'enregistrements (ou de n-uplets, dans le cas de systèmes relationnels) qui seront employés pour stocker des informations dans la base de données.

Ces spécifications constituent ce qui est en général appelé le "schéma conceptuel" de la base (*voir* figure 1.8), et incluent le nombre d'éléments de chaque type d'enregistrement, ainsi que le type et la taille ou le format des données qui peuvent constituer la valeur de ces éléments (*e.g.*, entier plus grand que zéro, chaîne de 4 caractères, et réel avec deux décimales).

```
SCHEMA NAME IS  base-1
    . . .
    RECORD NAME IS   Projet;
        . . .
        02 nom;              TYPE IS CHARACTER  13
        02 responsable;      TYPE IS CHARACTER  30
        02 budget;           TYPE IS FIXED DECIMAL  8

    RECORD NAME IS   Circuit;
        . . .
        02 numéro;           TYPE IS FIXED DECIMAL  5
        02 concepteur;       TYPE IS CHARACTER  30
        . . .
```

FIGURE 1.8
Spécification partielle du schéma conceptuel d'une base de données, à l'aide du LDD défini dans [CODA 78].

Le texte de ce schéma conceptuel est ensuite codé par le SGBD, dans un processus connu comme l'*initialisation de la base*, afin qu'il puisse le manipuler de manière efficace lors de l'exécution des expressions du *langage de manipulation de donnés* (LMD) qui doit être utilisé, par les concepteurs ou par les programmeurs d'applications, pour "peupler" la base, c'est-à-dire pour créer, supprimer ou mettre à jour les enregistrements des types définis dans le schéma conceptuel.

Dans ces SGBD, certaines modifications du schéma conceptuel (*e.g.*, la suppression ou la modification des éléments des types d'enregistrements définis auparavant) requièrent parfois la réinitialisation et le repeuplement

de la base, ainsi que la recompilation des programmes d'application qui manipulent les enregistrements modifiés.

Si l'on suppose que chaque objet peut être représenté par un seul enregistrement, et que les éléments des enregistrements correspondent aux propriétés structurelles des objets, alors on peut dire que l'objectif de définir un schéma conceptuel est de décrire le type d'objets qui seront représentés dans la base de données. Définir un schéma conceptuel implique donc une connaissance exacte des propriétés de ces objets, et cette connaissance est acquise en analysant au préalable les objets en question.

Etant donné que concevoir consiste à définir les propriétés d'un objet ou de plusieurs objets qui n'existent pas encore [Prei 83], et que les connaissances acquises pendant le processus de conception peuvent servir à une meilleure connaissance des objets à concevoir et remettre en cause la définition de ces objets [Davi 81], les SGBD utilisés dans les environnements de CAO doivent permettre la modification dynamique du schéma conceptuel, c'est-à-dire sans avoir besoin de réinitialiser et de repeupler la base de données, ou de recompiler les programmes d'application.

Cette fonctionnalité est indispensable pour intégrer les différents outils de CAO dans un seul système [East80, JuLL86, StaA86], et son absence est, d'après Staley et Anderson, la limitation des SGBD la plus souvent mentionnée dans les environnements de conception.

1.6. Maintien de l'identité des objets

Pour identifier de manière non-ambiguë chaque objet dans les bases de données des SGBD conventionnels, il est nécessaire d'inclure dans leur schéma conceptuel la définition d'une ou de plusieurs propriétés qui peuvent constituer une «clé» [Codd 70], c'est-à-dire des propriétés dont l'ensemble de valeurs peut être utilisé pour distinguer les objets de chaque type d'objets. Ainsi, les OCC peuvent être identifiés, par exemple, par leur nom, tandis que les versions des OCC peuvent être identifiées par leur numéro plus le nom de l'OCC associé[1].

Parmi les multiples problèmes [Kent79, KhoC86] qui résultent de cette manière d'identifier les objets se trouve le fait que les propriétés composant la clé doivent toujours être les mêmes (*e.g.*, il n'est possible de supprimer du schéma conceptuel aucune des propriétés constituant une clé), et que leurs

[1]La possibilité d'inclure dans un schéma conceptuel certaines propriétés qui peuvent être utilisées comme des propriétés clés n'implique, aucunement, que tous les SGBD soient capables de vérifier que ces propriétés n'ont pas les mêmes valeurs pour deux objets du même type. Ceci est une contrainte qui doit parfois être mise en vigueur par les programmes d'application.

valeurs ne peuvent jamais changer (*e.g.*, on ne peut pas changer le nom d'un OCC).

Etant donné qu'au cours d'un processus de CAO les concepteurs peuvent avoir besoin de modifier le schéma conceptuel de la base de données, pour ajouter, supprimer et/ou modifier des objets et des propriétés définies auparavant (y comprises les propriétés clés!), il est nécessaire que le SGBD présente des concepts ou des techniques permettant de maintenir l'identité des objets indépendamment des changements que leur structure peut subir.

1.7. Représentation d'informations sémantiques

Chacune des associations qui peuvent exister entre les différents objets d'un système de CAO, et qui doivent être représentées dans la base de données, est caractérisée par un certain nombre de contraintes sur la structure des objets associés et sur la manière dont ils peuvent être manipulés.

Par exemple, les descriptions d'un OCC ne peuvent pas être, en même temps, des descriptions d'un autre OCC. Par contre, le concepteur d'un OCC peut être en même temps le concepteur d'autres OCC. D'autre part, les concepteurs des OCC peuvent en général travailler dans des projets de conception différents, et les descriptions des OCC doivent en général appartenir à l'un de types de description connus par le système de CAO (*e.g.*, description logique, description électrique, etc., dans la CAO de systèmes VLSI).

Ces contraintes, appelées **contraintes d'intégrité**, sont des règles qui doivent être respectées pour maintenir la base de données cohérente du point de vue sémantique, c'est-à-dire conforme à la réalité qu'elle représente. Cependant, leur mise en vigueur n'est en général pas une action réalisée par les SGBD conventionnels, mais par les concepteurs et par les programmes d'application.

En effet, dans ces SGBD, toute association entre objets peut être représentée par les éléments constituant les enregistrements des fichiers, ou les n-uplets des relations, mais il n'y a que très peu de règles (*e.g.*, des constituants clés et des valeurs non-nulles) qui peuvent être représentées dans la base de données en vue de définir la signification de ces associations.

Cela implique que, pour les SGBD conventionnels, il n'y a pas vraiment de différence entre les associations représentées par les éléments de ces enregistrements ou de ces n-uplets. D'après les connaissances contenues dans la base de données, la sémantique des associations entre les objets en cours de conception et leurs descriptions est la même que, par exemple,

celle des associations entre ces objets et leurs concepteurs ou entre ces objets et leurs composants.

Ceci étant, pour maintenir l'intégrité sémantique de la base de données, il est absolument indispensable que les programmeurs d'applications et les concepteurs connaissent les multiples contraintes qui caractérisent chacune des associations représentées par chaque élément des enregistrements de chaque fichier qu'ils doivent manipuler.

Cette tâche, évidement très difficile à réaliser, constitue l'un des problèmes les plus ardus [Buch84 et MerA87] dans les environnements de conception. En outre, cette façon de contrôler l'intégrité sémantique de la base de donnés est

a) inefficace, car elle complique la conception, la spécification et la modification de la base de données, le développement et la maintenance des programmes d'application, et l'analyse de l'intégrité sémantique,
b) dangereuse parce qu'elle peut amener au développement de programmes définissant de manière antagoniste la sémantique de l'application, et parce que rien n'empêche les concepteurs d'introduire des erreurs dans la base.

Ainsi, l'un des principaux besoins dans les environnements de conception est que le SGBD soit capable de mettre en vigueur un nombre plus grand de contraintes d'intégrité, afin que les programmeurs d'applications et les concepteurs puissent concentrer leur attention respectivement sur le développement d'algorithmes et sur les processus de conception, au lieu d'occuper la plupart de leur temps dans des actions concernant le contrôle de l'intégrité de la base de données.

Mais cela n'est évidement possible que si le SGBD connaît les contraintes qui doivent être mises en vigueur. De plus, ces connaissances doivent pouvoir être enrichies, raffinées et, en général, modifiées au cours des processus de conception, car elles peuvent dériver de la méthodologie et de la technologie utilisées, des standards adoptés (internationaux, nationaux, et/ou particuliers à un projet), et d'autres éléments qui peuvent toujours évoluer.

Par ailleurs, ces connaissances ne peuvent pas toutes être définies de manière déclarative. La vérification[1] de certaines contraintes d'intégrité peut requérir des analyses très complexes (aussi bien qualitatives que

[1]Dans cette thèse, nous utilisons indistinctement les termes vérification et validation. Théoriquement, la vérification requiert des preuves formelles, tandis que la validation implique des preuves par l'évidence. Pourtant, en CAO, la plupart des outils dits de vérification (*e.g.*, les simulateurs) valident en réalité les objets conçus.

quantitatives) qui ne peuvent être représentées que par des procédures qui utilisent, en plus, des connaissances particulières au domaine d'application.

Ce type d'analyses et de connaissances est indispensable pour vérifier, par exemple, qu'il n'y a pas de conflits spatiaux dans un arrangement 3D, que les différentes implémentations d'une même description sont «équivalentes», et que les interfaces de deux modules électroniques que l'on veut connecter sont «compatibles».

En résumé, les SGBD des environnements de conception doivent permettre la représentation de connaissances qui puissent les rendre capables de valider une plus grande variété de contraintes d'intégrité.

1.8. Représentation des propriétés opératoires des objets

Le maintien de l'intégrité sémantique des bases de données n'implique pas uniquement des processus agissant de manière analytique pour valider des contraintes d'intégrité. Au contraire, la plupart des opérations qui modifient les objets requièrent, en plus, des processus agissant de manière constructive (*e.g.*, créer, supprimer et modifier d'autres objets) pour assurer cette intégrité.

Par exemple, la destruction de n'importe quel OCC implique la destruction de toutes ses versions, mais aucune destruction ne doit se faire si l'objet en question est un composant d'un autre objet. D'autre part, la compilation d'une description quelconque d'un OCC implique, dans certaines applications, la création d'un autre objet connu comme le "modèle simulable" de cet OCC, mais cette compilation ne peut avoir lieu que si la description en question a subi, au préalable, un processus de vérification syntaxique et sémantique.

Cela implique que les SGBD des environnements de conception doivent permettre non seulement la représentation de connaissances concernant les propriétés structurelles des objets des applications, mais aussi des connaissances concernant leurs **propriétés opératoires**, c'est-à-dire la manière dont ils peuvent être manipulés, les contraintes qui doivent être respectées lors de ces manipulations, et les effets de ces manipulations sur d'autres objets de la base de données.

1.9. Traitement flexible d'erreurs et de cas exceptionnels

En CAO, les OCC n'atteignent pas un état cohérent tout d'un coup. La nature même des processus de conception implique de traiter avec le nouveau et l'inconnu [L&al 85], et les concepteurs peuvent parfois avoir

besoin d'introduire volontairement des incohérences pour tester une idée furtive.

En conséquence, les SGBD des environnements de conception doivent tolérer et gérer des incohérences temporelles, à durée variable, dans les bases de données. Il est même souhaitable que la vérification de certaines contraintes d'intégrité ne soit pas réalisée de manière automatique, après chaque mise à jour, mais à la demande et sous le contrôle des concepteurs.

Cette vérification "différée" de contraintes d'intégrité n'est réalisée par les SGBD conventionnels que dans le cas d'opérations constituant une transaction, c'est-à-dire une collection d'opérations qui doit être considérée par le SGBD comme une «unité atomique d'exécution» [DelA 82]. De plus, si à la fin de la transaction le SGBD constate qu'une contrainte d'intégrité n'a pas été respectée, il procède à la destruction des travaux effectués par chacune des opérations de la transaction.

Etant donné que la CAO implique de longs processus, une telle approche du traitement d'erreurs et de cas exceptionnels peut occasionner la perte de plusieurs heures, jours, voire semaines de travail! Le mécanisme du contrôle de l'intégrité des SGBD des environnements de conception doit donc permettre:

a) la définition de contraintes, aussi bien au moyen de prédicats que d'algorithmes, formulés dans un langage de haut niveau, qui rendent un résultat booléen et incluent des accès à la base de données,
b) la définition des réactions ("handlers") aux violations de ces contraintes (*e.g.*, envoi d'un message ou déclenchement d'autres actions),
c) l'ajout, la suppression et la modification de contraintes d'intégrité et des réactions aux violations, tout le long du processus de conception,
d) l'activation et l'inactivation du mécanisme de vérification de contraintes d'intégrité,
e) la désignation des contraintes qui doivent être vérifiées, et des réactions en cas de violations.

1.10. Discussion

Les différentes fonctionnalités que nous avons analysées dans les sections précédentes ne sont évidemment pas nécessaires uniquement dans les environnements de conception. Le besoin de manipuler des objets composites et des objets ayant plusieurs versions est aussi une caractéristique des applications de bureautique et de génie logiciel, entre

autres. De la même manière, l'organisation d'objets dans des hiérarchies en fonction de leurs propriétés structurelles et opératoires est cruciale dans le domaine de la biologie et de l'intelligence artificielle.

D'autre part, la possibilité de modifier de manière dynamique le schéma conceptuel de la base de données, et de maintenir l'identité des objets indépendamment des changements que leur structure peut subir, est certainement liée à une exigence plus générale connue comme "indépendance logique et physique des données", et définie comme l'immunité des programmes d'application aux modifications du schéma conceptuel ou de son implémentation dans l'ordinateur.

Enfin, d'une manière ou d'autre, chacune de ces fonctionnalités à été mentionnée, individuellement, comme une fonctionnalité nécessaire dans d'autres applications. Toute de même, un fait qui semble particulier à la CAO est que ces fonctionnalités, toutes, sont nécessaires dans chacun des environnements de conception, et cela implique que les différents concepts et techniques qui peuvent être utilisés pour les intégrer dans un seul SGBD doivent constituer un contexte cohérent et robuste.

L'approche qui consiste à développer autour des SGBD conventionnels un module ou sous-système pour chacune des nouvelles fonctionnalités (*voir* figure 1.9) a essentiellement deux inconvénients. Le premier est que les concepts et les techniques utilisés pour développer chaque module peuvent être antagonistes à celles utilisés pour développer les autres sous-systèmes. De plus, même s'il n'y a pas d'antagonismes, cette approche peut rendre très difficile, voire impossible, le développement d'une interface unique avec les utilisateurs.

Le deuxième inconvénient est que ces sous-systèmes peuvent parfois être conçus comme des modules autonomes, dans ce sens que leur fonctionnement ne dépend ni du fonctionnement ni des ressources (*e.g.*, des données et des algorithmes) utilisés par d'autres sous-systèmes. Cela diminue le nombre de problèmes créés par l'existence de concepts ou de techniques antagonistes, mais implique, forcément, la présence redondante de ressources, ce qui occasionne d'autres problèmes qui sont autant ou plus difficiles que ceux des antagonismes (*e.g.*, le contrôle de l'intégrité sémantique, l'évolution des ces modules et les performances globales du SGBD "étendu").

Une deuxième approche qui a été utilisée (*e.g.*, [Hosk 79], [SucW 79], et [Wilm 79]), pour intégrer dans les environnements de conception les fonctionnalités de gestion de données dont les concepteurs ont besoin, consiste à développer des SGBD *"ad hoc"*, c'est-à-dire des SGBD qui connaissent la sémantique d'une application ou d'un type spécifique d'applications (*e.g.*, CAO de circuits intégrés).

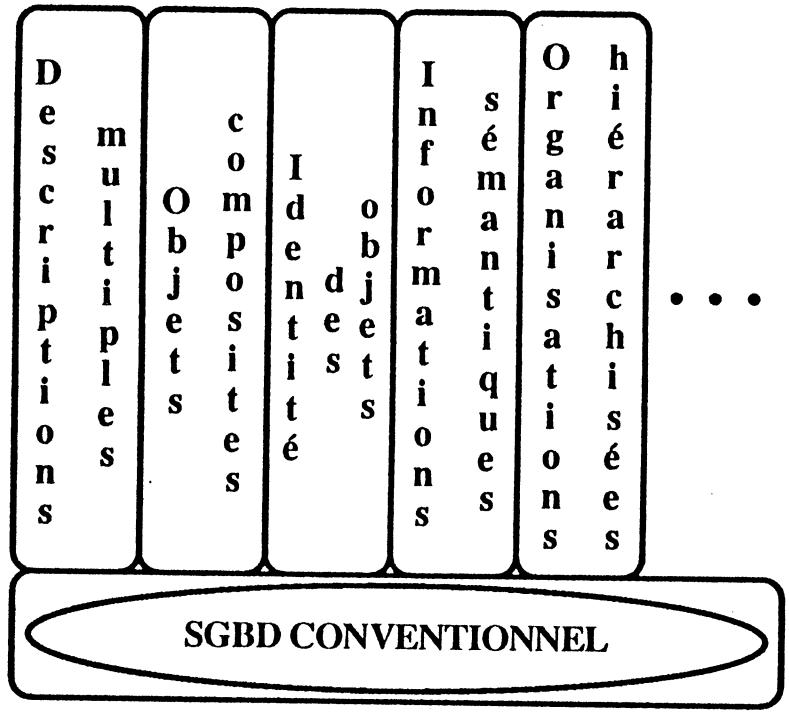


FIGURE 1.9

Le plus grand avantage d'utiliser cette approche est que les SGBD résultants ont des performances excellentes, car ils n'incluent que les concepts et les techniques strictement nécessaires pour *a*) stocker dans la base un nombre limité et invariable de types d'objet, *b*) réaliser les opérations particulières à ces objets, et prédéfinies par le système, et *c*) donner des réponses à un ensemble prédéterminé de requêtes. Rien d'autre.

Le principal inconvénient de ces SGBD est que leurs connaissances ne peuvent pas être modifiées. A cet égard, ils sont encore plus rigides que les SGBD conventionnels, car ceux-ci ont au moins la notion de "schéma conceptuel", qui permet d'enrichir un peu leurs connaissances de base. Ainsi, toute évolution technologique rend les SGBD *ad hoc* immédiatement obsolètes donc inutiles, aucun nouvel outil ne peut être intégré à l'environnement de conception, si ses besoins de gestion de données ne sont pas un sous-ensemble des besoins des autres outils, etc.

Une troisième approche consiste à développer de nouveaux SGBD avec des concepts plus riches, plus puissants, plus expressifs que ceux des "modèles de données" des SGBD conventionnels (*e.g.*, les modèles hiérarchisé, réseau et relationnel), afin de pouvoir représenter dans les bases de données une plus grande variété de connaissances et, de ce fait, pouvoir utiliser les nouveaux SGBD dans des environnements divers.

Cette approche a été suivie dans tous les travaux de recherche qui sont à l'origine des infrastructures de modélisation connues comme "modèles sémantiques de données" (*e.g.*, le modèle relationnel binaire de J. Abrial [Abri 74], le modèle Entité-Association [Chen 76], RM/T [Codd 79], et SHM [SmiS 77]). Malheureusement, ces travaux n'ont pas pris en compte les besoins de gestion de données particuliers aux environnements de conception et, en conséquence, d'autres recherches ont dû être menées soit pour étendre les modèles sémantiques existants, soit pour développer des nouvelles infrastructures (*e.g.*, [BatB 84], [Katz 82], [LoPV 83] et [Rieu 85]).

La conception de ces infrastructures a été notamment guidée par l'intérêt de développer des structures de données permettant de représenter dans les bases de données une plus grande variété de connaissances, afin de pouvoir utiliser ces SGBD dans des environnements divers, et dans ce sens notre travail de recherche est similaire à tous ceux qui ont été réalisés avec la troisième approche.

Néanmoins, une différence fondamentale entre ces travaux et le nôtre est que nous avons porté un intérêt particulier au développement des concepts et des techniques qui peuvent rendre le SGBD capable de s'adapter à des environnements différents et aux évolutions de ces environnements. Cet intérêt nous a amené au développement d'une infrastructure de représentation de connaissances que nous présentons dans la suite de cette thèse.

CHAPITRE 2

Concepts fondamentaux

2. Concepts fondamentaux

2.1. Introduction

Notre décision de développer une infrastructure de représentation de connaissances pour la gestion de données en CAO à été influencée par les résultats des recherches sur les modèles sémantiques, réalisées dans le domaine des bases de données, mais aussi par ceux des recherches sur la représentation de connaissances, réalisées dans le domaine de l'intelligence artificielle, et des recherches sur la spécification et le développement de programmes, réalisées dans le domaine des langages de programmation.

Cette infrastructure contient ainsi plusieurs concepts et techniques empruntés à ces domaines, qui dans la plupart des cas ont été modifiés ou étendus en vue de pouvoir les intégrer dans un seul contexte cohérent et robuste, capable de satisfaire aux besoins relatifs à la gestion de données dans les environnements de CAO.

Notre objectif dans ce chapitre est de définir et d'analyser les concepts fondamentaux de notre infrastructure de représentation de connaissances, ainsi que de mettre en relief leurs principales similitudes et différences avec des concepts analogues utilisés dans ces domaines de recherche.

Par souci de simplicité, tout au long du chapitre ces concepts seront illustrés à l'aide d'exemples graphiques, au lieu d'introduire des structures de données ou des opérateurs qui sont particulières à une implémentation spécifique. Des concepts additionnels, et tous les opérateurs intrinsèques à notre infrastructure de représentation de connaissances, seront présentés dans les chapitres suivants.

2.2. Approche centrée-objet

Le postulat fondamental des infrastructures de modélisation centrées-objet est que les éléments de tout univers conceptuel peuvent être décrits en termes d'**objets** dont les propriétés définissent leurs associations avec d'autres objets.

Dans ces infrastructures, chaque objet est modélisé comme une seule unité, et non comme des multiples enregistrements ou n-uplets de plusieurs fichiers ou relations. De plus, les propriétés des objets peuvent être non seulement des valeurs simples tels que les nombres et des chaînes de caractères, mais d'autres objets de n'importe quelle taille ou complexité.

Par ailleurs, ces propriétés peuvent être partagées ou héritées par d'autres objets.

Dans notre infrastructure de représentation de connaissances tout objet peut être représenté dans une **base d'objets** gérée par un système que nous appelerons **OBMS**. Ce système permet l'utilisation de tous les concepts et techniques de modélisation constituant l'infrastructure et, pour cette raison, le terme OBMS sera employé ici pour dénoter aussi bien le système que l'infrastructure.

Certains objets sont appelés **élémentaires**, car ils sont des unités indépendantes dont la sémantique est supposée connue. Ils n'ont donc pas besoin d'être définis en termes d'autres objets. Le numéro 4096, la chaîne de caractères "xyz 123/4A", et les expressions symboliques de Lisp, sont des exemples d'objets élémentaires.

D'autres objets sont appelés **non-élémentaires** parce que leur sémantique doit être définie en termes d'autres objets élémentaires ou non. Autrement dit, leur signification dépend de leurs associations avec d'autres objets.

L'objet "ordinateur-1", par exemple, peut être défini en termes de ses associations avec un certain processeur, un système d'exploitation, l'ingénieur qui l'a conçu, et le projet de conception auquel il appartient. En même temps, ce projet peut être un objet non-élémentaire défini en termes de son budget, son responsable technique et le nombre d'employés qui y travaillent. Un "réseau sémantique" [Quil 68] représentant ces associations est illustré dans la figure 2.1.

En accord avec l'approche centrée-objet, dans OBMS les associations entre objets peuvent être définies comme des **propriétés structurelles** des objets non-élémentaires. Chacune de ces propriétés a un nom et une valeur. La valeur identifie l'objet (ou les objets) associé(s) à celui ayant la propriété en question, et le nom est un identificateur symbolique utilisé pour distinguer cette propriété parmi les autre propriétés du même objet.

La figure 2.2 contient une représentation centrée-objet du réseau sémantique de la figure 2.1, inspirée de la notion de schéma (*en anglais:* frame) de M. Minsky [Mins 75].

Nous appelons **structure d'un objet** l'ensemble des noms de ses propriétés structurelles. Celles-ci sont aussi appelées "propriétés statiques" parce que l'ensemble de leurs valeurs décrit, à chaque instant, l'**état d'un objet**.

Le **comportement d'un objet**, c'est-à-dire la manière dont cet objet peut changer d'état, est déterminé par ses **propriétés opératoires** ou "dynamiques". La définition de ces propriétés sera examinée au chapitre 3.

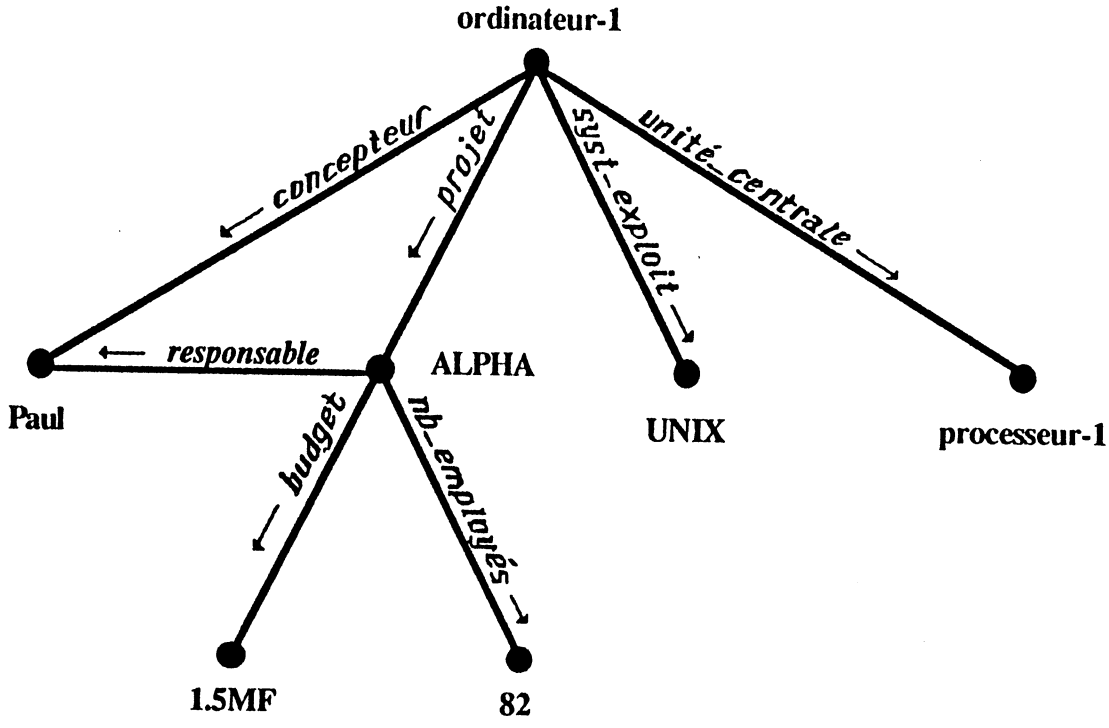


FIGURE 2.1
Description des objets ordinateur-1 et ALPHA, en termes de leurs associations avec d'autres objets.

Objet ordinateur-1

concepteur	: Paul	
projet	: ALPHA	
syst-exploit	: UNIX	
unité-centrale	: processeur-1	

Objet ALPHA

responsable	: Paul	;le concepteur d'ordinateur-1.
budget	: 1.5 MF	
nb-employés	: 82	

FIGURE 2.2
Une description centrée-objet du réseau sémantique illustré dans la figure 2.1.

2.3. Identification des objets

Depuis la publication en 1974 du travail de J. Abrial, concernant la sémantique des données [Abri 74], plusieurs chercheurs (*e.g.*, [Codd 79], [Kent 79] et [KhoC 86]) ont argumenté les multiples avantages d'utiliser des identificateurs, générés par le système, pour distinguer les objets indépendamment de leur structure, leur état ou leur localisation. Ces identificateurs sont appelés **surrogates**, et leurs caractéristiques les plus importantes sont les suivantes:

a) Ils sont créés par le système lorsque l'utilisateur lui demande la création d'un objet.
b) Un surrogate est détruit par le système lorsque l'utilisateur demande la destruction de l'objet identifié par ce surrogate.
c) Les surrogates permettent d'identifier de manière non-ambiguë chacun des objets créés par les utilisateurs.
d) Toutes les propriétés structurelles et opératoires des surrogates sont connues par le système.
e) La seule différence entre deux surrogates quelconques est leur valeur. Celles-ci sont absolument indépendantes de toute propriété des objets qu'ils identifient, et n'impliquent au niveau conceptuel aucune relation d'ordre entre ces objets.
f) Les utilisateurs ne peuvent pas modifier la valeur, la structure, le comportement, la représentation externe, etc., des surrogates.
g) L'emploi de surrogates n'est pas antagoniste à la définition d'attributs "clé" (*Cf.* section 3 au chapitre suivant), et peut être transparent pour la plupart des utilisateurs du système, au-delà des niveaux physique et conceptuel.

Par souci d'uniformité, dans OBMS ces surrógates sont eux aussi considérés comme des objets, mais élémentaires, et ils ne sont utilisés que pour identifier des objets non-élémentaires. Sans cette dernière restriction, on aurait eu besoin d'avoir des surrogates pour identifier des surrogates. Les objets élémentaires sont identifiés par leur valeur.

L'emploi de surrogates a été introduit dans notre infrastructure de représentation de connaissances pour satisfaire au besoin mentionné dans la section 6 du chapitre 1, à savoir, la possibilité de maintenir l'identité de chaque objet indépendamment des modifications que sa structure, son état ou sa localisation peuvent subir.

2.4. Mécanismes d'abstraction

La définition de modèles conceptuels d'applications complexes, telles que celles de la CAO, est en général caractérisée par une certaine hésitation en

ce qui concerne les propriétés et les contraintes que l'on doit représenter dans la base, afin de pouvoir mettre en œuvre les processus souhaités (c'est-à-dire des transactions et des requêtes).

Comme tout processus de conception empirique et heuristique, cette modélisation a un caractère provisoire et itératif (plusieurs alternatives de représentation sont en général examinées), et la "qualité" du modèle résultant dépend de l'expérience et des connaissances de ses concepteurs, bien entendu, mais aussi de la quantité de détails qui doivent être considérés à chaque instant, lorsque l'on modélise des systèmes complexes.

L'outil intellectuel le plus puissant pour faire face à la complexité est, sans doute, l'abstraction. Tout processus d'abstraction consiste à ne pas considérer (temporairement au moins) certains détails du problème traité, en vue de mettre l'accent sur ceux qui sont les plus significatifs. L'abstraction offre des mécanismes organisateurs utilisés dans la plupart des méthodologies de modélisation proposées, par exemple, dans les domaines des bases de données, de l'intelligence artificielle, et du génie logiciel (*e.g.*, [BoMW 84], [Gutt 77], [Parn 72], [SmiS 77] et [Wirt 71]). Pour notre infrastructure de modélisation, nous avons adopté et étendu deux des mécanismes d'abstraction qui ont reçu la plus grande attention dans ces domaines: l'agrégation et la généralisation.

2.4.1. Agrégation

Le mécanisme d'agrégation permet de considérer une collection d'objets **agrégés** comme un seul objet **agrégat** de plus haut niveau (c'est-à-dire plus abstrait), sans tenir compte de leurs particularités. L'ordinateur-1 et le projet ALPHA illustrés dans la figure 2.2 sont des exemples d'objets agrégats.

L'agrégation représente les associations **composant-de** entre objets, et constitue des hiérarchies, comme celle de la figure 2.1, avec les objets agrégats placés au-dessus des objets agrégés. Ce mécanisme est donc adéquat pour satisfaire aux exigences des concepteurs mentionnées dans la section 2 du chapitre 1.

Dans OBMS, les propriétés qui représentent les associations composant-de entre objets sont appelées **attributs**. Vues comme des relations binaires (au sens mathématique du terme), les associations représentées par les attributs des objets sont transitives, mais les hiérarchies formées par leur fermeture transitive ne sont pas forcément des arbres (*voir* figure 2.3). Ceci permet de modéliser des objets ayant des informations en commun, comme des objets qui partagent un composant contenant ces informations communes.

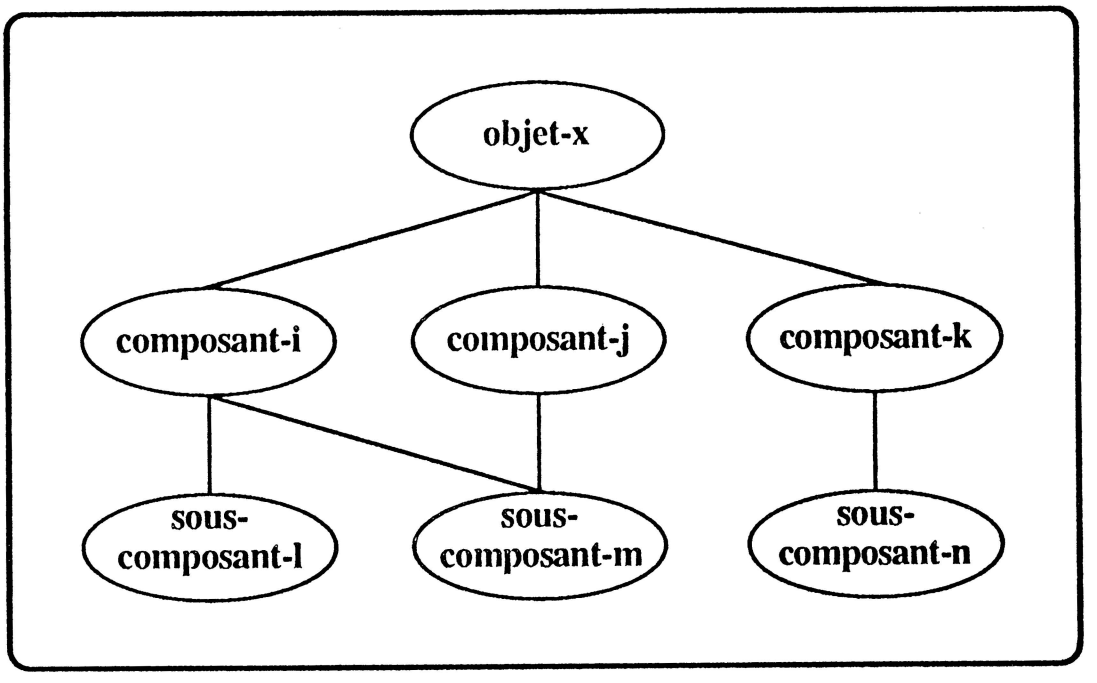


FIGURE 2.3
Exemple d'une hiérarchie formée par agrégation d'objets.

L'un des avantages de ce partage est qu'il réduit la redondance (donc le volume) d'informations dans la base, c'est-à-dire l'une des sources d'incohérence lors des opérations de mise à jour. D'autres propriétés des associations composant-de seront décrites dans les sections suivantes.

2.4.2. Généralisation

Le mécanisme de généralisation permet de regrouper une collection d'objets dans une seule unité plus abstraite en accord avec leurs similitudes et en ignorant leurs différences. Il peut être utilisé de deux manières distinctes dans la plupart des langages de programmation centrés-objet (*e.g.*, Smalltalk-80 [GolR 83] et Loops [SteB 86]), dans les langages de représentation de connaissances utilisés en intelligence artificielle et basés sur la notion de schémas (*e.g.*, KRL [BobW 77] et Shirka [Rech 85]), et dans certains langages de définition de données associés aux modèles sémantiques de bases de données (*e.g.*, Bêta [Brod 82] et Lambda [LoPV 83]).

La première forme de généralisation permet de considérer une collection d'objets **instance** comme un seul objet **classe**, plus abstrait, qui définit de manière précise les attributs communs à ces instances, sans tenir compte de leurs différences. Par exemple, les objets ordinateur-1, ordinateur-2, ..., et ordinateur-N peuvent être généralisés pour constituer la classe Ordinateur (*voir* figure 2.4). Cette forme de généralisation est appelée "généralisation

par adhésion" (*en anglais:* generalization by membership) dans [Codd 79], et représente les associations **instance-de** entre objets.

L'une des propriétés les plus importantes de ce type d'associations est que toute instance doit posséder, au moins[1], les attributs définis par leurs classes. Cette règle est une contrainte "universelle" d'intégrité que doivent vérifier toutes les instances de chaque classe, indépendamment du domaine d'application, et correspond au **principe d'induction de propriétés** défini dans [MyBW 80].

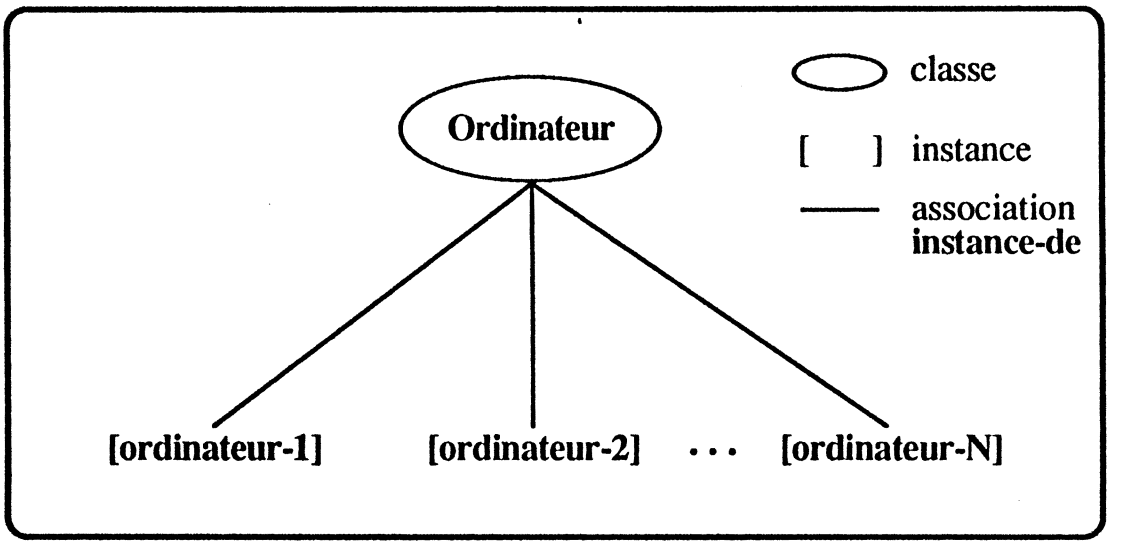


FIGURE 2.4

La deuxième forme de généralisation permet de considérer une collection de classes, dites **sous-classes** ou spécialisations, comme une seule **super-classe**, plus abstraite, qui décrit tous les attributs partagés par ses sous-classes. Cette forme de généralisation (appelée "généralisation par inclusion" dans [Codd 79]) représente les associations **spécialisation-de** entre objets, et constitue des hiérarchies (taxinomies) avec les classes les plus abstraites placées au-dessus de leurs spécialisations.

Les classes représentant des ordinateurs, des additionneurs, et des unités arithmétiques et logiques, peuvent être généralisées pour constituer, par exemple, la classe représentant tous les modules électroniques (Module, dans la figure 2.5). De la même façon, la classe Ordinateur peut être une généralisation des classes représentant des micro-ordinateurs, des mini-ordinateurs et des ordinateurs universels.

1Certains systèmes de modélisation (y compris OBMS) permettent d'associer une instance à plusieurs classes. Cette classification multiple est l'une des raisons par lesquelles une instance peut avoir des attributs qui ne sont pas définis dans une classe donnée.

L'une des conséquences les plus importantes de cette organisation hiérarchique est que les sous-classes doivent avoir tous les attributs définis par leurs super-classes, en plus de leurs éventuels attributs propres, et que (comme suite logique) toutes les instances de chaque classe sont considérées aussi comme des instances de ses super-classes[1].

En accord avec la figure 2.5, par exemple, tous les ordinateurs sont considérés comme des modules électroniques, car la classe Ordinateur doit avoir tous les attributs de la classe Module (*e.g.*, le nom du projet auquel ils appartiennent et l'ingénieur qui les ont conçu), en plus de ses attributs propres (*e.g.*, unité centrale et système d'exploitation). Dans notre infrastructure de représentation de connaissances, les attributs qui sont propres à un objet sont appelés **caractéristiques**.

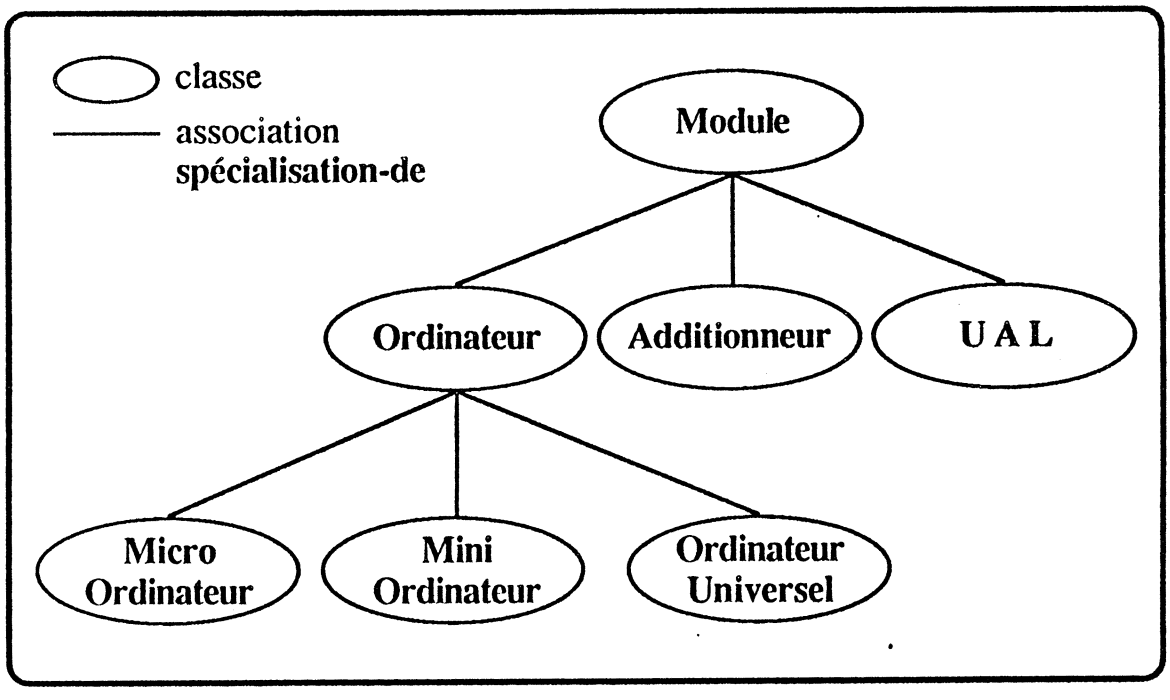


FIGURE 2.5

Cette corrélation entre les propriétés des super-classes et celles de leurs sous-classes est régie par un principe qui est connu sous le nom d'**héritage de propriétés**, et qui représente (dans les associations entre les super-classes et leurs sous-classes) une contrainte universelle d'intégrité analogue à celle du principe d'induction de propriétés intrinsèque aux associations entre les classes et leurs instances.

1Dans notre infrastructure de représentation de connaissances, toute classe peut avoir une ou plusieurs super-classes (*voir* figure 2.6), et les attributs additionnels ne sont pas obligatoires.

En fait, on peut constater que dans les associations spécialisation-de il y a aussi bien de l'héritage d'attributs des super-classes vers les sous-classes, que de l'héritage d'instances des sous-classes vers leurs super-classes. Dans [MyBW 80], les règles d'intégrité régissant ces deux cas d'héritage sont appelées, respectivement, la "contrainte structurelle" et la "contrainte d'extension" des associations spécialisation-de (*en anglais:* the intensional and extensional IS-A constraints).

L'induction et l'héritage de propriétés peuvent être utilisés dans les processus de modélisation conceptuelle non seulement pour représenter des contraintes d'intégrité, mais aussi pour réduire la complexité et le volume des spécifications de propriétés des objets.

Par exemple, la description d'une sous-classe doit contenir la définition de ses caractéristiques (c'est-à-dire ses attribut propres), mais il n'y a pas besoin de dupliquer la description de sa super-classe pour définir ses autres attributs. De la même manière, les instances de la sous-classe n'ont pas besoin d'être redéfinies au niveau de la super-classe.

Outre l'analogie entre les principes d'induction et d'héritage de propriétés, les associations instance-de et spécialisation-de ont d'autres similitudes qui concernent non seulement les propriétés structurelles des objets associés, mais aussi le comportement, c'est-à-dire les propriétés opératoires, de ces objets.

Par exemple, dans les systèmes de modélisation où les instances ne peuvent être associées qu'à une seule classe, et les classes ne peuvent avoir qu'une seule super-classe, une règle commune est que la destruction d'une classe implique toujours la destruction de ses instances et de ses sous-classes.

De manière similaire, lorsque les instances peuvent être associées à plusieurs classes et les classes peuvent avoir plusieurs super-classes, la destruction d'une classe n'implique en général que la destruction de ses instances qui ne sont pas associées à d'autres classes, et la destruction de ses sous-classes qui n'ont pas d'autres super-classes.

L'ensemble de toutes les règles ou contraintes d'intégrité qui doivent être respectées par les objets, lorsqu'ils sont associés en utilisant les mécanismes d'agrégation et de généralisation, définit la sémantique des associations composant-de, instance-de et spécialisation-de. Les exemples précédents montrent pourtant que ces règles ne sont pas toujours les mêmes dans les différentes systèmes de modélisation existants. En conséquence, nous allons préciser quelle est, dans notre infrastructure de représentation de connaissances, la sémantique des associations décrites dans ce chapitre, en spécifiant leurs principales contraintes d'intégrité.

Contraintes sur les associations composant-de

aa1)	L'association composant-de n'est définie qu'entre deux objets instance ou entre deux objets classe.
aa2)	Si A et B sont des objets classe, A est un composant de B lorsque celui-ci a des instances ayant un ou plusieurs attributs dont les valeurs sont des instances de A. (*Règle provisoire.*)
aa3)	Si un objet X est composant d'un objet Y, alors Y est un agrégat de X.
aa4)	Tout objet peut avoir un nombre quelconque de composants et être composant d'un nombre quelconque d'agrégats.
aa5)	Si un objet X est composant d'un objet Y, alors tous les composants de X sont aussi des composants de Y (transitivité).
aa6)	Tout objet est composant et agrégat de lui-même (réflexivité).
aa7)	Toute instance hérite des caractéristiques et, en général, de tous les attributs de ses composantes. Ces caractéristiques et ces attributs sont appelés **caractéristiques et attributs internes.**
aa8)	Aucun objet ne peut être détruit s'il est composant d'un autre objet.

Contraintes sur les associations instance-de

ci1)	L'association "x est instance-de A" est définie uniquement si x est un objet instance et A est un objet classe.
ci2)	Si x est une instance de A, alors A est une classe de x.
ci3)	Toute instance peut avoir un nombre quelconque de classes, et toute classe peut avoir un nombre quelconque d'instances.
ci4)	Si x est instance d'une classe A et celle-ci est sous-classe de B, alors x est aussi une instance de B. Autrement dit, les instances de toute classe sont aussi des instances de ses super-classes.
ci5)	Chaque instance doit avoir tous les attributs définis par ses classes et, en plus, elle peut avoir des caractéristiques qui ne sont pas définies par ces classes. (*Règle provisoire.*)
ci6)	La destruction d'une classe A entraîne la destruction préalable de toutes ses instances qui ne sont pas associées à une autre classe. Ainsi, si en accord avec la règle aa8 ces instances ne peuvent pas être détruites, alors la classe A ne peut pas être détruite non plus. (Voir aussi les règles cc8 et cc9.)

Contraintes sur les associations spécialisation-de

cc1)	L'association binaire "spécialisation-de" est aussi appelée sous-classe-de et n'est définie qu'entre objets classe.
cc2)	Si un objet A est une sous-classe d'un objet B, alors B est une super-classe de A.
cc3)	Toute classe peut avoir un nombre quelconque de sous-classes et de super-classes.
cc4)	Les sous-classes de chaque classe sont aussi des sous-classes de toutes ses super-classes (transitivité).
cc5)	Aucune classe A ne peut être spécialisation d'une classe B si celle-ci est une spécialisation de A (asymétrie).
cc6)	Toute classe est sous-classe et super-classe d'elle-même (réflexivité).
cc7)	Chaque classe hérite de tous les attributs de ses super-classes et, par conséquent, elle ne peut ni redéfinir ni spécialiser ces attributs; pourtant, elle peut avoir d'autres caractéristiques additionnelles. (Voir aussi la règle ci4.)
cc8)	La destruction d'une classe A entraîne la destruction préalable de toutes ses sous-classes qui n'ont pas une autre super-classe. (Voir aussi la règle ci6.)
cc9)	Avant d'être détruite, toute classe lègue ses super-classes et ses caractéristiques, à ses sous-classes qui ont d'autres super-classes et qui, par conséquent, ne peuvent pas être détruites. (*Règle provisoire.*)

Cette spécification ne contient pas toutes les contraintes universelles intrinsèques à ces associations, car la description de certaines règles requiert l'emploi de concepts qui seront définis dans les chapitres suivants. Par ailleurs, les règles qui ont été signalées comme provisoires seront redéfinies en utilisant ces nouveaux concepts. L'annexe A contient une récapitulation des règles définies ci-dessus (incluant leurs redéfinitions), ainsi que des règles correspondant à d'autres associations qui seront analysées dans le chapitre 4.

2.5. Discussion

L'emploi de la première forme de généralisation dans les processus de modélisation conceptuelle implique la distinction entre deux **catégories d'objets** différentes: les instances et les classes. Les termes "classe" et "instance", utilisés pour dénoter les objets de ces deux catégories, se trouvent dans la plupart des infrastructures de modélisation centrée-objet développées dans les domaines des bases de données, de l'intelligence

artificielle et des langages de programmation. Toutefois, la connotation du concept de classe n'est pas toujours la même dans toutes ces infrastructures.

Dans cette section nous décrivons trois différences qui sont à notre avis les plus importantes, et qui concernent la spécification et l'héritage de propriétés, la classification et la spécialisation d'objets, et la représentation de leur comportement. Deux remarques additionnelles, relatives aux associations composant-de, sont présentées pour conclure ce chapitre.

2.5.1. La spécification et l'héritage de propriétés

Dans certaines systèmes tels que Act 1 [Lieb 81], KRL et Loops, toute classe est vue comme un archétype des objets qu'elle généralise (c'est-à-dire ses instances et ses sous-classes), et ceux-ci peuvent avoir des propriétés contredisant les spécifications contenues dans la classe.

Dans de tels systèmes il est possible de définir par exemple la classe Ordinateur en spécifiant que la propriété nombre-de-processeurs est égale à 1, et de définir ensuite l'instance ordinateur-1 en indiquant qu'il a 2 processeurs, ou la sous-classe Ordinateur_Universel en spécifiant que le nombre de processeurs est égale à 3.

Dans ce cas, la spécification et l'héritage de propriétés sont des connaissances par défaut qui sont utilisées dans certains types de raisonnement non-monotone[1], et qui peuvent être paraphrasées par des expressions du style «si la valeur de la propriété "P" n'est pas connue, alors admettre qu'elle est "v1" (c'est-à-dire la valeur par défaut)».

Dans d'autres systèmes, en particulier ceux utilisés dans le domaine des bases de données, la spécification et l'héritage de propriétés ne sont pas des connaissances par défaut, mais des connaissances concernant des règles ou contraintes d'intégrité qui doivent toujours être vérifiées pour maintenir les objets sémantiquement cohérents. Par conséquent, aucune instance ou sous-classe ne peut avoir des propriétés contredisant celles définies par de leurs (super-)classes. Toutefois, ceci n'empêche pas la possibilité de:

a) raffiner la spécification de propriétés lors de la définition de sous-classes,
b) avoir des instances ayant des propriétés additionnelles à celles définies par leurs classes, c'est-à-dire des propriétés qui n'ont pas de contraintes spécifiés au niveau de leurs classes.

[1]Raisonner de manière non-monotone implique le fait d'admettre des hypothèses qui peuvent être abandonnées à la lumière de nouvelles informations [CohF 82]. Les différentes logiques qui ont été proposées pour formaliser ce type de raisonnement sont décrites dans [Bobr 80] et dans [GenN 87].

En accord avec la remarque (*a*), certains systèmes tels que Taxis [MyBW 80] et Shirka permettent, par exemple, de spécifier que la propriété nombre-de-processeurs de la classe Ordinateur doit être un entier plus grand que zéro, et de spécifier ensuite que la même propriété dans la sous-classe Ordinateur_Universel doit être un entier plus grand que 2.

Ce raffinement de propriétés n'est pas autorisé dans notre infrastructure de représentation de connaissances, car cela pourrait produire des incohérences dans les bases d'objets gérées par OBMS, du fait que la spécification de propriétés des classes peut être réalisée de manière procédurale (*Cf.* chapitre 3), et qu'actuellement il n'est pas toujours possible de vérifier formellement (c'est-à-dire par des manipulations syntaxiques) la cohérence logique de deux procédures écrites avec des langages tels que Lisp, Pascal, C ou Fortran.

Par ailleurs, il y a des systèmes de modélisation qui permettent de représenter aussi bien des contraintes d'intégrité que des valeurs par défaut (*e.g.*, KEE [FikK 85] et Shirka). Dans ce cas, si la spécification de contraintes et de valeurs par défaut se fait de manière procédurale, il est encore impossible de vérifier la cohérence logique entre ces deux spécifications.

D'autre part, même si la spécification des valeurs par défaut est limitée à des constantes (*e.g.*, propriété-x = 12), leur cohérence logique avec les procédures qui définissent les contraintes d'intégrité ne peut pas toujours être vérifiée si ces procédures représentent des règles qui dépendent de l'état des objets de la base (*e.g.*, SI état-1 ALORS propriété-x $\leq$ 10 SINON propriété-x $\geq$ 15). Pour ces raisons, la spécification de valeurs par défaut n'est pas autorisée dans notre infrastructure de représentation de connaissances.

En outre, en accord avec la remarque (*b*), OBMS et KEE, entre autres, permettent de spécifier, par exemple, que l'ordinateur-1 a une certaine taille de mémoire ou une certaine date de création, même si ces propriétés ne constituent pas l'une des parties considérées comme "essentielles" pour représenter dans une base d'objets le concept "ordinateur".

Nous avons déjà mentionné que la définition de modèles conceptuels d'applications de CAO implique souvent des hésitations concernant les propriétés et les contraintes que l'on doit représenter dans la base d'objets, et que le déroulement des processus de conception peut rendre nécessaire la modification du schéma ou modèle conceptuel défini auparavant.

Dans ces conditions, il s'avère très utile de pouvoir spécifier des propriétés d'objets qui, à un certain moment, semblent importantes. Quelque temps

après, ces propriétés peuvent être considérées même indispensables et, à ce moment là, le système de modélisation doit permettre la spécification des contraintes d'intégrité correspondantes.

2.5.2. La classification et la spécialisation d'objets

Une deuxième différence associée au concept de classe concerne les processus de classification et de spécialisation. Nous avons déjà mentionné que certains systèmes, comme OBMS, permettent d'associer une instance à plusieurs classes tandis que d'autres ne le permettent pas. Tout de même, cette classification multiple est parfois limitée par une contrainte qui indique que toutes les classes d'une instance doivent avoir un ancêtre commun, c'est-à-dire une super-classe commune, ou une super-classe de leurs super-classes commune, etc.

Par exemple, la classe Mémoire illustrée dans la figure 2.6 est un ancêtre commun aux classes Mémoire_à_Bulles, RAM et Mémoire_Capacitive, mais aussi aux classes RAM_à_Bulles et Mémoire_Capacitive.

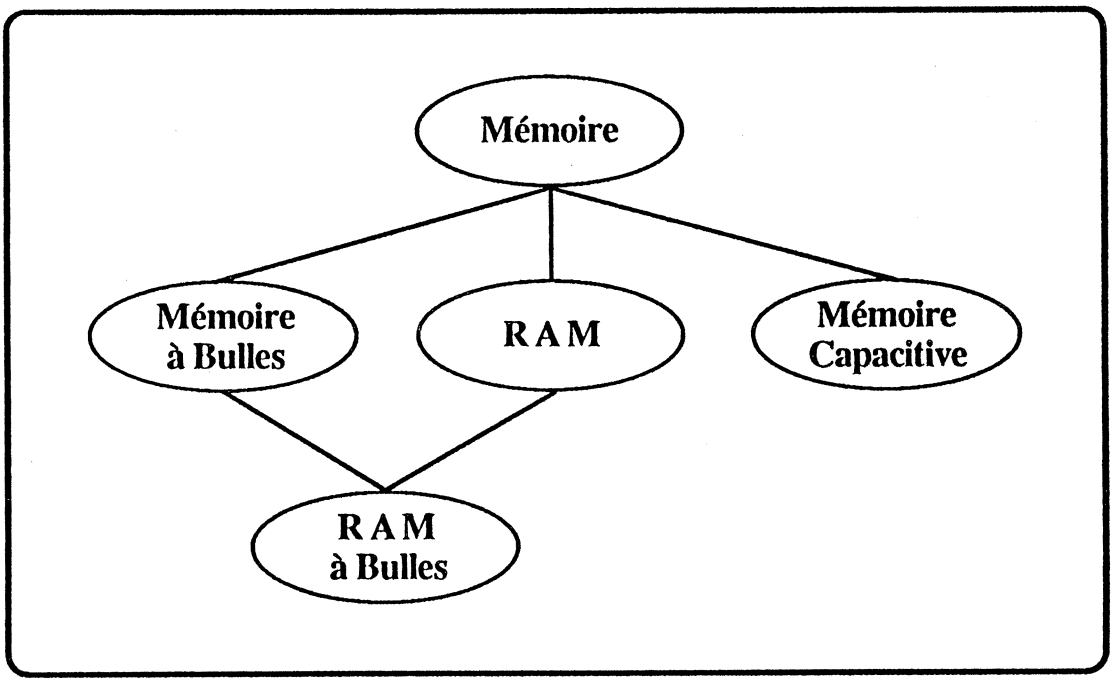


FIGURE 2.6

Cette contrainte est essentiellement imposée pour simplifier l'implémentation du système de modélisation et, par conséquent, elle n'a pas été incluse dans OBMS. De plus, dans OBMS les instances n'ont pas forcément besoin d'être associées à des objets classe. En d'autres termes, les

concepteurs peuvent créer des objets, les analyser, les modifier, les faire participer comme des composants d'autres objets, etc., avant qu'ils soient classifiés dans une classe quelconque.

Cela veut dire que les concepteurs peuvent classifier un objet, et déclencher la vérification de toutes les contraintes d'intégrité associées, après qu'ils considèrent que cet objet se trouve dans un état cohérent. Cette fonctionnalité, qui à notre connaissance n'existe pas dans d'autres systèmes, a été introduite dans OBMS pour satisfaire partiellement les besoins mentionnés dans la section 9 du chapitre 1.

En ce qui concerne le processus de spécialisation, dans certains systèmes toute sous-classe doit forcément raffiner une ou plusieurs caractéristiques de ses super-classes, ou posséder des caractéristiques additionnelles. Cette contrainte est parfois imposée pour permettre au système de classifier automatiquement les instances lors de leur création[1].

Dans OBMS cette contrainte n'est pas imposée non seulement parce que le système ne fait pas de la classification automatique, mais parce que l'absence de cette contrainte permet aux concepteurs d'avoir leur libre arbitre pour regrouper les différentes classes, même en fonction de critères "externes", c'est-à-dire en fonction de propriétés qui ne sont pas et qui n'ont pas besoin d'être représentées dans la base d'objets.

2.5.3. Le comportement des objets

Une troisième différence associée au concept de classe concerne la spécification de propriétés opératoires (c'est-à-dire le comportement) des instances. Dans tout système de modélisation, la structure des objets impose un certain nombre de contraintes sur la manière dont ces objets peuvent changer d'état.

Par exemple, dans la section précédente nous avons vu que les instances ne peuvent pas être modifiées de manière arbitraire, lorsqu'elles sont associées à des classes qui imposent des contraintes sur leurs attributs. Nous avons vu aussi que la destruction d'une classe entraîne la destruction de ses instances qui ne sont pas associées à d'autres classes.

---

[1]Dans le SGBD TIGRE [LoPV 83] et dans le système BD-CAO [Rieu 85], par exemple, si la classe Projet indique que la propriété "budget" de tout projet doit être plus grande que 500 KF, la sous-classe Projet_International peut être définie, en spécifiant que le budget doit être plus grand que 2 MF. Ceci étant, si l'utilisateur demande la création d'une instance de la classe Projet, en spécifiant que la valeur du budget est égale à 10 MF, le système peut classifier automatiquement cette instance dans la sous-classe Projet_International.

Toutefois, dans la plupart des cas ces règles ne sont pas suffisantes pour représenter dans la base toutes les propriétés opératoires des objets des applications de CAO. Ces règles ne décrivent pas, par exemple, le fait que la modification d'un composant d'un objet en cours de conception, tel qu'un circuit électronique ou une machine quelconque, peut occasionner la dégradation des performances de ce circuit ou de cette machine et que, par conséquent, l'objet en cours de conception doit subir un processus particulier de vérification.

En vue de représenter les propriétés opératoires des objets, dans certains systèmes (notamment ceux du domaine des langages de programmation) chaque classe doit contenir la description des opérateurs de ses instances. Dans ce cas, ces opérateurs constituent le seul moyen de manipuler les instances de chaque classe.

Cette approche de modélisation qui caractérise essentiellement les systèmes basés sur la notion de types abstraits (*e.g.*, CLU [LSAS 77] et Alphard [ShWL 77]) se trouve aussi dans des infrastructures centrées-objet (*e.g.*, Smalltalk-80 et Loops), et dans certains cas (*e.g.*, Taxis) les opérateurs sont considérés comme des objets qui peuvent être organisés dans des hiérarchies comme celles des super-classes et sous-classes (*voir* figure 2.7), de telle sorte qu'ils peuvent aussi hériter des propriétés de leurs "super-opérateurs" (*e.g.*, le type de paramètres, les préconditions d'exécution, etc.).

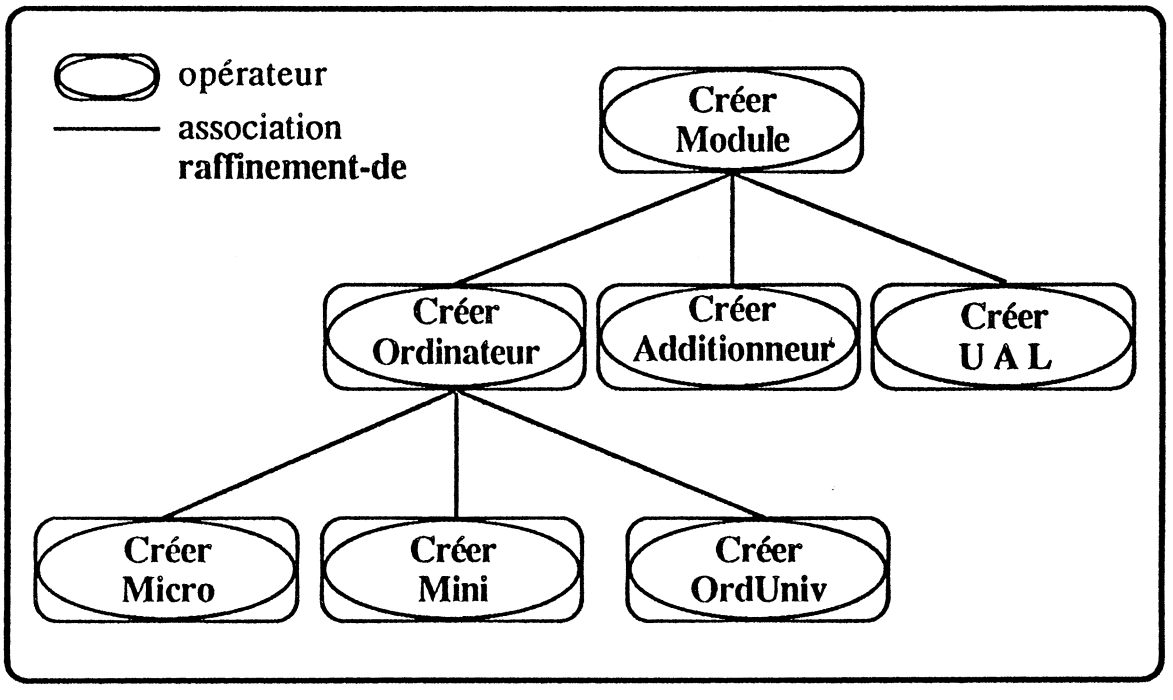


FIGURE 2.7
Exemple d'organisation hiérarchisée des opérateurs d'objets.

Par exemple, si les ordinateurs sont des cas particuliers de module électronique, il est évident que leur création doit respecter toutes les préconditions de création de modules électroniques (*e.g.*, vérifier que aucune version du module électronique n'a été définie auparavant, et que l'utilisateur a le droit de réaliser cette opération).

Ceci étant, il est souhaitable que le système de représentation de connaissances permette la définition d'opérateurs tels que "Créer_Module", contenant ces préconditions d'exécution, de telle sorte que l'on puisse définir l'opérateur "Créer_Ordinateur" héritant des ces préconditions et, en plus, contenant d'autres préconditions propres à la création d'ordinateurs (*e.g.*, vérifier que le processeur indiqué par la propriété "unité_centrale", caractéristique de chaque ordinateur, soit capable d'adresser tout l'espace de mémoire défini par sa propriété "taille de mémoire").

D'autres systèmes (*e.g.*, fun [CarW 86] et LM [Harp 86]) permettent de définir des opérateurs qui appartiennent pas forcément à une seule classe mais à toute une collection de classes qui n'ont pas besoin d'être associées comme des spécialisations les unes des autres. Ces opérateurs peuvent ainsi avoir comme opérandes des instances de classes différentes, et pour cette raison ils sont appelés **opérateurs polymorphes.**

Cette approche de modélisation semble être adéquate aux applications de bases de données, du fait qu'elles ont parfois besoin d'opérateurs associés à plusieurs classes [McLS 80]. Le verrouillage[1] d'objets, préalable à certaines opérations de mise à jour, en constitue un exemple. Nous verrons au chapitre suivant que, dans OBMS, il est possible de définir des "objets opérateur" dont les propriétés déterminent les propriétés opératoires d'instances de classes éventuellement différentes.

2.5.4. Les associations composant-de

Pour conclure le présent chapitre, nous allons faire deux remarques concernant les associations composant-de. La première est que dans certaines infrastructures de modélisation aucun objet ne peut être en même temps composant de deux objets différents [WonM 77], tandis que dans OBMS tout objet peut être composant de plusieurs objets. Le principal avantage de ce partage a été mentionné dans la section 2.4.1. Un autre avantage est la simplification du processus de spécification de propriétés des objets.

---

[1] Verrouiller un objet consiste essentiellement à empêcher l'accès concurrent à cet objet. Une description détaillée des différents protocoles de verrouillage utilisés dans le domaine des bases de données se trouve dans [DelA 82].

Il y a aussi des systèmes (*e.g.*, SMH+ [Brod 82]) où les associations composant-de sont vues comme des relations binaires d'ordre partiel, c'est à dire, transitives, asymétriques et réflexives. Cependant, dans OBMS la contrainte d'asymétrie n'est pas imposée, car il y a plusieurs cas où les associations entre objets sont symétriques (*e.g.*, frère-de, placé-à-côté-de, etc.).

Tout de même, les concepteurs peuvent toujours spécifier qu'un objet n'est pas partageable, et qu'une association doit être asymétrique et même acyclique (c'est-à-dire que aucun agrégat ne peut être composant de ses composants, ou de ses sous-composants, etc.), lorsqu'ils le considèrent nécessaire. La spécification de telles contraintes constitue le sujet du chapitre suivant.

CHAPITRE 3

Paradigme de modélisation

3. Paradigme de modélisation

3.1. Introduction

Au chapitre précédent nous avons mentionné que dans notre infrastructure de représentation de connaissances il y a des objets élémentaires, dont la sémantique est supposée connue, et des objets non élémentaires, dont la sémantique est définie en termes de leurs associations avec d'autres objets.

Nous avons mentionné aussi que ces associations peuvent être représentées comme des propriétés structurelles des objets non élémentaires, et caractérisées par les contraintes structurelles et opératoires qui doivent être respectées par les objets associés.

Etant donné que les associations entre les objets des applications de CAO sont en général nombreuses et de types très variés (*e.g.*, concepteur-de, version-de, spécialisation-de, etc.), la modélisation de ces applications requiert non seulement la représentation de larges structures imbriquées de manière complexe (cela peut être réalisé en utilisant tout simplement des "records" de Pascal ou des listes de Lisp), mais aussi la représentation d'un grand nombre de contraintes qui doivent être mises en vigueur pour maintenir la base d'objets cohérente du point de vue sémantique.

Dans le présent chapitre nous allons décrire les concepts, les techniques et les opérateurs qui peuvent être utilisés pour créer et détruire des objets, établir, mettre à jour, supprimer, et connaître leurs associations, et représenter dans les bases d'objets les différentes contraintes d'intégrité qui doivent être mises en vigueur par OBMS.

3.2. Les propriétés prédéfinies

Dans notre infrastructure de représentation de connaissances, les contraintes d'intégrité universelles, intrinsèques aux associations composant-de, instance-de, et spécialisation-de, sont connues et mises en vigueur par le système. Cela veut dire que le concepteur de la base n'a besoin ni de les spécifier ni de les valider.

De plus, les noms des propriétés correspondant aux associations instance-de et spécialisation-de sont prédéfinis par le système. Par conséquent, pour représenter ces associations dans une base d'objets il ne faut que spécifier la valeur de ces propriétés, que nous appelerons "propriétés prédéfinies", dans l'un des objets associés.

L'objet ordinateur-1 peut, par exemple, être défini comme une instance de la classe Ordinateur en incluant son identificateur dans la propriété (prédéfinie) "instances" de cette classe. L'expression OBMS qui permet d'établir cette association est la suivante[1]:

```
( :+instance  %Ordinateur %ordinateur-1 )
```

Au niveau conceptuel, l'association instance-de implique l'existence de la propriété "instances" des objets classe, mais aussi l'existence de la propriété "classes" des objets instance. Par conséquent, la classification de l'ordinateur-1 peut s'effectuer d'une manière alternative qui consiste à inclure l'identificateur de la classe Ordinateur dans la propriété (prédéfinie) "classes" de l'instance ordinateur-1:

```
( :+classe  %ordinateur-1 %Ordinateur )
```

De manière tout à fait analogue, la classe Ordinateur peut être définie comme une spécialisation de la classe Module (*Cf.* figure 2.5) en incluant l'identificateur de la première dans la propriété (prédéfinie) "sous-classes" de la deuxième, ou l'identificateur de la deuxième dans la propriété (prédéfinie) "super-classes" de la première:

```
( :+sous-classe  %Module %Ordinateur )

ou      ( :+super-classe  %Ordinateur %Module )
```

Un fait important concernant les propriétés prédéfinies est que, indépendamment que l'on utilise :+instance ou :+classe, et :+sous-classe ou :+super-classe, ces simples actions suffisent pour représenter dans la base d'objets toutes les contraintes sur ordinateur-1 et Ordinateur intrinsèques aux associations instance-de, ainsi que les contraintes sur Ordinateur et Module intrinsèques aux associations spécialisation-de.

---

[1]Les symboles précédés par le signe "%" sont utilisés dans cette thèse pour représenter aussi bien le surrogate des objets (c'est-à-dire leur identificateur généré par le système), que les variables contenant ces identificateurs. Ainsi, le symbole %Ordinateur représente le surrogate de l'objet Ordinateur. Une description de la structure et du comportement des opérateurs OBMS se trouve dans l'annexe B.

Symétriquement, la suppression de l'association instance-de entre les objets ordinateur-1 et Ordinateur peut se réaliser en éliminant l'identificateur du premier de la propriété "instances" du deuxième, ou l'identificateur du deuxième de la propriété classes du premier:

```
( :-instance  %Ordinateur %ordinateur-1 )

ou      ( :-classe     %ordinateur-1 %Ordinateur )
```

et la suppression de l'association spécialisation-de entre les classes Ordinateur et Module peut s'effectuer en éliminant l'identificateur de la première de la propriété "sous-classes" de la deuxième, ou l'identificateur de la deuxième de la propriété "super-classes" de la première:

```
( :-sous-classe     %Module %Ordinateur )

ou      ( :-super-classes  %Ordinateur %Module )
```

Enfin, pour savoir quelles sont les classes auxquelles a été associée une instance telle que ordinateur-1, où les objets qui sont associés comme des sous-classes, des super-classes ou des instances d'une classe telle que Module ou Ordinateur, il suffit d'analyser la valeur des propriétés représentant ces associations.

Par exemple, en accord avec les figures 2.4 et 2.5, les résultats rendus par les opérateurs OBMS qui permettent de faire cette analyse de propriétés sont les suivants:

```
( :$classes  %ordinateur-1 )

=  %Ordinateur

( :$instances  %Ordinateur )

=  %ordinateur-1, %ordinateur-2, ..., %ordinateur-N

( :$sous-classes  %Ordinateur )

=  %Micro_Ordinateur, %Mini_Ordinateur, %Ord_Universel
```

```
( :$super-classes %Ordinateur )
= %Module
( :$super-classes %Module )
= ( )
```

3.3. La création et la destruction de classes

La technique utilisée dans la section précédente pour établir, supprimer et connaître les associations entre objets, par simple ajout (+), élimination (-) et analyse ($) de valeurs des propriétés des objets, caractérise de manière globale le paradigme de modélisation qui peut être utilisée pour la représentation de connaissances dans toute base d'objets gérée par OBMS, de la même façon que le paradigme d'envoi de messages caractérise la plupart des langages de programmation orientée-objet, ou que le paradigme de composition de fonctions caractérise les langages fonctionnels.

En effet, même la création d'objets peut être réalisée de cette manière dans OBMS, car chaque base d'objets est aussi un objet dont les propriétés structurelles représentent ses associations avec les objets qui sont définis dans cette base.

Par exemple, la création de la classe Ordinateur dans la base d'objets BASE-1 consiste à ajouter l'identificateur de cette classe (c'est-à-dire son surrogate généré par le système) dans la propriété prédéfinie "classes" de cette base:

```
( :+classe %BASE-1 'Ordinateur )
```

Les opérateurs OBMS sont des fonctions, c'est-à-dire, tous rendent un résultat. En particulier, l'opérateur précédent retourne le surrogate de la classe créée:

```
%Ordinateur <-- ( :+classe %BASE-1 'Ordinateur )
```

De plus, ces opérateurs évaluent leurs arguments et peuvent ainsi être utilisés comme des arguments d'autres opérateurs. Cependant, par souci de clarté, dans cette thèse nous n'utilisons en général pas la composition de fonctions. Nous avons préféré utiliser dans nos exemples le symbole "<--" pour dénoter, de manière générique, tout opérateur qui peut être employé pour affecter une valeur à une variable (*e.g.*, ":=" en Pascal, et "setq" en Lisp), et utiliser cette variable comme argument des opérateurs subséquents.

D'autre part, le surrogate des objets classe peut toujours être connu en utilisant l'opérateur ":classe?", qui reçoit comme argument le nom d'une classe, tandis que le surrogate des objets base peut être connu en utilisant l'opérateur ":base?":

```
( :classe? 'Ordinateur )

= %Ordinateur

(:base? 'BASE-1)

= %BASE-1
```

Par ailleurs, le surrogate des objets base peut aussi être connu lors de la création de ces objets:

```
( :+base 'BASE-1 )

= %BASE-1
```

En outre, l'exemple concernant la création de la classe Ordinateur met en relief le fait que les noms des propriétés ne sont que des identificateurs symboliques utilisés pour distinguer les différentes propriétés de chaque objet. Autrement dit, ces noms n'ont aucune connotation concernant la sémantique des associations représentées par ces propriétés. Par conséquent, les propriétés qui ont le même nom dans deux objets distincts ne représentent pas nécessairement le même type d'association entre objets.

Dans le cas précédent, la seule similitude entre la propriété "classes" des objets instance et des objets base est qu'elles représentent des associations avec un objet classe, et cette similitude peut être utilisée pour définir des

opérateurs polymorphes tels que :+classe[1]. Pourtant, les contraintes d'intégrité intrinsèques aux associations entre une classe et une base sont complètement différentes des contraintes sur les associations entre une classe et une instance. En d'autres termes, la sémantique de ces associations n'est pas du tout la même. Une association entre une base et une classe n'impose, par exemple, aucune contrainte concernant les attributs des instances de cette classe.

Dans OBMS les seules contraintes concernant les associations entre une base et une classe sont les suivantes:

bc1)	Deux classes associés à la même base ne peuvent pas avoir le même nom.
bc2)	La suppression d'une association entre une base et une classe entraîne la destruction préalable de cette classe. Ainsi, si en accord avec les règles relatives à la destruction d'objets (*e.g.*, aa8, ci6 et cc8) cette classe ne peut pas être détruite, alors la classe et la base ne peuvent pas être dissociées.

En accord avec la règle bc2, par exemple, l'expression

```
( :-classe %BASE-1 %Ordinateur )
```

entraîne la destruction de la classe Ordinateur. Toutefois, cette destruction ne peut pas être réalisée si, par exemple, une instance de cette classe est un composant d'un autre objet et, en plus, elle n'est pas associée à une autre classe, car en accord à la règle aa8 cette instance ne peut pas être détruite et, par conséquent, la classe Ordinateur non plus (*Cf.* règle ci6). Dans ce cas, la classe Ordinateur et la BASE-1 ne peuvent pas être dissociées et le système en signale les raisons.

D'autre part, si les concepteurs veulent connaître la liste de classes qui ont été créées dans une base telle que BASE-1, ils peuvent utiliser l'expression

```
( :$classes %BASE-1 )
```

[1]L'opérateur :+classe est polymorphe, car il associe un objet X à une classe, que cet objet X soit une instance ou une base. Dans l'implémentation d'OBMS plusieurs opérateurs polymorphes ont été définis, et plusieurs types de polymorphisme ont été utilisés, mais l'emploi du polymorphisme est un choix d'implémentation tout à fait indépendant du paradigme de modélisation présenté dans ce chapitre et, en général, de l'infrastructure de représentation de connaissances définie dans cette thèse. En conséquence, nous n'analyserons pas les "particularités polymorphes" des opérateurs d'OBMS.

3.4. La création et la destruction d'instances

Le paradigme de modélisation qui peut être utilisé dans notre infrastructure de représentation de connaissances peut être vu comme un processus similaire à celui de l'élaboration de représentations graphiques comme celles des figures 2.1, 2.3 et 2.4:

a) On crée d'abord un nœud (objet) et, ensuite, on établit ses associations avec d'autres nœuds, créés auparavant, à l'aide d'arcs (propriétés) qui peuvent être ajoutés (+), supprimés (-) ou tout simplement analysés ($).
b) Plusieurs types d'arcs (doubles, simples, continus, discontinus, etc.) peuvent être utilisés pour représenter les différents types d'associations, et plusieurs types de nœuds (des cercles, des carrés, etc.) peuvent être utilisés pour représenter les différentes catégories d'objets.
c) Le résultat de ce processus est un réseau (base d'objets) qui constitue un modèle conceptuel d'une application ou système spécifique.

Par exemple, pour représenter dans la base d'objets BASE-1 l'ordinateur-1 illustré dans la figure 2.2, on crée d'abord un objet instance, au moyen de l'opérateur qui permet d'ajouter l'identificateur de cette instance (c'est-à-dire son surrogate généré par le système) dans la propriété prédéfinie "instances" de cette base

```
%ordinateur-1 <-- ( :+instance %BASE-1 )
```

On spécifie ensuite toutes les propriétés représentant ces associations avec d'autres objets. Les propriétés "concepteur", "projet", "syst_exploit" et "unité_centrale", illustrées dans la figure 2.2, représentent les composants du concept "ordinateur-1", c'est-à-dire les caractéristiques de cet objet (*Cf.* section 2.4.1).

Les noms des propriétés représentant les associations composant-de entre objets ne sont pas prédéfinis dans OBMS et, en conséquence, leur spécification doit inclure leur nom et leur valeur. Par exemple:

```
( :+caractéristique %ordinateur-1 'concepteur %Paul )
```

Par ailleurs, ces caractéristiques peuvent être définies lors de la création d'un objet:

```
( :+instance %BASE-1   ;Création de l'ordinateur-1.
      'concepteur      %Paul
      'unité_centrale  %processeur-1
      'syst_exploit    "UNIX"
      'projet          %ALPHA)
```

Il faut cependant noter que cette possibilité d'abréviation est une question de "sucre syntaxique" qui n'est pas indispensable au niveau strictement conceptuel.

Toujours avec le même paradigme de modélisation, si les concepteurs veulent connaître la liste d'instances qui ont été créées dans une base telle que BASE-1, ils peuvent utiliser l'expression

```
( :$instances %BASE-1 )
```

Au chapitre précédent, nous avons mentionné que dans OBMS les instances n'ont pas besoin d'être associées à des objets classe, et la création de l'ordinateur-1, illustrée ci-dessus, en constitue un exemple. Toutefois, le même exemple montre que la création d'objets instance implique leur association avec un objet base.

Dans OBMS, ces associations entre des objets base et des objets instance sont régies par une seule contrainte d'intégrité:

bi1) La suppression d'une association entre une base et une instance entraîne la destruction préalable de cette instance. Ainsi, si en accord avec les règles relatives à la destruction d'objets (*e.g.*, aa8 et ci6) cette instance ne peut pas être détruite, alors la base et l'instance ne peuvent pas être dissociées.

En accord avec cette règle, l'expression

```
( :-instance %BASE-1 %ordinateur-1 )
```

entraîne la destruction de l'ordinateur-1, mais cette destruction ne peut pas être réalisée si, par exemple, cet objet est un composant d'un autre objet (*Cf.* règle aa8). Dans ce cas, l'ordinateur-1 et la BASE-1 ne peuvent pas être dissociées et le système en signale la raison.

Un fait qui doit être remarqué est que les actions de contrôle de la cohérence, illustrées dans les exemples précédents, sont réalisées par OBMS en fonction des règles bc1, bc2, bi1 et celles définies au chapitre précédent, et que ces règles ne sont jamais suffisantes pour définir toute la sémantique des applications de CAO. Effectivement, ces règles ne définissent aucune des contraintes d'intégrité spécifiques à chacune de ces applications (*e.g.*, «l'unité centrale des ordinateurs doit être un objet de la classe correspondant aux processeurs»).

Dans la section suivante nous allons décrire la manière dont les concepteurs peuvent spécifier les contraintes d'intégrité qui ne sont pas connues par le système et qui définissent la signification (sémantique) des propriétés non prédéfinies telles que "concepteur", "unité_centrale", etc.

3.5. Les caractéristiques filtrées

Dans OBMS, plusieurs contraintes structurelles relatives aux associations entre les objets agrégats et agrégés peuvent être spécifies comme des contraintes sur la valeur de leurs caractéristiques et, en général, sur la valeur de leurs attributs, c'est-à-dire comme des contraintes sur les propriétés qui représentent les associations entre un objet et ses composants.

Ces spécifications sont de nature générique, en ce sens qu'elles ne définissent pas des contraintes particulières à une association spécifique ou "factuelle" entre deux instances (*e.g.*, «le concepteur de l'ordinateur-1 doit être Paul»), mais des contraintes sur un type d'associations qui caractérisent les instances d'une classe d'objets (*e.g.*, «les concepteurs des ordinateurs doivent être des ingénieurs»).

En d'autres termes, ces spécifications sont les composants essentiels des objets classe, d'après la définition de ces objets donnée au chapitre précédent (*Cf.* section 2.4.2). En conséquence, dans OBMS elles constituent la valeur des caractéristiques des classes.

Par exemple, la classe Ordinateur illustrée dans la figure 3.1 indique que l'unité centrale de toutes ses instances doit être une instance de la classe Processeur. L'expression OBMS qui permet de spécifier cette contrainte est:

```
( :+caractéristique-f %Ordinateur
        'unité_centrale ':est-un-Processeur? )
```

où le terme ":est-un-Processeur?" dénote un prédicat unaire dont la valeur est VRAI si et seulement si son argument appartient à la classe Processeur.

CLASS Ordinateur	
super-classes	: Module
sous-classes	: Micro_Ordinateur, Mini_Ordinateur, Ordinateur_Universel
instances	: ordinateur-1, ordinateur-2, ..., ordinateur-N
unité_centrale	: instance de Processeur

FIGURE 3.1
Structure de la classe Ordinateur

Après cette spécification, le système vérifie que ce prédicat prend la valeur VRAI lorsqu'il reçoit comme argument chacune des valeurs affectées à la propriété unité_centrale des instances de la classe Ordinateur. Si l'une des évaluations de ce prédicat retourne FAUX, alors le système n'accepte pas la spécification de cette contrainte et indique la raison.

De plus, si cette spécification est acceptée par le système, le prédicat :est-un-Processeur? sera évalué chaque fois qu'une nouvelle instance est associée à la classe Ordinateur, ou que la valeur de la propriété unité_centrale de l'une de ses instances est modifiée, en utilisant comme argument la valeur de cette propriété. Si cette évaluation retourne la valeur FAUX, le système ne permet pas la modification de l'instance ou l'association de la nouvelle instance à la classe Ordinateur, afin de maintenir l'intégrité sémantique de la base d'objets.

Ce processus de **filtrage** réalisé par le système à l'aide de prédicats est la raison pour laquelle dans OBMS les propriétés des objets classe, telles que unité_centrale, qui caractérisent les objets de manière prédicative, sont appelées caractéristiques filtrées ou, tout simplement, **caractéristiques-f**. D'autres caractéristiques non filtrées des objets classe seront décrites dans la section 3.7.

En outre, les prédicats associés aux objets classe peuvent être des prédicats n-aires. Par exemple, pour spécifier que le budget annuel de tout projet de conception doit être supérieur à 500 000 francs, le concepteur de la base d'objets peut utiliser l'expression

```
( :+caractéristique-f %Projet 'budget '(> 500000) )
```

Dans OBMS, ces prédicats sont représentés dans les bases d'objets comme des fonctions booléennes qui peuvent être écrites dans un langage de programmation tel que Lisp, PL/1, Pascal, C, Fortran ou Assembleur.

La spécification de prédicats dans OBMS est une technique de représentation de connaissances par **attachement procédural**, utilisée dans certains systèmes de modélisation développés en intelligence artificielle. Dans OBMS, cet attachement procédural donne au système un caractère extensible, car il permet de représenter, dans les bases d'objets, des connaissances particulières à une application spécifique qui ne sont pas prédéfinies par OBMS, et qui sont nécessaires pour maintenir ces bases dans un état cohérent, du point de vue sémantique.

3.6. Les descripteurs

Au chapitre précédent nous avons mentionné que dans OBMS la valeur d'une propriété identifie un ou plusieurs objets associés à celui ayant cette propriété. Cela signifie que toute propriété peut en principe être multivaluée.

Ceci étant, le mécanisme de représentation de contraintes sur les propriétés des objets doit permettre non seulement la spécification de contraintes sur chacun des éléments constituant la valeur de ces propriétés, mais aussi sur les cardinalités de chaque propriété (c'est-à-dire le nombre maximal et minimal d'éléments qui peuvent constituer sa valeur), et sur la possibilité d'avoir ou pas des éléments dupliqués dans leurs valeurs.

Dans OBMS toutes ces spécifications sont regroupées dans des objets que nous appelons **descripteurs**, et qui constituent la valeur des caractéristiques des objets classe. Ces objets sont créés par le système lors de la définition des caractéristiques des classes, et leurs propriétés contient les spécifications fournies par les utilisateurs. Dans le cas des caractéristiques-f, les propriétés de leurs descripteurs (appelés aussi descripteurs-f) sont:

a) la propriété **filtre**, qui contient le prédicat qui doit être utilisé par le système pour réaliser le filtrage décrit dans la section précédente.
b) la propriété **cardinalités**, dont la valeur par défaut est représentée par "(0 *)", pour indiquer que les caractéristiques des instances peuvent avoir un nombre quelconque de valeurs.
c) la propriété **ensemble**, dont la valeur (booléen) est VRAI par défaut et indique que la propriété ne peut pas avoir des éléments dupliqués.

La figure 3.2 illustre la structure du descripteur correspondant à la caractéristique unité_centrale de la classe Ordinateur.

CLASS Ordinateur				
unité_centrale	: [filtre	=	:est-un-Processeur?
	cardinalités	=	(0 *)
	ensemble	=	VRAI]

FIGURE 3.2
Structure du descripteur de la propriété unité_centrale.

Cette distinction entre les contraintes relatives à la valeur des caractéristiques, vue comme un seul objet (c'est-à-dire les contraintes définies par les propriétés "cardinalités" et "ensemble"), et les contraintes relatives à chacun de ses éléments (c'est-à-dire celles définies par le filtre), permet de les spécifier, de les modifier et de les analyser de manière indépendante.

Dans l'implémentation actuelle d'OBMS, toutes ces actions peuvent être réalisées en utilisant seulement trois opérateurs (:filtre, :cardinalités et :ensemble), qui peuvent être invoqués avec deux ou avec trois arguments. Par exemple, pour spécifier que les ordinateurs doivent avoir au moins une unité centrale et au plus 6, les concepteurs peuvent utiliser l'expression suivante:

```
( :cardinalités %Ordinateur 'unité_centrale '(1 6) )

= (1 6)          ; la réponse du système.
```

Par contre, si les concepteurs veulent seulement connaître les contraintes sur les cardinalités de cette propriété, ils peuvent utiliser le même opérateur, mais avec deux arguments:

```
(:cardinalités %Ordinateur 'unité_centrale )

= (1 6)
```

Cette approche a été suivie en vue de réduire au minimum possible le nombre total d'opérateurs de base du système, et en profitant du fait que les propriétés des descripteurs sont monovaluées. Ce qui permet de remplacer les opérateurs d'ajout et de suppression de valeurs par un seul opérateur de mise à jour.

Cependant, au niveau strictement conceptuel, les avantages et les inconvénients d'OBMS ne résultent pas des opérateurs implémentés, mais des concepts (*e.g.*, objets, propriétés, et contraintes d'intégrité) et des techniques (*e.g.*, l'emploi de mécanismes d'abstraction, l'identification d'objets par des surrogates, et la représentation de contraintes à l'aide de descripteurs) qui peuvent être utilisés pour la représentation de connaissances dans des bases d'objets.

3.7. Les caractéristiques préétablies et calculées

Très souvent les concepteurs ont besoin de caractériser les objets non seulement de manière prédicative comme dans les exemples précédents, mais aussi en spécifiant une valeur précise pour un attribut donné (*e.g.*, «le superviseur de tout module électronique en cours de conception doit être Paul»), ou une fonction qui retourne cette valeur (*e.g.*, «la structure d'un circuit doit être représentée par une liste contenant le nom du circuit, lorsqu'il n'a pas de composants, ou par une liste contenant les noms de ses sous-composants les plus internes»).

Dans ces cas, les caractéristiques sont appelées, respectivement, **caractéristiques-p** (préétablies) et **caractéristiques-c** (calculées), et peuvent être spécifiées de la manière suivante:

```
( :+caractéristique-p %Module 'superviseur %Paul )
```

```
( :+caractéristique-c  %Circuit
      'structure '(:calculer-structure) )[1]
```

En outre, si l'objet classe spécifie déjà de manière concrète la valeur d'une propriété quelconque, il est tout à fait inutile de spécifier cette valeur dans toute les instances de la classe en question. En fait, OBMS interdit de telles "ré-spécifications". Si une classe quelconque a des caractéristiques prédéfinies ou calculées, elles sont induites à ses instances et, en conséquence, celles-ci ne peuvent ni les redéfinir ni les spécialiser.

Pour la même raison, les descripteurs utilisés pour représenter ces spécifications n'ont pas les propriétés "cardinalités" et "ensemble". En effet, les descripteurs des caractéristiques préétablies (appelés aussi descripteurs-p) n'ont qu'une seule propriété qui s'appelle "valeur" et qui contient la valeur préétablie de la caractéristique en question.

De manière similaire, les descripteurs des caractéristiques calculées, appelés aussi descripteurs-c, ont une seule propriété (la propriété "fonction") qui contient la fonction qui doit être invoquée pour obtenir la valeur d'une caractéristique donnée.

En réalité, la distinction entre les caractéristiques préétablies et les caractéristiques calculées (ainsi qu'entre les descripteurs-p et les descripteurs-c) n'existe qu'au niveau des opérateurs :+caractéristique-p et :+caractéristique-c, et cette distinction n'a été introduite que pour simplifier la spécification des caractéristiques préétablies.

En effet, dans OBMS la valeur de ces caractéristiques est un descripteur-c dont la propriété "fonction" contient la fonction identité, c'est-à-dire une fonction qui retourne son argument comme résultat. Ainsi, l'expression

```
( :+caractéristique-p  %Module 'superviseur %Paul )
```

est absolument équivalente à l'expression

```
( :+caractéristique-c  %Module
      'superviseur '(:identité %Paul ) )
```

[1]Une spécification alternative est donnée comme exemple dans la section 5 de l'annexe B.

Cela implique que si cette vision plus uniforme de l'infrastructure de représentation de connaissance est considérée plus adéquate aux concepteurs, l'opérateur :+caractéristique-p est inutile.

Par uniformité aussi, les valeurs de toutes les propriétés prédéfinies des objets classe peuvent être vues comme des descripteurs dont les cardinalités sont invariablement "(0 *)", la propriété "ensemble" est toujours VRAI, et le filtre est spécifié par le système, en utilisant comme paramètres de certains prédicats prédéfinis les identificateurs de classes donnés par l'utilisateur.

Par exemple, la valeur de la propriété "super-classes" de la classe Ordinateur illustrée dans la figure 3.1, peut être vue comme un descripteur dont le filtre est le prédicat ":spécialisation-valide?" (*voir* figure 3.3), qui prend la valeur VRAI si et seulement si les classes Ordinateur et Module vérifient toutes les contraintes intrinsèques aux associations spécialisation-de.

CLASS Ordinateur		
super-classes	: [	filtre = (:spécialisation-valide? %Module)
		cardinalités = (0 *)
		ensemble = VRAI]

FIGURE 3.3
Valeur de la propriété super-classes de la classe Ordinateur, vue comme un descripteur.

En général, la spécification de descripteurs peut être vue comme un processus similaire à la spécification des "démons" [Wins 84] dans certaines infrastructures de représentation de connaissances, développées dans le domaine de l'intelligence artificielle, c'est-à-dire des procédures qui implémentent les mécanismes de raisonnement nécessaires pour:

a) déterminer si les valeurs des propriétés des objets satisfont certaines contraintes d'intégrité (c'est-à-dire des procédures similaires aux filtres des descripteurs-f dans OBMS),
b) obtenir la valeur d'une propriété (c'est-à-dire des procédures similaires aux fonctions des descripteurs-c dans OBMS).

Le concept de démon est pourtant plus puissant que celui de descripteurs dans OBMS, car les démons sont parfois (*e.g.*, [WinH 84]) des procédures qui peuvent être activées, entre autres, pour connaître la valeur par défaut

d'une propriété, et pour enchaîner des opérations qui doivent se suivre pour assurer l'intégrité sémantique de la base d'objets.

Par exemple, lorsqu'un concepteur est licencié, c'est-à-dire, lorsqu'une instance de la classe Concepteur est éliminée de la base d'objets, il faut d'abord enlever l'identificateur de ce concepteur de la propriété "employés" de tous les projets où il travaille.

La raison pour laquelle nous n'avons pas inclus dans OBMS la notion de valeur par défaut a été décrite dans la section 5 du chapitre précédent, lors de l'analyse du concept de classe. En ce qui concerne l'enchaînement d'opérations, nous verrons dans la section 3.11 que les utilisateurs d'OBMS peuvent définir des opérateurs tels que "Licencier_Concepteur", et que ces opérateurs peuvent inclure la spécification de toutes les actions qui doivent être effectuées pour maintenir cohérente la base d'objets lorsque l'on licencie un concepteur.

Par ailleurs, la suppression et l'analyse des caractéristiques des objets classe peuvent être réalisées à l'aide des opérateurs :-caractéristique et :$caractéristique, que ces caractéristiques soient filtrées, calculées ou préétablies. Pour supprimer, par exemple la caractéristique "structure" de la classe Circuit, les concepteurs peuvent utiliser l'expression

```
( :-caractéristique %Circuit 'structure )
```

De manière similaire, l'expression suivante permet de connaître les caractéristiques de la classe Ordinateur, illustrée dans la figure 3.1:

```
( :$caractéristiques %Ordinateur )

= unité_centrale    [ filtre = :est-un-Processeur?
                      cardinalités   = (0 *)
                      ensemble       = VRAI ]
```

Ce dernier opérateur ne montre pourtant que les attributs propres à la classe dénotée par son argument. Autrement dit, il ne retourne pas les attributs dont cette classe hérite de ses super-classes. Ainsi, si la classe Ordinateur est une sous-classe de la classe Module, et si celle-ci a la structure illustrée dans la figure 3.4, alors les attributs de la classe Ordinateur sont "unité_centrale" et "superviseur".

CLASSE Module	
sous-classes	: Ordinateur, Processeur,...
concepteurs	: instances d'Ingénieur ;=> caractérist-f.
superviseur	: Paul ;=> caractéristique-p.

FIGURE 3.4
Structure de la classe Module.

Lorsque l'on s'intéresse à connaître non seulement les caractéristiques mais tous les attributs d'une classe, on doit utiliser l'opérateur ":$attributs":

```
( :$attributs %Ordinateur )

= superviseur       [ valeur  = %Paul ]
  unité_centrale    [ filtre  = :est-un-Processeur?
                      cardinalités = (0 *)
                      ensemble     = VRAI ]
```

3.8. Les attributs des instances

Dans la section précédente nous avons vu que toute classe peut avoir des caractéristiques préétablies, qui spécifient une valeur précise pour une propriété de ses instances, et des caractéristiques calculées, qui spécifient la fonction qui retourne cette valeur, et nous avons mentionné que les propriétés définies de cette manière sont induites aux instances des classes.

Les caractéristiques préétablies et les caractéristiques calculées sont donc des attributs des instances (*Cf.* section 2.4.1), car elles ne sont pas particulières à chaque instance mais communes à toutes les instances de chaque classe. Ceci étant, il est nécessaire de compter sur des opérateurs permettant d'analyser les caractéristiques et les attributs des instances, mais aussi leurs caractéristiques et leurs attributs internes, c'est-à-dire les caractéristiques et les attributs dont toute instance hérite de ses composants (*Cf.* règle aa7).

Les opérateurs OBMS qui permettent d'analyser ces propriétés sont :$caractéristiques, :$attributs, :$caractéristique-interne et :$attribut-interne. Par exemple, d'après la figure 3.1, la classe Ordinateur est une sous-classe de la classe Module; d'autre part, d'après la figure 3.5, l'unité centrale de l'ordinateur-1 est instance de la classe Processeur, et cette classe

est aussi une sous-classe de la classe Module. Ceci implique donc que les classes Ordinateur et Processeur héritent des propriétés de la classe Module; en particulier la caractéristique-p "superviseur" illustrée dans la figure 3.4.

CLASSE Processeur	
super-classe	: Module, Classe-C23
instances	: processeur-1, processeur-21,...
technologie	: c-mos ;=> caractéristique-p.
INSTANCE ordinateur-1	
classes	: Ordinateur
concepteurs	: Paul
projet	: ALPHA
unité_centrale	: processeur-1
syst_exploit	: UNIX
INSTANCE processeur-1	
classes	: Processeur, Classe-CX
concepteurs	: Valéria, Daniéla
date_création	: 15/février/87

FIGURE 3.5

De plus, la classe Module spécifie que le superviseur de ses instances doit toujours être Paul, et la classe Processeur spécifie que la technologie utilisée pour construire ses instances doit toujours être c-mos.

Lorsque la base d'objets est dans cet état, les résultats des opérateurs qui permettent d'analyser les attributs de l'ordinateur-1 son les suivants:

```
( :$caractéristiques %ordinateur-1 )

=   concepteur        (%Paul)
    projet            (%ALPHA)
    unité_centrale    (%processeur-1)
    syst_exploit      (UNIX)
```

```
( :$attributs %ordinateur-1 )

=   superviseur       (%Paul)
    concepteur        (%Paul)
    projet            (%ALPHA)
    unité_centrale    (%processeur-1)
    syst_exploit      (UNIX)

( :$caractéristique-interne
    'date_création:unité_centrale  %ordinateur-1 )

=   15/février/87

( :$caractéristique-interne
    'technologie:unité_centrale  %ordinateur-1 )

=   ( )

( :$attribut-interne
    'technologie:unité_centrale  %ordinateur-1 )

=   c-mos
```

Ces résultats méritent deux commentaires additionnels qui permettront de mieux connaître le comportement des opérateurs que l'on vient d'utiliser. Premièrement, il est intéressant de noter que pour obtenir les résultats des opérateurs :$attributs, :$caractéristique-interne, et :$attribut-interne, le système doit utiliser un type de raisonnement que nous appelons **raisonnement par abstraction**, car il implique l'emploi de mécanismes d'inférence qui sont basés sur les principes d'induction et d'héritage de propriétés inhérents aux associations créées par l'emploi de mécanismes d'abstraction.

En effet, dans ce type de raisonnement, le système reconnaît les hiérarchies d'agrégation et de généralisation définies dans la base d'objets, et détermine les propriétés des objets en fonction de leurs associations avec des objets des niveaux d'abstraction supérieurs et inférieurs.

Deuxièmement, nous devons mentionner que l'argument technologie:unité_centrale, utilisé avec les opérateurs :$caractéristique-interne et :$attribut-interne, dénote la propriété "technologie" de tous les objets constituant la valeur de la propriété unité_centrale de l'ordinateur-1.

Cette "notation pointée" est un **mécanisme d'indexation** reconnu par ces opérateurs, et permet de construire des arguments dont la structure générale est $a_1:a_2:...:a_n$, où a_i dénote l'ensemble d'objets qui constituent la valeur de l'attribut a_{i+1} des objets dénotés par a_{i+2}.

Dans OBMS, les arguments construits de cette manière sont appelés **arguments structurés**[1], et leur emploi est similaire à l'emploi de «chaînes de références» dans le langage ARIEL [MacG 85], d'«attributs de référence» dans le langage GEM [Zani 83] et de «liens» dans le langage OPAL [CopM84 et MSOP86]. En effet, ces arguments structurés peuvent être utilisés pour formuler des requêtes à l'aide de l'opérateur :retrouver. Par exemple, l'expression

```
( :retrouver    '(concepteurs date_création:unité_centrale)
                %Ordinateur
                '(concepteurs:unité_centrale '= '(%Valéria %Daniéla))
```

peut être utilisée pour formuler la requête

> *«Retrouver les "concepteurs" et la "date de création de l'unité centrale" de tous les ordinateurs dont les "concepteurs de leur unité centrale" soient Valéria et Daniéla»*

En général, l'opérateur :retrouver peut être utilisé pour formuler des requêtes dont la syntaxe peut être paraphrasée comme

```
RETROUVER       x de y de z  ET  abc  ET ... ET  p de q
DE LA CLASSE    C
LORSQUE         f de p = r de q  ET  mm > xx  OU...
                ET h de k contient RETROUVER...
```

Dans [KhoC 86] et [Zani 83], les auteurs présentent des exemples pour montrer que l'emploi d'arguments structurés, et de surrogates, simplifient la plupart des requêtes en éliminant le besoin de spécifier explicitement des thêta-produits et d'employer des variables domaine ou n-uplet.

[1] Dans l'implémentation actuelle d'OBMS les arguments structurés doivent en réalité être précédés par un dièse (*e.g.*, #:technologie:unité_centrale), mais dans nos exemples nous ne l'utiliserons pas, car il a été introduit uniquement pour simplifier cette implémentation.

3.9. Les objets subordonnés

Lors de la spécification des règles d'intégrité universelles, inhérentes aux associations instance-de et spécialisation-de, nous avons mentionné que la destruction d'une classe implique la destruction des instances et des sous-classes qui ne sont pas associées à d'autres classes (*Cf.* règles ci6 et cc8).

Cette contrainte n'a pas été imposée sur les associations entre objets agrégats et objets agrégés, car le partage de composants n'est évidemment possible que si ces objets agrégés ont une existence indépendante de l'existence de leurs agrégats. Pourtant, dans toutes les applications, il y a toujours des objets dont l'existence indépendante n'a aucun sens. L'adresse d'un fournisseur et les éléments d'interface d'un module électronique sont des exemples de ce type d'objets.

Dans notre infrastructure de représentation de connaissances, les objets dont l'existence dépend de l'existence de leurs agrégats sont appelés **objets subordonnés**. Parmi ces objets il y a ceux qui peuvent être partagés par plusieurs objets agrégats (*e.g.*, deux objets peuvent avoir la même adresse), et ceux qui ne sont pas partageables (*e.g.*, les éléments d'interface des modules électroniques).

Par ailleurs, il y a aussi des objets qui ne sont ni subordonnés ni partageables. Enfin, toutes les combinaisons sont possibles et, en conséquence, dans OBMS la subordination et la partageabilité sont deux contraintes qui doivent être spécifiées indépendamment l'une de l'autre. Dans cette section nous allons présenter les opérateurs relatifs à la subordination, et dans la section suivante nous décrirons la manière de spécifier qu'un objet ne peut pas être partagé.

La contrainte de subordination peut être spécifiée dans OBMS à l'aide de l'opérateur :+subordonné. Par exemple, pour représenter dans une base d'objets que les éléments d'interface des modules électroniques doivent être détruits lorsque le module qu'ils composent est détruit, les concepteurs peuvent utiliser l'expression

```
( :+subordonné %Module %Elément_Interface )
```

Dans ce cas, la classe Elément_Interface est l'objet subordonné et la classe Module est l'objet "superordonné". De plus, toutes les sous-classes de la classe Elément_Interface sont aussi des subordonnées de la classe Module, et les éléments d'interface (c'est-à-dire les instances) sont subordonnés des modules électroniques.

Il faut cependant noter que nous n'avons pas encore spécifié que les éléments d'interface ne peuvent pas être partagés. En d'autres termes, chaque instance de la classe Elément_Interface peut être utilisé pour constituer l'interface de plusieurs modules électroniques et, dans ce cas, la destruction de l'un de ces modules ne doit pas impliquer la destruction de ses éléments d'interface, car cela laisserait la base d'objets dans un état incohérent.

Pour éviter ces incohérences, dans OBMS la destruction d'objets est régie par les règles d'intégrité ci6 et cc8, mais aussi par les règles suivantes:

aa9)	La destruction d'un objet (classe ou instance) entraîne la destruction préalable de tous ses composants subordonnés qui n'ont pas un autre superordonné.
cc9)	Avant d'être détruite, toute classe lègue ses super-classes, ses caractéristiques et ses subordonnés, à ses sous-classes qui ont d'autres super-classes et qui, par conséquent, ne peuvent pas être détruites.

La règle cc9 permet de préserver, dans les bases d'objets, des connaissances qui sont représentées au niveau des classes et qui sont nécessaires pour maintenir l'intégrité sémantique de ces bases (*e.g.*, les contraintes représentées par les caractéristiques-f, les contraintes de subordination, etc.), même si ces classe sont détruites.

Du fait que dans un environnement de conception la contrainte de subordination peut être considérée inadéquate au bout d'un certain temps d'utilisation de la base d'objets (*Cf.* section 5 du chapitre 1), OBMS permet toujours de supprimer cette contrainte à l'aide de l'opérateur :-subordonné. Par exemple:

```
( :-subordonné %Module %Elément_Interface )
```

D'ailleurs, pour la représentation de connaissances concernant la subordination d'objets, les concepteurs peuvent utiliser tous les opérateurs "standard" du paradigme de modélisation caractéristique d'OBMS. Par exemple:

```
( :+superordonné %Elément_Interface %Module )

( :-superordonné %Elément_Interface %Module )
```

```
( :$subordonnés %Module )

( :$superordonnés %Elément_Interface )

( :$subordonnés %Elément_Interface )
```

3.10. Les assertions

Dans la modélisation d'applications complexes comme celles de la CAO, la portée d'un grand nombre de contraintes d'intégrité n'inclut pas uniquement une seule propriété d'un seul objet, mais plusieurs propriétés d'un même objet, une seule propriété de plusieurs objets, ou même plusieurs propriétés de plusieurs objets.

Dans ces cas, la technique de représentation de contraintes basée sur l'emploi de descripteurs s'avère inadéquate, car elle requiert la spécification de la même contrainte dans chacune des propriétés, et chacune des classes incluses dans la portée de cette contrainte.

Par exemple, pour spécifier que la date de création du "modèle simulable" d'un objet en cours de conception (OCC) doit toujours être postérieure à la date de création des modèles simulables de ses composantes, les concepteurs doivent inclure, aussi bien dans la propriété "modèle simulable" que dans la propriété "composants" de la classe OCC, le prédicat qui doit être utilisé par le système pour vérifier cette contrainte. Si le nom de ce prédicat est "dates-valides?", alors les expressions OBMS qui doivent être utilisées sont:

```
( :+caractéristique-f %OCC
     'modèle_simulable    '(dates-valides?) )

( :+caractéristique-f %OCC
     'composants          '(dates-valides?) )
```

L'inclusion du prédicat "dates-valides?" dans le descripteur des deux caractéristiques est indispensable si elles peuvent être modifiées indépendamment l'une de l'autre, et si le système vérifie la validité de cette opération en utilisant seulement le prédicat du descripteur de la propriété en question.

Très fréquemment les concepteurs ont aussi besoin spécifier des "clés" (*Cf.* section 6 du chapitre 1) pour certains objets. Dans ces cas, si une clé est

composée de plusieurs caractéristiques, le prédicat correspondant à cette contrainte doit encore être inclus dans le descripteur de toutes ces caractéristiques.

Dans OBMS, la spécification de ces contraintes peut être réalisée de manière prédicative, mais l'association entre les prédicats correspondants et les objets classe est représentée par une propriété prédéfinie des objets classe, appelée "assertions".

Pour spécifier, par exemple, que la clé des objets en cours de conception (c'est-à-dire les instances de la classe OCC) est constituée par leurs propriétés "nom" et "projet", les concepteurs peuvent utiliser l'expression suivante[1]:

```
( :+assertion  %OCC '(:clé? 'nom 'projet) )
```

La principale différence entre les prédicats des descripteurs et ceux des assertions est que les premiers reçoivent comme premier argument chacun des éléments constituant la valeur de la caractéristique en question, tandis que les prédicats des assertions reçoivent comme premier argument l'identificateur (surrogate) de la classe ayant ces assertions.

Par exemple, le prédicat ":est-un-Processeur?", utilisé dans la section 3.5 pour caractériser les ordinateurs, est évalué par OBMS lorsque l'ordinateur-1 est associé à la classe Ordinateur, en utilisant l'expression suivante:

```
( :est-un-Processeur?  %processeur-1 )
```

Par contre, le prédicat ":clé?" associé à la classe OCC est évalué en utilisant l'expression

```
( :clé?  %OCC 'nom 'projet )
```

[1]Dans OBMS, l'emploi d'attributs clés n'est pas antagoniste avec l'emploi de surrogates. Plusieurs clés peuvent être définies pour chaque classe d'objets, et les surrogates peuvent être vus comme la clé principale, car ils servent à identifier les objets *dans* la base d'objets, tandis que les clés définies par les utilisateurs servent en général à identifier les objets *hors de* la base d'objets.

Cela veut dire que tout ce qui peut être validé à l'aide de prédicats spécifiés pour les caractéristiques peut être validé au moyen des prédicats des assertions. Néanmoins, le coût de la validation (en temps d'exécution) dans le cas d'opérations de mise à jour est beaucoup plus grand lorsque l'on définit des contraintes à l'aide d'assertions au lieu de le faire à l'aide de descripteurs.

Effectivement, pour maintenir l'intégrité des bases d'objets le système évalue non seulement le prédicat de la caractéristique modifiée, mais aussi toutes les assertions associées à la classe (ou aux classes) de l'instance ayant cette caractéristique.

Ainsi, si une classe a par exemple 40 caractéristiques mais aucune assertion, et si l'une des ces caractéristiques est modifiée dans l'une de ses instances, le contrôle de l'intégrité n'implique que l'évaluation d'un prédicat: celui du descripteur de la caractéristique en question. Par contre, si la caractérisation de cette classe se fait en utilisant 40 assertions, toute mise à jour requiert l'évaluation de 40 prédicats.

En d'autres termes, la représentation de contraintes à l'aide d'assertions est beaucoup plus puissante mais beaucoup plus coûteuse que la représentation de contraintes à l'aide de descripteurs et, en conséquence, l'emploi d'assertions doit toujours être limité aux cas où une contrainte d'intégrité ne peut pas être représentée autrement.

L'une de ces situations est la spécification de contraintes concernant les propriétés prédéfinies. Dans la section précédente, par exemple, nous avons mentionné que dans certains cas les objets de la base ne doivent pas être partagés. Dans OBMS, un objet est partagé si sa propriété prédéfinie "agrégats" est multivaluée, et la valeur de cette propriété peut être connue à l'aide de l'opérateur :$agrégats.

Par exemple, d'après la figure 3.1, les instances de la classe Processeur (*e.g.*, le processeur-1) peuvent être partagées par plusieurs instances de la classe Ordinateur (*e.g.*, l'ordinateur-1 et l'ordinateur-55):

```
( :$agrégats %processeur-1 )

= %ordinateur-1, %ordinateur-55
```

En vue de spécifier que les instances de la classe processeur ne doivent pas être partagées par plusieurs objets, les concepteurs ne peuvent pas utiliser des descripteurs, car le "filtre" de la propriété agrégats est défini par le

système (*Cf.* section 3.7). Toutefois, les concepteurs peuvent représenter cette contrainte dans la base d'objets comme une assertion sur la classe Processeur, en définissant un prédicat qui prend la valeur VRAI uniquement lorsque la propriété "agrégats" des processeurs est monovaluée.

3.11. Les objets opérateur

Tout le long de ce chapitre nous avons montré que les opérateurs OBMS, qui peuvent être utilisés pour créer, modifier et supprimer des objets, vérifient le respect de toutes les contraintes d'intégrité connues par le système, et que ces connaissances peuvent être enrichies en associant aux objets classe des prédicats concernant les caractéristiques de leurs instances. Il faut cependant noter que ces prédicats définissent l'ensemble des états valides des objets, mais ils ne spécifient pas toutes les règles qui doivent être respectées pour changer l'état de ces objets.

Par exemple, au chapitre 1 nous avons mentionné que la compilation d'une description quelconque d'un objet en cours de conception (OCC) implique, dans certains environnements de CAO, la création d'un autre objet connu comme le "modèle simulable" de cet OCC, et que cette compilation peut être réalisée seulement si la description en question a subi, au préalable, un processus de vérification syntaxique et sémantique.

Un autre exemple a été donné dans la section 7, lors de notre discussion concernant les démons qui peuvent être utilisés pour enchaîner des opérations qui doivent se suivre en vue de maintenir cohérentes les bases d'objets, du point de vue sémantique.

Ni l'enchaînement d'opérations, ni la vérification des préconditions qui doivent exister pour effectuer ces opérations ne sont des actions réalisées par les opérateurs OBMS. Par conséquent, ces opérateurs ne sont pas suffisants pour représenter dans une base d'objets toutes les propriétés opératoires (c'est-à-dire le comportement) des objets des applications de CAO.

Néanmoins, les concepteurs peuvent définir de nouveaux **objets opérateur**, dont les propriétés structurelles dénotent aussi bien la procédure qui doit être utilisée pour manipuler d'autres objets de la même base d'objets, que les préconditions qui doivent être vérifiées avant l'exécution de cette procédure.

De plus, la procédure des objets opérateur peut être écrite dans les mêmes langages de programmation qui peuvent être utilisés pour définir les prédicats des descripteurs et des assertions, et peuvent contenir des

invocations à des opérateurs définis auparavant, y compris les opérateurs OBMS.

Par exemple, si la compilation de descriptions, mentionnée ci-dessus, doit être représentée dans une base d'objets comme un processus qui consiste en deux actions, créer une instance de la classe Modèle_Simulable, et associer cette instance à la description en question, à travers sa propriété "modèle", alors la procédure de l'opérateur correspondant peut être définie en Lisp de la manière suivante:

```
(de CompiDesc (description)
  (let (( modèle (:+instance)))  ;la création d'un objet instance.
    (:+instance %Modèle_Simulable modèle) ;son classification.
    (:+caractéristique description 'modèle modèle))) ;l'association.
```

Ceci étant, les concepteurs peuvent définir, par exemple dans la base BASE-1, l'objet opérateur correspondant, en utilisant l'expression OBMS

```
( :+opérateur %BASE-1
   'Compiler_Description 'CompiDesc)
```

et lui ajouter la précondition concernant la vérification des descriptions, en utilisant l'expression

```
( :+précondition %Compiler_Description
        '(:vérifiée? description) )
```

Après avoir réalisé ces opérations, la structure de l'opérateur Compiler_Description est la suivante:

```
OPERATEUR  Compiler_Description
    paramètres     : description
    préconditions  : (:vérifiée? description)
    code           : (let (( m (:+instance)))
                        (:+instance %Modèle_Simulable)
                        (:+caractéristique description 'modèle m)))
```

La possibilité de définir dans les bases d'objets des opérateurs tels que Compiler_Description présente plusieurs avantages. Premièrement, elle libère les concepteurs du besoin de connaître et de vérifier toutes les

conditions qui doivent être satisfaites pour maintenir l'intégrité sémantique des bases d'objets lors des opérations de mise à jour, car cette vérification est réalisée par OBMS.

Deuxièmement, elle permet de construire très facilement des interfaces contenant tous les opérateurs de haut niveau requis par les concepteurs et par les programmes d'application. Ainsi, au lieu d'utiliser des multiples opérateurs primitifs tels que :+instance ou :+caractéristique, les programmeurs d'applications et les concepteurs peuvent utiliser un nombre réduit d'opérateurs qui modélisent de manière précise le comportement des objets des applications.

Dans ce cas, le code des programmes d'application n'est pas seulement plus concis mais aussi plus fiable, du fait que tous les programmes d'application qui doivent effectuer une certaine action particulière dans la base d'objets peuvent utiliser la même procédure, c'est-à-dire celle de l'opérateur invoqué. Par ailleurs, la gamme des changements que l'on doit effectuer, lorsque l'on modifie la structure d'une classe d'objets, est limitée aux opérateurs qui manipulent les instances de cette classe.

3.12. Discussion

Dans ce chapitre nous avons présenté des concepts et des techniques qui permettent de représenter, dans une base d'objets, un nombre important de connaissances concernant les objets qui peuvent être créés et manipulés dans les environnements de conception, et nous avons montré que ces connaissances peuvent être utilisées non seulement par les concepteurs, mais aussi par OBMS pour maintenir l'intégrité sémantique des bases d'objets.

En particulier, nous avons montré, au travers d'exemples concrets, que les concepts et les techniques présentés dans ce chapitre constituent une infrastructure de représentation de connaissance cohérente et robuste, qui permet l'emploi d'un paradigme de modélisation simple mais adéquat pour satisfaire à la plupart des exigences mentionnées au chapitre 1.

En effet, l'identification d'objets au moyen de surrogates, la possibilité de définir et de manipuler des objets agrégats comme une seule unité, et d'organiser ces objets dans des hiérarchies d'agrégation et de généralisation, ainsi que la possibilité de représenter dans les bases d'objets une grande variété de contraintes d'intégrité à travers les descripteurs et les assertions, et le comportement des objets à l'aide d'opérateurs de haut niveau, satisfont toutes les exigences mentionnées dans les sections 2, 6, 7 et 8, du chapitre 1, et certaines exigences mentionnées dans les sections 4 et 9 du même chapitre.

De fait, les exigences décrites dans la section 3.4 ne sont pas totalement satisfaites car elles impliquent la possibilité de représenter et d'organiser plusieurs descriptions d'un objet dans la même base d'objets, mais les concepts et les techniques nécessaires pour satisfaire à ces demandes seront présentées au chapitre suivant. Nous devons donc faire dans cette section quelques commentaires concernant les exigences décrites dans la section 3.5 et celles de la section 3.9 qui n'ont pas été traitées dans le présent chapitre.

Dans OBMS les concepteurs peuvent toujours modifier le "schéma conceptuel" de la base d'objets, du fait que le système ne fait aucune distinction entre les spécifications concernant la structure des objets, les spécifications concernant les "occurrences" ayant cette structure, et les spécifications concernant les contraintes d'intégrité structurelles et opératoires qui doivent être satisfaites par ces "occurrences", en ce sens que toutes ces spécifications sont contenues dans la même base (c'est-à-dire elles ne sont pas des fichiers indépendants "codés" de manière différente), et représentées de manière uniforme: comme des objets.

En ce qui concerne le traitement flexible d'erreurs et de cas exceptionnels (*Cf.* section 3.9), au chapitre précédent nous avons mentionné que dans OBMS les concepteurs peuvent classifier un objet, et déclencher la vérification de toutes les contraintes associées, après qu'ils considèrent que cet objet se trouve dans un état cohérent, et dans le présent chapitre nous avons montré qu'en effet les opérations de création et de classification d'objets sont deux opérations différentes et indépendantes l'une de l'autre.

Par ailleurs, nous avons montré que les contraintes d'intégrité peuvent être définies, modifiées, associées aux objets de la base, et supprimées, pendant toute la durée de vie de la base d'objets, et que la définition de ces contraintes peut se faire à l'aide de prédicats qui peuvent être écrits dans des langages de programmation divers.

Néanmoins, nous n'avons pas inclus dans notre infrastructure de représentation de connaissances des concepts ou des techniques pour satisfaire à d'autres exigences importantes relatives au traitement d'erreurs et de cas exceptionnels. En particulier, la possibilité de désigner les réactions ("handlers") qui doivent être activés en cas de violations des contraintes d'intégrité.

Effectivement, dans OBMS le système avorte toujours le processus qui est à l'origine d'une violation des contraintes d'intégrité, et ce comportement n'est pas souhaitable dans la plupart des cas. Mais du fait qu'OBMS permet la définition d'opérateurs de haut niveau, les concepteurs peuvent utiliser un subterfuge que nous allons décrire ci-dessous, pour conclure ce chapitre.

Pour un traitement simple des erreurs et des cas exceptionnels, les utilisateurs d'OBMS peuvent définir un opérateur tel que :+réaction, dont la propriété "code" dénote une fonction qui accepte comme arguments un nombre quelconque de listes, dont la première contient le nom et les arguments de l'opérateur qui doit être invoqué si un erreur se produit pendant l'évaluation des autres listes, et ces autres listes peuvent être des appels à des opérateurs définis auparavant.

Par exemple, l'expression

```
( :+réaction
      (<opérateur-1> [<arguments>])
      (<opérateur-2> [<arguments>])
        . . .
      (<opérateur-N> [<arguments>]) )
```

produit l'évaluation des opérateurs opérateur-2 à opérateur-N et rend comme résultat celui de l'opérateur-N. Toutefois, si au cours de l'évaluation de ces opérateurs une erreur se produit, le système invoque l'opérateur-1 et, dans ce cas, le résultat de :+réaction est celui de cet opérateur-1.

L'opérateur :+réaction peut être défini en Le_Lisp [Chai 86] de la manière suivante:

```
(df :+réaction  (handler . opérateurs)
  (let ((résultat (catcherror t (eprogn opérateurs))))
    (if résultat (car résultat) (eval handler))))
```

Cet exemple montre encore que la possibilité de définir des descripteurs, des assertions et des opérations de haut niveau donne à OBMS (le système et l'infrastructure) un caractère extensible.

CHAPITRE 4

Objets génériques et objets représentatifs

4. Objets génériques et objets représentatifs

«Aussitôt que nous aurons découvert quelles sont les similitudes [entre certains objets, situations ou processus] pertinentes à la prédiction et au contrôle d'événement futurs, nous aurons tendance à considérer ces similitudes comme fondamentales et les différences comme insignifiantes. On pourra alors dire que nous avons développé un concept abstrait pour englober l'ensemble d'objets ou de situations en question.»

C. HOARE

4.1. Introduction

Lors de notre analyse des besoins concernant la gestion de données dans les environnements de CAO, au chapitre 1, nous avons vu que les objets en cours de conception sont en général des **objets génériques**, en ce sens qu'ils peuvent avoir plusieurs versions, alternatives, variantes, révisions, répliques, et/ou d'autres types d'**objets représentatifs**, et nous avons mentionné que la possibilité de faire coexister tous ces objets dans la même base, et de représenter leurs associations, est l'une des principales exigences des concepteurs.

Notre objectif dans le présente chapitre est de décrire les concepts et les techniques que nous avons développés pour satisfaire cette exigence, et qui ont été intégrés dans notre infrastructure de représentation de connaissances.

4.2. L'approche ontologique

Un circuit VLSI pouvant avoir plusieurs descriptions[1] (au niveau système, au niveau portes logiques, au niveau électrique,...), des descriptions pouvant avoir de nombreuses alternatives d'implémentation, et des alternatives pouvant avoir des versions multiples, sont des exemples d'objets génériques.

En CAO, en bureautique et en génie logiciel, les objets ayant plusieurs versions sont sans doute le type d'objets génériques dont la représentation dans des bases de données a été l'objet du plus grand nombre de recherches (*e.g.*, [BobG 80], [DaLW 84], [KaAC 86], [RieN 86] et [Zdon 84]). Il serait en fait plus adéquat de dire que c'est la représentation de versions qui a été

[1] Les différentes "descriptions" des systèmes VLSI sont parfois dénommées *représentations*, *vues* ou *perspectives*, mais dans cette thèse nous n'utilisons pas ces termes comme des synonymes.

l'objet de tels travaux, car très souvent l'existence d'objets génériques n'est représentée qu'implicitement par l'existence de leurs versions.

Dans ces cas, le seul moyen de modéliser les propriétés d'un objet générique est de les spécifier dans toutes les représentations de ses versions, en décrivant de cette manière le fait que chaque version doit avoir tous les attributs de son objet générique (*e.g.*, le nom du projet correspondant, le niveau de description et l'identificateur du fichier ou document contenant les contraintes fonctionnelles ou structurelles qui doivent être vérifiées), en plus de ses attributs propres (*e.g.*, date de création, concepteurs et composants).

Ainsi, chaque version d'un objet en cours de conception (OCC) est une sorte de "photographie" [Adib 81] ou reproduction d'un objet équivalent, défini auparavant, dans laquelle on a introduit un certain nombre de modifications qui distinguent la reproduction de l'objet "moule", afin de décrire les valeurs des propriétés structurelles ou état de l'objet générique (virtuel) à un instant donné.

Un autre point de vue est celui de D. Batory et W. Kim [BatK 85] qui montrent les avantages de représenter de manière explicite les objets génériques, et proposent une infrastructure de modélisation dans laquelle chaque version hérite des propriétés de son objet générique. Dans [McNB 83] les auteurs préconisent aussi la représentation explicite des objets génériques, et indiquent que toutes les versions d'un objet générique sont «des instances de son attribut "versions"», chacune ayant les caractéristiques de cet objet générique.

Tout de même, un fait important est que dans tous ces travaux il n'y a pas de consensus concernant les propriétés qui caractérisent les versions. En réalité la plupart de ces chercheurs reconnaissent l'existence de plusieurs types de "versions" ou objets représentatifs (*e.g.*, des révisions, des alternatives, des variantes, des versions paramétrées,...), distinguées en fonction d'un certain nombre de propriétés structurelles et opératoires, mais chaque groupe de recherche définit des propriétés distinctives différentes.

Dans [KaAC 86], par exemple, chaque version d'un OCC décrit, à un niveau donné (niveau système, niveau électrique,...), une configuration particulière, c'est-à-dire la manière dont cet objet peut être implémenté. A chaque niveau de description, les différentes versions sont appelées alternatives si leurs configurations sont équivalentes.

D'autre part, dans [RieN86], chaque version d'un OCC est une agrégation des différentes descriptions de cet objet, et ces descriptions sont des objets (génériques) qui peuvent avoir plusieurs configurations ou alternatives d'implémentation.

D'ailleurs, dans [McNB 83], chaque version est un objet (générique) qui peut avoir plusieurs alternatives d'implémentation, et qui décrit (à un niveau donné) toutes les caractéristiques partagées par ses alternatives. De plus, ces alternatives sont des objets (génériques) qui peuvent avoir un nombre quelconque de répliques (appelées aussi copies ou "instances"), qui sont utilisées pour définir les alternatives d'implémentation d'autres OCC et qui ont tous les attributs de leur alternative générique. Pour ces auteurs, une "configuration" n'est pas une alternative d'implémentation mais une agrégation des différentes descriptions équivalentes de l'OCC (comme les versions dans [RieN 86]).

Dans [BatK 85], chaque OCC est un objet générique qui peut avoir plusieurs descriptions; chaque description est aussi un objet générique qui peut avoir plusieurs configurations; chaque configuration est encore générique en ce sens qu'elles peuvent avoir des versions multiples; et chaque version est également générique puisqu'elles peuvent avoir un nombre quelconque de répliques, qui héritent de ses caractéristiques, de la même manière que dans [McNB 83].

Malgré toutes ces différences, il a été toujours proposé d'accroître le "bagage ontologique" des infrastructures de modélisation existantes, en incluant "version", "alternative", "réplique", etc. comme nouveaux termes (définis par le système) dans le domaine d'axiomatisation et, par ce moyen, éviter le besoin de définir leur sémantique dans la base de données ou dans les programmes d'application.

Les exemples ci-dessus montrent pourtant que ces termes peuvent être utilisés pour représenter des types différents d'objets, dans des applications distinctes. Par exemple, le terme "version" a été défini comme un synonyme de "description" dans [KaAC 86], comme un objet générique d'alternatives dans [McNB 83], comme une agrégation de descriptions dans [RieN 86], et comme un objet représentatif d'alternatives et générique de répliques dans [BatK 85], et ce terme pourrait être utilisé, dans d'autres applications, comme un synonyme de "variante" ou de "révision", ou même pour représenter un autre objet quelconque.

Par conséquent, l'inclusion de tels termes dans une infrastructure de représentation de connaissances restreint son domaine d'utilisation à celui des applications particulières où les objets génériques et représentatifs ont exactement la sémantique prédéfinie.

4.3. Les associations représentatif-de

Une analyse globale et inductive des exemples présentés dans la section précédente, met en relief que les associations entre les objets génériques et leurs représentatifs résultent d'un processus d'abstraction dans lequel une collection d'instances (c'est-à-dire les objets représentatifs) est généralisée pour constituer un seul objet de plus haut niveau (c'est-à-dire l'objet générique) qui définit toutes les caractéristiques partagées par chaque objet de la collection.

Les mêmes exemples montrent que ces associations, que nous appelons associations **représentatif-de**, organisent ces objets dans des hiérarchies avec les objets génériques placés au-dessus de leurs représentatifs (*voir* figures 4.1 et 4.2), et imposent sur ces objets certaines contraintes d'intégrité qui sont analogues à celles intrinsèques aux associations instance-de et spécialisation-de.

Par exemple, n'importe quel objet représentatif doit avoir toutes les caractéristiques de son objet générique et, en conséquence, il est considéré comme une instance de toutes les classes auxquelles appartient son objet générique. D'ailleurs, si un objet générique X est représentatif d'un objet plus générique Y, tous les représentatifs de l'objet X sont aussi considérés comme des représentatifs de l'objet Y. En accord avec les figures 2.4 et 4.1, par exemple, toutes les descriptions de l'objet vue-v1 sont des descriptions de l'objet ordinateur-1 et des instances de la classe Ordinateur.

Par conséquent, de la même façon que la majorité des infrastructures de représentation de connaissances permettent l'emploi du mécanisme de généralisation pour définir des classes et leurs associations avec leurs sous-classes (au lieu de présenter un ensemble particulier de classes et super-classes, spécifiques à une certaine application, telles que "Module", "UAL" et "Ordinateur"), nous avons choisi d'étendre l'utilisation et la puissance expressive de ce mécanisme, afin de pouvoir définir des objets génériques et leurs associations avec leurs représentatifs, au lieu d'inclure dans notre infrastructure de modélisation un ensemble inaltérable quelconque d'objets génériques ou représentatifs.

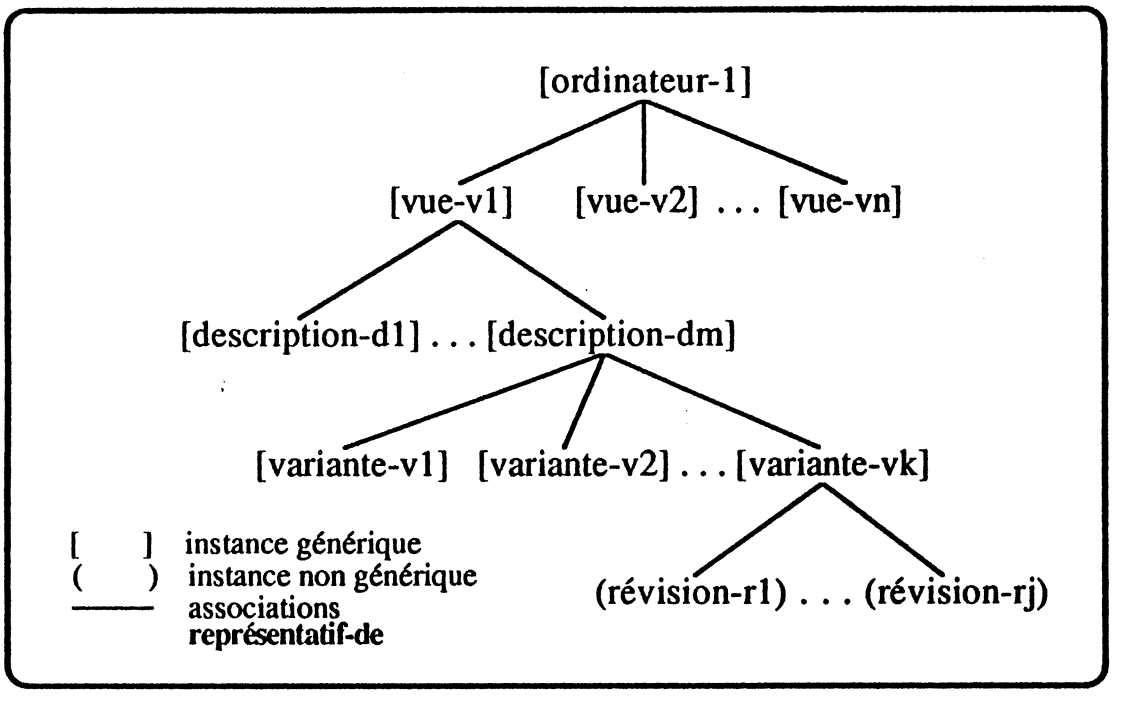


FIGURE 4.1

Associations représentatif-de entre certains objets d'une application hypothétique où chaque ordinateur peut avoir de multiples vues, chaque vue peut avoir plusieurs descriptions, chaque description peut avoir un nombre quelconque de variantes, et chaque variante peut avoir plusieurs révisions.

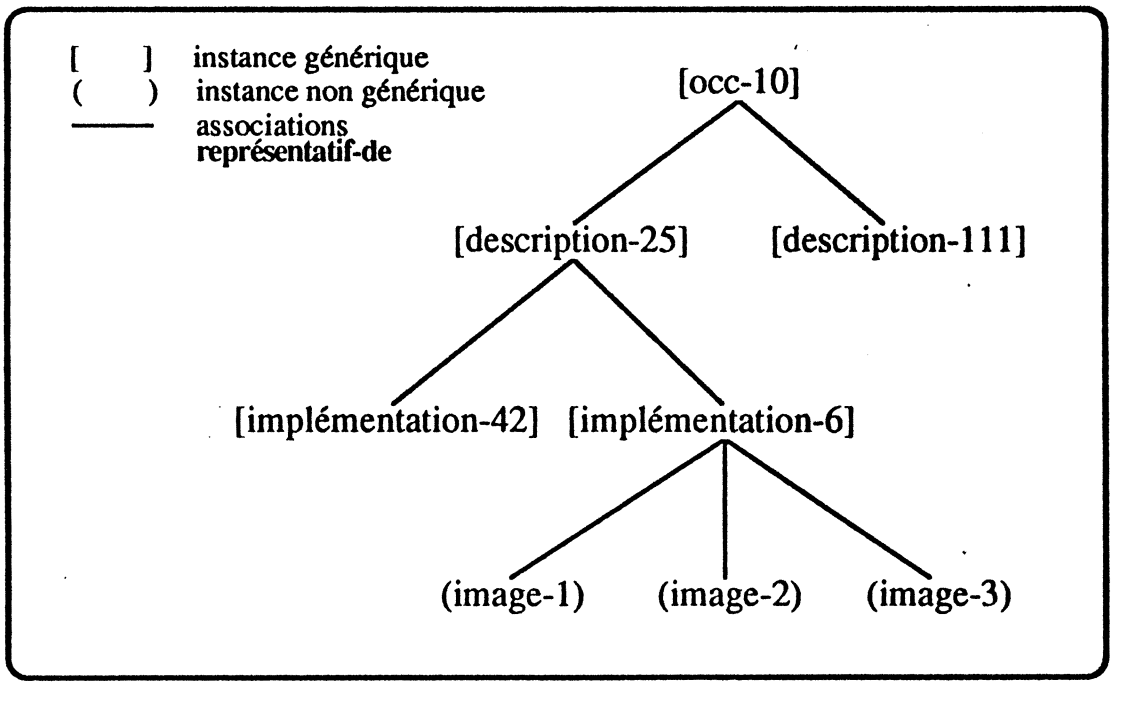


FIGURE 4.2

Associations représentatif-de entre certains objets d'une application hypothétique où chaque objet en cours de conception peut avoir plusieurs descriptions, chaque description peut avoir de multiples implémentations, et chaque implémentation peut avoir un nombre quelconque d'images.

4.4. Une nouvelle forme de généralisation

Une manière simple de permettre dans notre infrastructure de représentation de connaissances l'emploi d'une nouvelle forme de généralisation pour définir des objets génériques et leurs associations avec leurs représentatifs, tout en restant fidèles au paradigme de modélisation décrit au chapitre précédent, consiste à prédéfinir dans cette infrastructure une propriété des objets instance, que nous allons appeler la propriété "générique" et qui représente les associations représentatif-de, de la même manière que nous avons prédéfini la propriété "classes" pour représenter les associations "instance-de".

Cette propriété prédéfinie va permettre aux utilisateurs la possibilité de spécifier, par exemple, le fait que l'objet description-25 est représentatif de l'objet occ-10 (*Cf.* figure 4.2), par la simple inclusion de l'identificateur de cet objet en cours de conception, dans la propriété "générique" de cette description:

```
( :+générique  %description-25 %occ-10 )
```

Au niveau conceptuel, ce type d'association implique aussi bien la propriété "générique" des objets représentatifs que la propriété "représentatifs" des objets génériques. Par conséquent, une manière alternative d'établir l'association précédente est:

```
( :+représentatif  %occ-10 %description-25 )
```

Dans les environnements qui ont plusieurs classes d'objets génériques et d'objets représentatifs, les concepteurs ont aussi besoin de spécifier (contraindre) la classe d'instances qui peuvent être des représentatifs de chaque classe d'objets génériques.

Par exemple, en accord avec la figure 4.2, les représentatifs des objets en cours de conception ne peuvent être que des descriptions, les représentatifs des descriptions ne peuvent être que des implémentations, et les représentatifs des implémentations ne peuvent être que des images.

Ces contraintes ne sont pas universelles mais particulières à une certaine application et, par conséquent, elles ne sont pas prédéfinies dans notre infrastructure de représentation de connaissances. Cependant, elles peuvent

être spécifiées d'une manière similaire à celle utilisée au chapitre précédent pour caractériser les objets (*Cf.* section 3.5), car chaque classe a une propriété prédéfinie dont le nom est "représentatifs" et la valeur indique, lorsqu'elle n'est pas nulle, que

a) toutes les instances de cette classe sont des objets génériques,
b) les représentatifs de ces instances doivent appartenir à la classe spécifiée par cette valeur non nulle.

Ainsi, pour représenter dans une base d'objets le fait que les représentatifs des objets en cours de conception doivent être des instances de la classe Description, les concepteurs peuvent utiliser l'expression suivante:

```
( :+représentatif %OCC %Description )
```

Par ailleurs, étant donné que l'existence de la propriété "représentatifs" dans les classes des objets génériques implique aussi la propriété "générique" dans les classes des objets représentatifs, cette propriété a été prédéfinie et, ainsi, la contrainte précédente peut aussi être spécifiée en utilisant l'expression

```
( :+générique %Description %OCC )
```

De plus, en accord avec le paradigme de modélisation présenté au chapitre 3, dans OBMS la suppression et l'analyse des associations représentées par les propriétés "générique" et "représentatifs", des objets classe et des objets instance, peuvent être réalisées au moyen des opérateurs :-générique, :-représentatif, :$génériques et :$représentatifs.

Pour supprimer par exemple l'association entre l'ordinateur-1 et son représentatif vue-v1, illustrée dans la figure 4.1, les concepteurs peuvent utiliser les expressions

```
( :-représentatif %ordinateur-1 %vue-v1 )

ou     ( :-générique %vue-v1 )
```

Egalement, pour indiquer au système que les instances de la classe Révision ne sont plus des représentatifs de la classe Variante (*Cf.* figure 4.1), les expressions OBMS qui peuvent être utilisées sont:

(**:-représentatif** %Variante %Révision)

ou (**:-générique** %Révision)

Un fait important concernant les opérateurs polymorphes :+générique, :-générique, :+représentatif et :-représentatif, est qu'ils vérifient le respect de toutes les contraintes d'intégrité intrinsèques aux associations représentatif-de, afin de maintenir sémantiquement cohérentes les bases d'objets. Dans notre infrastructure de représentation de connaissances, ces contraintes sont les suivantes:

gr1)	L'association représentatif-de n'est définie qu'entre deux objets instance ou entre deux objets classe.
gr2)	Si un objet X est représentatif d'un objet Y, alors Y est le générique de X.
gr3)	Tout objet (classe ou instance) peut avoir un nombre quelconque de représentatifs, mais aucun objet ne peut avoir plus d'un objet générique immédiat (*Cf.* règle gr8).
gr4)	La **classe représentative** d'une instance X est la classe définie par le descripteur de la propriété "représentatifs" de la classe de X, c'est-à-dire la classe à laquelle doivent appartenir tous les représentatifs de X.
gr5)	Si A est représentatif de B, et si A et B sont des objets classe, alors les instances de A ne peuvent avoir comme objets génériques que des instances de B. Autrement dit, une instance X ne peut être définie comme représentative d'une instance Y que si X appartient à la classe représentative de Y ou si celle-ci est nulle.
gr6)	Aucun objet ne peut être associé en même temps à deux classes définissant des représentatifs différents.
gr7)	Les objets représentatifs héritent de tous les attributs de leur objet générique et, en conséquence, *a)* ils ne peuvent ni redéfinir ni spécialiser ces attributs, mais ils peuvent avoir des attributs additionnels, et *b)* chacun de ces objets représentatifs est aussi une instance des classes auxquelles appartient son objet générique. En d'autres termes, les représentatifs de toute instance sont aussi des instances de toutes ses classes. Ceci implique que tout objet représentatif doit respecter aussi bien les contraintes d'intégrité définies par ses classes, que les contraintes d'intégrité définies par les classes de son objet générique. (Voir aussi la règle ci5.)

gr8)	Si un objet X est représentatif d'un objet Y, tous les représentatifs de X sont aussi des représentatifs de Y (transitivité).
gr9)	Aucun objet X ne peut être représentatif d'un objet Y si celui-ci est représentatif de X (asymétrie).
gr10)	Tout objet est représentatif et générique de lui-même (réflexivité).
gr11)	Aucun objet X ne peut être représentatif d'un objet Y si X et Y ont une classe commune.
gr12)	Aucune classe A ne peut être représentative d'une classe B si celle-ci est sous-classe ou super-classe de A, si le générique dont A hérité de ces super-classes n'est pas nul ou une super-classe (ancêtre) de A, ou si la définition d'attributs de A redéfinit ou raffine l'une des définitions d'attributs de B.
gr13)	Aucun objet ne peut être détruit s'il est représentatif d'un autre objet. (Voir aussi la règle gr14).
gr14)	La destruction d'un objet générique (classe ou instance) entraîne la destruction préalable de tous ses représentatifs. (Voir la règle aa8.)

De plus, l'existence d'objets génériques et d'objets représentatifs dans une base d'objets impose deux contraintes additionnelles sur les associations spécialisation-de entre les objets classe. Ces contraintes sont:

cc10)	Aucune classe A ne peut être spécialisation d'une classe B si celle-ci est une classe générique ou représentative de la classe A, c'est-à-dire si les instances de B ont été définies comme des objets génériques ou représentatifs des instances de la classe A.
cc11)	Aucune classe A ne peut être définie comme spécialisation d'une classe B, si la classe générique de la classe B n'est pas égale à, ou spécialisation de, la classe générique de la classe A.

En ce qui concerne les opérateurs :$génériques et :$représentatifs, dans l'implémentation actuelle d'OBMS ils peuvent être employés avec un argument booléen facultatif dont la valeur par défaut est VRAI, pour indiquer que la recherche d'objets génériques ou d'objets représentatifs doit se faire en utilisant des mécanismes qui sont basés sur les règles d'intégrité intrinsèques aux associations entre ces objets (*Cf.* règles gr1 à gr6, et gr8 à gr12).

Par exemple, en accord avec la figure 4.2, lorsque l'on s'intéresse à connaître les représentatifs de l'objet description-25 et les objets génériques de l'objet image-2, les résultats de ces opérateurs sont les suivants:

(**:$représentatifs** %description-25)

= %implémentation-42, %implémentation-6, %image-1, %image-2, %image-3

(**:$représentatifs** %description-25 FAUX)

= %implémentation-42, %implémentation-6

(**:$génériques** %image-2)

= %implémentation-6, %description-25, %occ-10

(**:$génériques** %image-2 FAUX)

= %implémentation-6

En fait, les opérateurs :$génériques et :$représentatifs ne sont pas les seuls opérateurs OBMS qui acceptent des arguments facultatifs, ou qui utilisent par défaut des mécanismes d'inférence. Les opérateurs :$classes, :$instances, :$subordonnés, :$agrégats et :$assertions, sont aussi des exemples d'opérateurs qui acceptent un argument additionnel qui peut être utilisé pour inhiber l'emploi automatique de mécanismes d'inférence.

4.5. Exemple de modélisation

Afin de mieux illustrer la manière dont la nouvelle forme de généralisation peut être utilisée pour la modélisation d'applications de CAO, nous allons analyser dans cette section certaines opérations qui peuvent être réalisées en vue de représenter, dans une base d'objets, les objets correspondant à l'application décrite dans la figure 3.2.

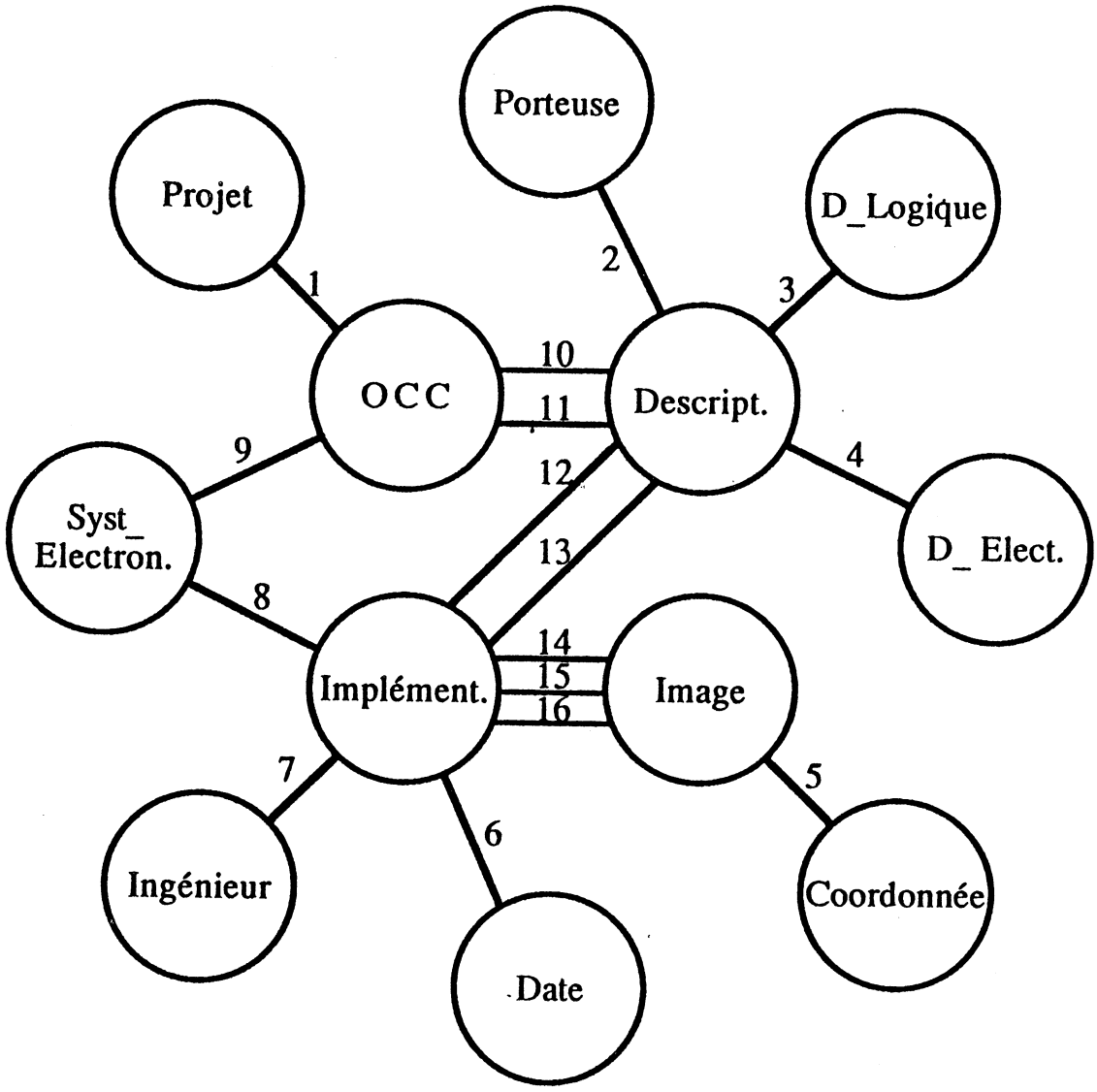
Cette application a des objets en cours de conception qui peuvent avoir plusieurs descriptions, des descriptions qui peuvent avoir plusieurs implémentations, et des implémentations qui peuvent avoir plusieurs images. Nous supposons donc que ces objets appartiennent aux classes Objet_en_Cours_de_Conception (OCC), Description, Implémentation et Image, et que les caractéristiques de ces classes sont celles illustrées dans la figure 4.3. Les différentes associations représentées par ces caractéristiques sont illustrées dans la figure 4.4.

En accord avec ces figures, toute instance de la classe Description a les attributs "interface" et "niveau", mais aussi l'attribut "projet" dont la classe

Description hérite de la classe OCC. Les instances de la classe Implémentation ont tous ces attributs et, en plus, elles ont les attributs "concepteurs", "date_création" et "composants". Les instances de la classe Images ont tous les attributs de la classe Implémentation, et en plus elles ont les attributs "système" et "emplacement".

CLASSE OCC		
super-classes	:	Système_Electronique
représentatifs	:	Description
instances	:	occ-10, occ-115, occ-55,...
projet	:	instance de Projet (1 1)
CLASSE Description		
sous-classes	:	Descrip_Logique, Descrip_Electrique,...
générique	:	OCC
représentatifs	:	Implémentation
instances	:	description-25, description-111,...
interface	:	instances de Porteuse (1 *)
niveau	:	système, logique, électrique,... (1 1)
CLASSE Implémentation		
générique	:	Description
représentatifs	:	Image
subordonnés	:	Image
instances	:	implémentation-42, implémentation-6...
concepteurs	:	instances d'Ingénieur
date_création	:	instance de Date (0 1)
composants	:	instances de Système_Electronique <acyclique>
CLASSE Image		
générique	:	Implémentation
superordonnés	:	Implémentation
instances	:	image-2, image-1, image-3, image-111
système	:	instance d'Implémentation (1 1)
emplacement	:	instance de Coordonnées (0 1)
assertions	:	<non partagée>

FIGURE 4.3
Les classes des objets génériques et représentatifs.



1.	projet de	O C C ---------->	Projet
2.	interface de.....	Description ---------->	Porteuse
3.	sous-classe de	Description ---------->	Description Logique
4.	sous-classe de	Description ---------->	Description Electrique
5.	emplacement de	Image ---------->	Coordonnées
6.	date de création de	Implémentation ---------->	Date
7.	concepteurs de	Implémentation ---------->	Ingénieur
8.	composants de	Implémentation ---------->	Système Electronique
9.	super-classe de	O C C ---------->	Système Electronique
10.	représentatifs de	O C C ---------->	Description
11.	générique de	Description ---------->	O C C
12.	représentatifs de	Description ---------->	Implémentation
13.	générique de	Implémentation ---------->	Description
14.	représentatif de	Implémentation ---------->	Image
15.	générique de	Image ---------->	Implémentation
16.	système de	Image ---------->	Implémentation

FIGURE 4.4
Associations entre les objets de l'application.

Toute instance de la classe Image a donc huit attributs:

a) système = instance de la classe Implémentation dont cette image est utilisée comme composant; les cardinalités de la propriété sont (1 1), c'est-à-dire qu'une image doit forcément être composant d'un et seulement d'un objet en cours de conception.
b) emplacement = coordonnées de son emplacement dans l'objet (implémentation) que cette image compose; les cardinalités de la propriété sont (0 1).
c) concepteurs = ingénieurs qui ont conçu cette image, vue comme une implémentation d'un objet en cours de conception.
d) date_création = date où cette image a été créée, vue comme une implémentation; les cardinalités son (O 1).
e) composants = composantes de cette image, vue comme une implémentation. D'après la classe Implémentation, ces composants doivent être des systèmes électroniques. Ainsi, les images peuvent être composées d'images d'autres objets en cours de conception, car en accord avec la règle gr7 toute image est aussi instance des classes Implémentation, Description et OCC, et tout OCC est un système électronique (voir la propriété super-classes de la classe OCC).
f) interface = porteuses de cette image, vue comme une description d'un objet en cours de conception; les cardinalités indiquent que l'interface doit avoir au moins une porteuse.
g) niveau = niveau dans lequel est décrite cette image (c'est-à-dire niveau système, niveau portes logiques, etc.), vue comme une description d'un objet en cours de conception; les cardinalités sont (1 1).
h) projet = projet auquel appartient cette image, vue comme un objet en cours de conception; les cardinalités sont (1 1).

De plus, la propriété "composants" de la classe Implémentation contient une contrainte (<acyclique>, dans la figure 4.3) pour éviter qu'un objet soit défini comme composant de lui-même, et la propriété "assertions" de la classe Image contient une contrainte (<non partagée>, dans la figure 4.3) pour éviter qu'une image participe à la composition de plusieurs objets. Par ailleurs, les images sont définies comme des subordonnées des implémentations, pour indiquer que la destruction de toute implémentation doit entraîner la destruction des images que le composent.

Pour représenter dans la base d'objets BASE-1 ces connaissances concernant les objets de l'application que nous sommes en train d'analyser, nous devons d'abord créer les classes:

```
%OCC            <--   ( :+classe %BASE-1 OCC )

%Description    <--   ( :+classe %BASE-1 Description )

%Implémentation <--   ( :+classe %BASE-1 Implémentation )

%Image          <--   ( :+classe %BASE-1 Image )
```

et ainsi de suite pour toutes les autres classes illustrées dans la figure 4.4 (e.g., Projet, Porteuse, Date, etc.).

Ensuite nous définissons leurs propriétés, pour représenter leurs différentes associations illustrées dans la figure 4.4. L'ordre de ces définitions (*e.g.*, d'abord leurs sous-classes ou leurs super-classes, puis leurs caractéristiques, etc.) n'est pas important. En vue de définir, par exemple, les propriétés de la classe Image nous pouvons utiliser les expressions OBMS suivantes:

```
( :+générique  %Image %Implémentation )

( :+superordonné  %Image %Implémentation )

( :+assertion  %Image ':non-partagé? )

( :+caractéristique-f  %Image
        système ':est-un-Implémentation? )

( :+caractéristique-f  %Image
        emplacement ':est-un-Coordonnées? )
```

Pour spécifier ces propriétés, les objets que l'on veut associer doivent évidement être connus du système. Nous ne pouvons pas, par exemple, spécifier que les images sont représentatives des implémentations si l'un de ces objets n'a pas été défini auparavant. Il en est de même pour les prédicats utilisés dans ces définitions. Pour spécifier par exemple les caractéristiques "système" et "emplacement", les fonctions dénotées par les prédicats ":est-un-Implémentation?" et ":est-un-Coordonnées?" doivent être connues du système.

Les prédicats de la forme :est-un-XX?, qui vérifient l'appartenance d'une instance à une classe dénotée par le suffixe XX, sont créés automatiquement

par le système lors de la création d'objets classe, en utilisant l'opérateur :est-un? qui accepte comme argument l'identificateur d'une instance et l'identificateur d'une classe, et vérifie que cette instance est associée à cette classe, ou à l'une des sous-classes ou des classes représentatives de cette classe, en analysant la propriété "instances" de ces classes. Ainsi, l'expression

```
( :est-un-Implémentation?  %implémentation-42 )
```

est tout à fait équivalente à l'expression

```
( :est-un?  %implémentation-42 %Implémentation )
```

à l'expression

```
( :est-un?  %implémentation-42
     ( :classe?  Implémentation ) )
```

et même à l'expression

```
( :est-un-OCC?  %implémentation-42 )
```

Par ailleurs, dans OBMS les caractéristiques filtrées des objets classe peuvent être spécifiées lors de la création de ces objets:

```
( :+classe  %BASE-1 Description
     'interface   ':est-un-Porteuse?
     'niveau      '(:élément-de? '(système logique électrique ...)) )
```

et la création automatique des prédicats "est-un-XX?" permet des spécifications récursives au moment de la création des classes. Par exemple, pour représenter dans la base BASE-1 le fait que les circuits sont composés de circuits, les concepteurs peuvent utiliser l'expression

```
( :+classe  %BASE-1 'Circuit
      'composants ':est-un-Circuit? )
```

au lieu de l'expression

```
%Circuit <-- ( :+classe  %BASE-1 'Circuit )
```

suivie de l'expression

```
( :+caractéristique-f  %Circuit
      'composants ':est-un-Circuit? )
```

Une autre solution possible au problème des spécifications récursives, qui a été proposée par D. Rieu et G. Nguyen [RieN 86], consiste à définir au préalable une super-classe de la classe récursive. Par exemple, pour définir la classe Circuit, on crée d'abord une classe telle que Super_Circuit, et ensuite on définit la classe Circuit comme une sous-classe de Super_Circuit, avec la caractéristique "composants" constituée de super-circuits. Cette démarche peut être réalisée dans OBMS en utilisant les expressions suivantes:

```
%Super_Circuit <-- (:+classe  %BASE-1 'Super_Circuit)

( :+sous-classe  %Super_Circuit
      (:+classe  %BASE-1 'Circuit
            'composants ':est-un-Super_Circuit?) )
```

ou même l'expression

```
( :+sous-classe  (:+classe  %BASE-1 'Super_Circuit)
      (:+classe  %BASE-1 'Circuit
            'composants ':est-un-Super_Circuit?) )
```

Cette solution (inutile dans OBMS) est tout à fait valide, car elle modélise bien le fait que les circuits peuvent être composés de circuits, du fait que toute instance de la classe Circuit est aussi une instance de la classe Super_Circuit. Néanmoins, en vue d'assurer l'intégrité sémantique de la base d'objet, cette solution requiert la spécification de contraintes additionnelles soit pour éviter que les composants des circuits appartiennent à d'autres sous-classes de la classe Super_Circuit, soit pour éviter que d'autres sous-classes de cette super-classe soient définies.

Une autre solution qui peut être utilisée dans OBMS, pour spécifier les caractéristiques des classes récursives au moment de leur création, même si la fonctionnalité concernant la génération automatique des prédicats est-un-XX? n'est pas implémentée, consiste à employer ensemble les prédicats "est-un?" et "classe?", avec le nom de la classe récursive:

```
( :+classe  %BASE-1 'Circuit
        'composants '(:est-un? (:classe? 'Circuit)) )
```

Cette solution est aussi valide parce que le prédicat :est-un? n'est évalué qu'au moment de l'instanciation de la classe Circuit, ou de la modification de la propriété "composants" de l'une de ses instances. D'autre part, la composition de fonctions est nécessaire, car le prédicat :est-un? requiert le surrogate de la classe, et celui-ci n'a pas encore été créé; cependant, il pourra être retrouvé par le prédicat :classe? au moment où le prédicat :est-un? est évalué.

Dans la plupart des cas, ces spécifications récursives, ainsi que les spécifications cycliques ne sont en réalité valables qu'au niveau des objets classe. Par exemple, la spécification de la propriété "composants" de la classe Implémentation est une spécification que nous appelons cyclique, car elle indique indirectement que les composants des implémentations peuvent être des implémentations, du fait que celles-ci sont aussi des systèmes électroniques.

Toutefois, dans la figure 4.3 la propriété "composants" de la classe Implémentations contient la contrainte "acyclique" pour indiquer que le système ne doit pas permettre l'existence d'une implémentation spécifique (c'est-à-dire une instance de la classe Implémentation) dont l'un des composants, ou des sous-composants, ou des composants des sous-composants, etc., est cette instance.

Ceci étant, dans la spécification de la caractéristique "composants" de la classe Implémentation, nous devons utiliser un prédicat tel que ":composant-valide?", dont la valeur est VRAI uniquement lorsque la

valeur de la propriété "composants" des implémentations est constitué par des objets qui respectent les contraintes "instance de Système_Electronique" et "<acyclique>", indiquées dans la figure 4.3:

```
( :+caractéristique-f  %Implémentation
      'composants ':composant-valide? )
```

La fonction dénotée par le prédicat :composant-valide? pourrait être écrite en Lisp, en utilisant des opérateurs prédéfinis par OBMS, de la manière suivante[1]:

```
(de composant-valide? (composant)
   (and (:est-un-Système_Electronique? composant)
            (not (:cyclique? (:lui-même) 'composants composant))) )
```

Pour compléter la définition des propriétés des classes illustrées dans la figure 4.3, nous devons spécifier aussi les cardinalités minimale et maximale des caractéristiques des classes (*e.g.*, "projet" de la classe OCC, et "date_création", de la classe Implémentation), et l'assertion de la classe Image, en utilisant les concepts et les techniques décrites dans les sections 6 et 10 du chapitre précédent:

```
( :cardinalités  %OCC 'projet '(1 1) )

( :cardinalités  %Description 'interface '(1 *) )

( :cardinalités  %Description 'niveau '(1 1) )
      .
      .
      .
( :+assertion  %Image ':non-partagée? )
```

Ceci étant, la création, la suppression et en général la manipulation des objets de l'application peuvent être représentées dans la base d'objets par la création, la suppression et la modification des instances de ces classes. Par exemple, la création d'une nouvelle description d'un objet en cours de conception, c'est-à-dire la création d'un représentatif d'un objet générique, peut être représentée dans la base d'objets comme un processus qui consiste à créer d'abord un objet instance, ensuite classifier cette instance dans la

[1]La description de la structure et du comportement des opérateurs :cyclique et :lui-même a été incluse dans la section 5 de l'annexe B.

classe Description, et finalement spécifier que cette instance est représentative d'un objet en cours de conception.

Dans ce cas l'ordre des actions réalisées est important, car en accord avec la règle gr4, que nous avons définie dans la section 4, les objets représentatifs d'un objet quelconque (dans ce cas l'objet en cours de conception) doivent appartenir à la classe représentative de cette objet (dans ce cas la classe Description).

Evidement, les concepteurs ne doivent pas tous être concernés par tous les détails relatifs à l'organisation et aux règles d'intégrité de la base d'objets. En général, l'organisation des bases en termes de classes et de sous-classes, d'objets génériques et d'objets représentatifs, etc., ainsi que l'ensemble des règles d'intégrité inhérentes à cette organisation, doivent être seulement connus des concepteurs de la base et de certains programmeurs d'applications.

Autrement dit, la modélisation d'applications de CAO doit inclure aussi la définition des opérateurs de haut niveau tels que "Créer_Description" ou "Sélectionner_Variante", requises par les concepteurs afin qu'ils n'aient besoin ni d'utiliser de multiples opérateurs primitives, ni de vérifier toutes les contraintes intrinsèques aux diverses associations entre objets.

Dans le chapitre précédent nous avons déjà montré la manière dont ces opérateurs peuvent être représentés dans une base d'objets. Néanmoins, pour conclure cette section, nous allons présenter ici un autre exemple relatif aux objets génériques et représentatifs.

D'après notre analyse concernant la création de descriptions d'objets en cours de conception, l'opérateur "Créer_Description", mentionné ci-dessus, doit être défini comme un opérateur OBMS dont la procédure de son propriété "code" réalise trois opérations primitives:

a) créer un objet instance dans la base d'objets (par exemple dans la base BASE-1)

```
%description <-- ( :+instance %BASE-1 <attributs>)
```

b) classifier cette instance dans la classe Description

```
( :+instance  %Description %description )
```

c) définir cette instance comme un représentatif de l'objet en cours de conception que l'on veut décrire

(**:+représentatif** <objet générique> %description)

Cette procédure doit donc recevoir comme arguments l'identificateur de l'objet générique, et les attributs de la description que l'on veut créer, c'est-à-dire son projet, son niveau et les porteuses qui constituent son interface. En Lisp, cette procédure pourrait être définie de la manière suivante:

```
(de CréerDesc (générique projet niveau . porteuses)
  (let ((%description
          (:+instance %BASE-1
                projet      projet
                niveau      niveau
                interface   porteuses)))
    ( :+instance %Description %description )
    ( :+représentatif <objet générique> %description )))
```

Ceci étant, la création de l'opérateur "Créer_Description" peut être réalisée à l'aide de l'expression

(**:+opérateur** %BASE-1 'Créer_Description 'CréerDesc)

Après cette spécification, la structure de l'opérateur Créer_Description est celle illustrée dans la figure 4.5.

OPERATEUR Créer_Description		
paramètres	:	générique, projet, niveau, porteuses
préconditions	:	( )
code	:	<le code de la fonction CréerDesc>

FIGURE 4.5

Une contrainte que l'on trouve très fréquemment dans les environnements de conception de systèmes électroniques, est que les différentes descriptions de chaque objet en cours de conception doivent avoir des interfaces équivalentes (*Cf.* [BatK 85], [McNB 83] et [RieN 86]), c'est-à-dire des

interfaces qui décrivent la même propriété mais à des niveaux d'abstraction différents (*e.g*, au niveau électrique, au niveau logique, etc.).

Dans notre infrastructure de représentation de connaissances, les algorithmes utilisés pour vérifier cette équivalence entre les interfaces peuvent constituer le prédicat que OBMS doit utiliser pour maintenir l'intégrité sémantique de la base d'objets, et ce prédicat peut être le filtre de la caractéristique "interface" de la classe Description, l'un des prédicats de la propriété "assertions" de cette classe, ou l'une des préconditions d'exécution de l'opérateur Créer_Description.

L'inconvénient de représenter cette contrainte dans la base d'objets comme une assertion sur la classe Description, est que ce prédicat est évalué chaque fois que les instances de cette classes sont modifiées, même si les modifications n'incluent pas des changements à la valeur de leur propriété "interface" (*Cf.* section 3.10). D'autre part, si la valeur de cette propriété n'est jamais modifiée pendant toute la durée de vie de la description en question, il est plus adéquat de vérifier la contrainte d'équivalence lors de la création de cette description et non lors de son association à la classe Description.

Si l'on décide de représenter cette contrainte comme une précondition de l'opérateur Créer_Description, si le prédicat qui dénote la fonction qui implémente les algorithmes de vérification est "interface-valide?", et si cette fonction requiert deux arguments qui sont l'identificateur de l'objet générique et la liste de porteuses de l'interface de la description que l'on veut créer, alors il peut être associé à l'opérateur Créer_Description en utilisant l'expression suivante:

```
( :+précondition  %Créer_Description
        '(:interface-valide? générique porteuses) )
```

4.6. Discussion

Dans ce chapitre nous avons présenté une extension du mécanisme de généralisation, qui permet de représenter dans des bases d'objets aussi bien les objets génériques créés et manipulés dans les environnements de CAO, que leurs associations avec leurs multiples objets représentatifs.

Cette extension a été réalisée en incluant, dans notre infrastructure de représentation de connaissances, des concepts et des techniques permettant l'emploi d'une nouvelle forme de généralisation qui organise les objets

génériques, et les objets représentatifs, en termes d'une association primitive que nous avons appelée représentatif-de.

Nous avons montré aussi que cette nouvelle forme de généralisation définit de manière adéquate la sémantique des objets génériques et des objets représentatifs, car les contraintes d'intégrité intrinsèques à l'association que cette forme de généralisation établit entre les objets, c'est-à-dire l'association représentatif-de, sont aussi des contraintes inhérentes aux associations que les concepteurs établissent entre les objets génériques et leurs représentatifs, dans les environnements de CAO.

Les analogies entre ces contraintes et celles des associations instance-de et spécialisation-de ont été mises en relief, et nous avons montré qu'elles résultent du fait que toutes ces associations sont établies par l'emploi de diverses formes de généralisation. D'autre part, la description de ces contraintes a mis aussi en évidence les différences entre ces types d'associations.

Par exemple, au chapitre 2 nous avons mentionné que dans OBMS toute instance peut appartenir à plusieurs classes, que les classes peuvent avoir plusieurs super-classes, et que la suppression d'une classe implique uniquement la suppression de ses instances et de ses sous-classes qui ne sont pas associées à une autre classe (*Cf.* règles ci3, ci6, cc3 et cc8). Par contre, dans le cas des associations représentatif-de, aucun objet ne peut être défini comme représentatif de deux objets différents (règle gr3), et la suppression d'un objet générique implique toujours la suppression de ses objets représentatifs (règle gr14).

Ces contraintes sur les associations représentatif-de ont été incluses dans OBMS, car nous n'avons trouvé aucune application où le comportement des objets génériques et de leurs représentatifs soit régie autrement (*e.g.*, une application où un objet puisse être version de deux objets en cours de conception différents).

Nous avons mentionné aussi que les instances et les sous-classes peuvent avoir des caractéristiques additionnelles à celles définies par les classes auxquelles ces objets sont associés, mais que cela n'est aucunement nécessaire. Toutefois, dans le cas d'objets représentatifs ces caractéristiques additionnelles semblent être "obligatoires", car elles sont toujours utilisées.

Nous n'avons pourtant pas inclus cette dernière contrainte dans notre infrastructure, car certaines applications pourraient avoir besoin de regrouper les différents représentatifs d'un objet générique en fonction de critères "externes" (c'est-à-dire non représentés dans la base d'objets), de la même façon qu'elles ont souvent besoin de définir des sous-classes qui n'ont pas forcément des propriétés additionnelles (*Cf.* section 5 du chapitre 2).

Enfin, nous considérons que la possibilité d'utiliser une nouvelle forme de généralisation pour définir des objets génériques, et pour distinguer leurs associations avec leurs objets représentatifs de leurs associations avec d'autres objets, constitue un outil conceptuel important qui simplifie la conception, la gestion et le contrôle de l'intégrité sémantique des bases d'objets pour la CAO.

CONCLUSIONS

Conclusions

L'emploi de bases de données est fondamental dans l'intégration des systèmes de CAO. L'objectif de cette thèse a été de présenter un ensemble cohérent et robuste de concepts et de techniques pour la conception, l'implémentation, la manipulation, le contrôle de l'intégrité sémantique, et l'évolution des bases de données des environnements de CAO.

La complexité des problèmes associés à ces activités est due, en partie, au fait que les informations créées et manipulées dans ces environnements se caractérisent par leurs corrélations multiples, et par la grande variété des contraintes d'intégrité qu'elles doivent vérifier pour se maintenir conformes à la réalité qu'elles représentent et cohérentes entre elles.

Comme il a été indiqué dans cette thèse, les concepts et les techniques des modèles de données, aussi bien conventionnels que sémantiques, ne sont pas complètement appropriés à la gestion de données en CAO. Le résultat est qu'une partie importante de cette gestion doit être réalisée manuellement, ou à l'aide de programmes spécialisés qui sont indépendants du SGBD, ce qui complique le contrôle de l'intégrité sémantique et l'évolution des bases de données, ainsi que le développement et la maintenance des programmes d'application.

Travail réalisé

En réponse à ces problèmes, notre première démarche a consisté à identifier de manière précise les exigences concernant la gestion de données dans les environnements de CAO, à partir d'une étude approfondie de la bibliographie et des expériences personnelles acquises pendant notre participation dans deux projets de recherche en CAO, le projet CASCADE [Merm 83], et le projet AIDA [Thon 87] du Programme Européen ESPRIT. Ces exigences ont été analysées dans le chapitre 1 de cette thèse, et nos conclusions à cet égard peuvent être récapitulées en termes des postulats suivants:

a) Les aspects structuraux, opératoires et sémantiques des bases de données doivent être intégrés et traités dans un contexte plus riche et plus uniforme que celui des modèles de données.
b) Les concepts et les techniques utilisés pour la modélisation des applications de CAO doivent permettre non seulement la conception de la base de données, mais aussi son implémentation et sa manipulation.

c) Le système de gestion de données doit être capable de s'adapter facilement aux évolutions permanentes des environnements de conception.

Motivés par notre expérience sur l'emploi de modèles sémantiques [Ania 85] et par les résultats des travaux sur les systèmes à base de connaissances, réalisés au Laboratoire ARTEMIS, de l'Institut IMAG, et décrits dans [Rech 85], nous avons ensuite réalisé une étude bibliographique détaillée des recherches qui ont proposé des solutions à des problèmes similaires, et qui ont été réalisées non seulement dans le domaine des bases de données, mais aussi dans d'autres domaines de recherche tels que l'intelligence artificielle et les langages de programmation. Les recherches qui ont eu la plus grande influence sur l'approche que nous avons suivie dans notre travail ont été décrites dans l'introduction de cette thèse.

Les résultats des démarches précédentes ont été à l'origine de l'infrastructure de représentation de connaissances que nous avons présentée dans les chapitres 2 à 4, et qui a été réalisée dans le système expérimental, OBMS, que nous avons développé comme véhicule de recherche pour valider les concepts et les techniques présentés dans cette thèse. Une description de ce système a été incluse dans l'annexe B. L'emploi d'OBMS pour contrôler de l'intégrité sémantique des bases d'objets du système CASCADE, a été décrit dans [MerA 87], et actuellement, nous sommes en train d'évaluer son adéquation au développement de Systèmes de Gestion de Processus de Conception (*en anglais*: Design Management Systems).

Evaluation du travail

Parmi les problèmes généraux mentionnés ci-dessus, cette thèse a mis l'accent sur les aspects relatifs à la modélisation des propriétés logiques des applications de CAO, telles qu'elles sont perçues au niveau conceptuel, et au contrôle de l'intégrité sémantique de l'information dans les environnements de conception.

Certains concepts et techniques des modèles sémantiques, développés dans le domaine des bases de données, ont été raffinés ou étendus pour satisfaire les exigences particulières des environnements de CAO. D'autres concepts et techniques développés dans les domaines de l'intelligence artificielle et des langages de programmation ont été adaptés aux problèmes de gestion de bases de données.

Ces concepts et ces techniques ont été intégrés avec des nouveaux concepts pour constituer une infrastructure de représentation de connaissances, centrée-objet, où la gestion de données est vue comme un processus de

modélisation dynamique qui consiste à représenter, dans des bases d'objets, l'état et l'évolution permanente des environnements de conception.

A cet égard, les principales contributions de cette thèse sont les suivantes. La notion de "bases d'objets" a été introduite pour intégrer les différents types de connaissances qui ont été traditionnellement traités séparément dans le domaine des bases de données. Les avantages de cette intégration (*e.g.*, la possibilité de mettre facilement en œuvre divers mécanismes de raisonnement, et de formuler des requêtes avec une connaissance moins précise de l'organisation de la base d'objets) ont été argumentés et illustrés au travers d'exemples.

L'emploi des "contraintes d'intégrité", du domaine des bases de données, a été étendu pour définir de manière uniforme tous les aspects structuraux, opératoires et sémantiques des bases d'objets, et il a été montré que cette approche permet de développer des systèmes de gestion de bases d'objets, tels qu'OBMS, capables de s'adapter à des environnements différents et évolutifs, et augmente la capacité pour ces systèmes de contrôler l'intégrité sémantique des bases d'objets.

Le paradigme de modélisation présenté dans cette thèse et inspiré du concept de réseau sémantique, utilisé dans l'intelligence artificielle, a été conçu pour simplifier l'implémentation, la manipulation et l'évolution des bases d'objets des applications complexes comme celles de la CAO.

Les mécanismes d'abstraction "par agrégation" et "par généralisation", utilisés dans certains modèles sémantiques, ont été adaptés et étendus pour résoudre essentiellement deux problèmes inhérents à la gestion de données des systèmes de CAO; à savoir, la définition et la manipulation d'objets composites et d'objets génériques, et l'organisation multi-hiérarchisée des objets.

En particulier, la notion d'objets génériques et celle d'objets représentatifs ont été introduites pour permettre la représentation d'associations qui sont caractéristiques des environnements de conception (*e.g.*, les associations entre un objet en cours de conception et ces différentes versions ou alternatives), et qui ne peuvent pas être représentées de façon directe dans les SGBD conventionnels.

La notion de "schéma" et la technique de représentation de connaissances "par attachement procédural", utilisées dans l'intelligence artificielle, ont été adaptées aux problèmes des bases de données en CAO, notamment en ce qui concerne la représentation de contraintes d'intégrité dont la vérification requiert l'emploi de connaissances particulières au domaine d'application.

Enfin, le concept de "méthode", caractéristique de certains langages orientés-objet, a été adapté au contexte des bases de données pour traiter les problèmes relatifs à la représentation des propriétés opératoires des objets créés et manipulés dans les environnements de conception.

D'autre part, il y a au moins deux limitations inhérentes à l'approche de modélisation proposée dans cette thèse. Les contraintes d'intégrité qui ne sont pas prédéfinies dans l'infrastructure peuvent être représentées à l'aide de prédicats dans les bases d'objets, ce qui donne à cette infrastructure le caractère extensible qui lui permet de s'adapter facilement à des environnements différents et évolutifs. Toutefois, la définition de ces prédicats requiert la connaissance du langage de programmation utilisé pour implémenter le système de gestion de bases d'objets, ou d'un langage de programmation qui puisse être interfacé de manière dynamique avec ce système.

Une remarque similaire peut être faite au sujet des requêtes, car celles-ci ne peuvent pas toujours être formulées (ou formulées de manière optimale) en utilisant tout simplement les fonctions primitives de l'infrastructure (*e.g.*, :$instances et :retrouver). Par exemple, les deux expressions suivantes peuvent être utilisées pour retrouver le nom des concepteurs qui ont conçu le plus grand nombre de circuits:

```
(:retrouver 'nom %Concepteur
    '(circuits-conçus '> (:productivité-moyenne)))

(let (pm (:productivité-moyenne))
    (:retrouver 'nom %Concepteur '(circuits-conçus '> pm)))
```

La deuxième expression est évidement beaucoup plus performante que la première, car elle n'évalue qu'une seule fois la «productivité moyenne» des concepteurs, mais cette optimisation requiert la connaissance d'un langage de programmation qui peut être celui utilisé pour implémenter le système de gestion de bases d'objets (comme dans l'exemple précédent), ou un autre qui puisse être interfacé de manière dynamique avec ce système. Ces limitations, ainsi que d'autres, constituent la base de nos suggestions pour des recherches futures.

Suggestions pour des recherches futures

Dans les environnements de CAO, l'être humain accède peu souvent de manière directe à la base de données (usuellement juste pour regarder rapidement son contenu ou son état général [L&al 85]). Il y accède plutôt à

travers les programmes d'application qui, à son tour, appellent le SGBD. Néanmoins, l'existence d'un langage de requêtes autonome, c'est-à-dire indépendant de tout autre langage de programmation, pourrait simplifier non seulement les accès directs à la base, mais aussi le développement des programmes d'application et, surtout, l'optimisation automatique des requêtes.

En fait, cette indépendance par rapport aux langages de programmation est importante aussi pour la définition des prédicats, car l'existence d'un langage formel prédicatif, unique, pourrait être utilisé pour vérifier la cohérence logique de l'ensemble des prédicats définis dans la base. Actuellement cette cohérence est difficile ou impossible à vérifier, dans la plupart des cas, du fait que les prédicats peuvent être écrits dans des langages différents dont la sémantique ne peut pas toujours être définie de manière formelle.

Deux autres sujets qui n'ont pas été traités de manière suffisamment approfondie dans cette thèse et qui, à notre avis, doivent être analysés avec plus de détail sont le traitement d'erreurs et de cas exceptionnels, et l'organisation des opérateurs des objets. A cet égard, nous considérons que l'infrastructure présentée dans cette thèse contient déjà plusieurs éléments de base qui seront nécessaires pour aborder ces deux problèmes dans un contexte centré-objet. Par exemple, la notion de classe peut être utilisée pour regrouper les erreurs et les cas exceptionnels en fonction de leurs propriétés structurelles et opératoires, et la notion d'opérateur peut être utilisée pour définir les réactions ou "handlers" correspondant à chacune de ces classes.

Par ailleurs, nous n'avons pas abordé les problèmes relatifs au contrôle de l'accès concurrent à la base ou à des informations confidentielles, ni ceux concernant la reprise après pannes, et la distribution des activités de conception à l'aide d'un serveur et de plusieurs stations de travail. Ces problèmes doivent pourtant être analysés dans des recherches futures, et les solutions de ces recherches peuvent impliquer soit des modifications soit des extensions aux concepts et aux techniques décrits dans cette thèse.

Enfin, les problèmes relatifs à la gestion de données en CAO, ainsi que leurs solutions, sont illimités. Il y a, par exemple, d'autres fonctionnalités, telles que le traitement des valeurs nulles et des connaissances incertaines, qui peuvent être utiles dans les environnements de conception et que nous n'avons pas incorporées dans notre infrastructure de représentation de connaissances. Toutefois, il y a aussi des complications additionnelles dans le fait d'incorporer ces fonctionnalités dans un système, en particulier en ce qui concerne ses performances.

Nous espérons que l'expérience acquise en utilisant les concepts et les techniques développés dans cette thèse permettra de mieux évaluer les avantages, et les inconvénients, d'inclure des nouvelles fonctionnalités dans les systèmes de gestion de données des environnements de CAO.

REFERENCES

Références

[Abri 74] Abrial J., Data semantics, *dans* J. Klimbie et K. Koffeman (eds.), *Data management systems*, North-Holland, Amsterdam, Pays-Bas, 1974, pp. 1-59.

[Adib 81] Adiba M., Derived relations: a unified mechanism for views, snapshots and distributed data, *Proc. 7th Int. Conf. on VLDB*, Cannes, France, septembre 1981, pp. 293-305.

[Ania 85] Ania I., Archivage et classement en bureautique: analyse, modélisation, représentation à l'aide du modèle TIGRE, *Rapport de DEA*, ENSIMA, Grenoble, France, septembre 1985.

[B&al 77] Bobrow D., Kaplan R., Kay M. Norman D., Thompson H. et Winograd T., GUS : a frame-driven dialog system, Artificial Intelligence, Vol. 8, n° 2 (avril 1977), pp. 155-173.

[BarD 81] Baroody A. et DeWitt D., An object-oriented approach to database system implementation, *ACM TODS*, Vol. 6, n° 4 (décembre 1981), pp. 577-601.

[BatB 84] Batory D. et Buchmann A., Molecular objects, abstract data types, and data models: a framework, *Proc. 10th Int. Conf. on VLDB*, Singapour, août 1984, pp. 172-184.

[BatK 85] Batory D. et Kim W., Modelling concepts for VLSI CAD objects, *ACM TODS*, Vol. 10, n° 3 (septembre 1985), pp. 322-346.

[BD3 83] Bases de données, nouvelles perspectives, *Rapport du groupe BD3, INRIA-ADI*, France, janvier 1983.

[BhaK 87] Bhateja R. et Katz R., VALKYRIE: a validation subsystem of a version server for computer-aided design data, *Proc. 24th DAC*, Miami, FL, juin 1987, pp. 321-327.

[BobG 80] Bobrow D. et Goldstein I., Representing design alternatives, *Proc. Conf. on Artificial Intelligence and Simulation of Behavior*, Amsterdam, Pays-Bas, juillet 1980.

[Bobr 80] Bobrow D. (ed.), *Artificial Intelligence*, Vol. 13, n° 1-2 (avril 1980), Numéro spécial sur la logique non-monotone.

[BobW 77] Bobrow D. et Winograd T., An overview of KRL, a knowledge representation language, *Cognitive Science*, Vol. 1, n° 1 (janvier 1977), pp. 3-46.

[BoMW 84] Borgida A., Mylopoulos J. et Wong H., Generalization/specialization as a basis for software specification, *dans* [BrMS 84], pp. 87-117.

[Borg 84] Borgida A., Groundwork on the problem of exceptions in information systems, *Rapport de recherche*, Department of Computer Science, Rutgers University.

[BrMS 84] Brodie M., Mylopoulos J. et Schmidt J., *On conceptual modelling*, Springer-Verlag, New York, NY, 1984.

[Brod 81] Brodie M., On modelling behavioural semantics of databases, *Proc. 7th Int. Conf. on VLDB*, Cannes, France, septembre 1981, pp. 32-42.

[Brod 82] Brodie M., Axiomatic definition for data model semantics, *Information Systems*, Vol. 7, n° 2 (1982), pp. 183-197.

[BroZ 80] Brodie M. et Zilles S. (eds.), Proc. Workshop on Data Abstraction, Databases and Conceptual Modelling, *ACM SIGPLAN Notices*, Vol. 16, n° 1 (janvier 1981).

[Buch 84] Buchmann A., Current trends in CAD databases, *CAD*, Vol. 16, n° 3 (mai 1984), pp. 123-126.

[CarW 86] Cardelli L. et Wegner P., On understanding types, data abstractions, and polymorphism, *ACM Computing Surveys*, Vol. 17, n° 4 (décembre 1985), pp. 471-522.

[CFHL 86] Chu K., Fishburn J., Honeyman P. et Lien E., A database-driven VLSI design system, IEEE Trans. on CAD, Vol. CAD-5, n° 1 (janvier 1986), pp. 180-187.

[Chai 86] Chailloux J., *Le_Lisp Version 15.2: manuel de référence.* INRIA, France, mai 1986.

[Chen 76] Chen P., The entity-relationship model: toward a unified view of data, *ACM TODS*, Vol. 1, n° 1 (mars 1976), pp. 9-36.

[CODA 78] Report on the CODASYL Data Description Language Committee, *Information Systems*, Vol. 3, n° 4 (1978).

[Codd 70] Codd E., A relational model of data for large shared data banks, *CACM*, Vol. 13, n° 6 (juin 1970), pp. 377-387.

[Codd 79] Codd E., Extending the database relational model to capture more meaning, *ACM TODS*, Vol. 4, n° 4 (décembre 1979), pp. 397-434.

[CohF 82] Cohen P. et Feigenbaum E., *The handbook of artificial intelligence*, Vol. 3, pp. 114-119, Pitman Books, Londres, 1982.

[CopM 84] Copeland G. et Maier D., Making Smalltalk a database system, *dans* B. Yormark (ed.), Proc. SIGMOD '84, Boston, MA, juin 1984, *ACM SIGMOD Record*, Vol. 14, n° 2, pp. 316-325.

[D&al 85] Dayal U., Buchman A., Goldhirsch D., Heiler S., Manola F., Orenstein J. et Rosenthal A., Probe: a research project in knowledge-oriented database systems. Preliminary analysis, *Rapport de recherche* CCA-85-03, Computer Corporation of America, Cambridge, MA, juillet 1985.

[DaDH 72] Dahl O., Dijkstra E. et Hoare C. (eds.), *Structured programming*, Academic Press, New York, NY, 1972.

[DahH 72] Dahl O. et Hoare C., Hierarchical program structures, *dans* [DaDH 72], pp. 175-220.

[DaLW 84] Dadam P., Lum V. et Werner H., Integration of time versions into a relational database system, *Proc. 10th Int. Conf. on VLDB*, Singapour, août 1984, pp. 509-522.

[Davi 81] David B., Méthodologie pour la construction de systèmes CAO: SIGMA-CAO, thèse d'état, USMG/INPG, Grenoble, France, septembre 1981.

[DelA 82] Delobel C. et Adiba M, *Bases de données et systèmes relationnels*, Dunod, Paris 1982.

[East 80] Eastman C., System facilities for CAD databases, *Proc. 17th DAC*, Minneapolis, MN, juin 1980, pp. 50-56.

[East 81] Eastman C., Database facilities for engineering design, *Proceedings of the IEEE*, Vol. 69, n° 10 (octobre 1981), pp. 1249-1263.

[FikK 85] Fikes R. et Kehler T., The role of frame-based representation in reasoning, *CACM*, Vol. 28, n° 9 (septembre 1985), pp. 904-920.

[GenN 87] Genesereth M. et Nilsson N., *Logical foundations of artificial intelligence*, Morgan Kaufmann, Los Altos, CA, 1987.

[GolR 83] Goldberg A. et Robson D., *Smalltalk-80: the language and its implementation*, Addison-Wesley, Reading, MA, 1983.

[Gutt 77] Guttag J., Abstract data types and the development of data structures, *dans* [Wegb 77], pp. 396-404.

[HamM 81] Hammer M. et McLeod D., Database description with SDM: a semantic database model, *ACM TODS*, Vol. 6, n° 3 (septembre 1981), pp. 351-386.

[Harp 86] Harper R., Introduction to Standard ML, *LFCS Report Series* ECS-LFCS-86-14, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, août 1986.

[Hoar 72] Hoare C., Notes on data structuring, *dans* [DaDH 72], pp. 83-174.

[Hosk 79] Hoskins E., Descriptive databases in some design/manufacturing environments, *Proc. 16th DAC*, San Diego, CA, juin 1979, pp. 421-436.

[Hull 84] Hullot J., *Programmer en Ceyx, Ceyx-Version 15*, INRIA, Le Chesnay, France, été 1984.

[JuLL 86] Jullien C., Leblond A. et Lecourvoisier J., A database interface for an integrated CAD system, *Proc. 23rd DAC*, Las Vegas NV, juin 1986, pp. 760-767.

[KaAC 86] Katz R., Anwarrudin M. et Chang E., A version server for computer-aided design data, *Proc. 23rd DAC*, Las Vegas NV, juin 1986, pp. 27-33.

[KatL 82] Katz R. et Lehman T., Storage structures for versions and alternatives, *Rapport technique* n° 479, Computer Sciences Department, University of Wisconsin-Madison, juillet 1982.

[Katz 82] Katz R., A database approach for managing VLSI design data, *Proc. 19th DAC*, Las Vegas NV, juin 1982, pp. 274-282.

[Katz 83] Katz R., Managing the chip design database, *Rapport technique* n° 506, Computer Sciences Department, University of Wisconsin-Madison, mai 1983.

[Kent 79] Kent W., Limitations of record-based information models, *ACM TODS*, Vol. 4, n° 1 (mars 1979), pp. 107-131.

[Kers 84] Kerschberg Larry. (ed.), *Proc. 1st Int. Workshop on Expert Database Systems*, Kiawah Island, South Carolina, octobre 1984, Institute of Information Management, Technology and Policy, College of Business Administration, University of South Carolina, Columbia, SC.

[Kers 86] Kerschberg Larry. (ed.), *Proc. 1st Int. Conf. on Expert Database Systems*, Charleston, South Carolina, avril 1986, Institute of information management, technology and policy, College of business administration, Univ. of South Carolina, Columbia, SC.

[KhoC 86] Khoshafian S. et Copeland G., Object identity, *dans* [Meyr 86], pp. 406-416.

[King 83] King J. (ed.), Special issue on AI and database research, *SIGART Newsletter*, n° 86 (octobre 1983), pp. 32-72.

[KinM 84] King R. et McLeod D., A unified model and methodology for conceptual database design, *dans* [BrMS 84], pp. 313-331.

[KlSW 86] Klahold P., Schlageter G. et Wilkes W., A general model for version management in databases, *Proc. 12th Int. Conf. on VLDB*, Kyoto, Japon, août 1986, pp. 319-327.

[L&al 85] Lockemann P., Dittrich K., Adams M., Bever M., Ferkinghoff B., Gotthard W., Kotz A., Liedtke R., Lüke B. et Mülle J., Database requirements of engineering applications: an analysis, *Rapport de recherche* 12/85, Forschungszentrum Informatik, Universität Karlsruhe, RFA, juillet 1985.

[Lieb 81] Lieberman H., A preview of Act 1, *Mémorandum* n° 625, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, juin 1981.

[Litw 80] Litwin W., Trie hashing, INRIA, Le Chesnay, France, novembre 1980, pp. 1-24.

[LoPV 83] López M., Palazzo J. et Vélez F., The TIGRE data model, *Rapport de recherche* TIGRE n° 2, Laboratoire de Génie Informatique (IMAG), Grenoble, France, novembre 1983.

[LSAS 77] Liskov B., Snyder A., Atkinson R. et Schaffert C., Abstraction mechanisms in CLU, *CACM*, Vol. 20, n° 8 (août 1977), pp. 564-576.

[MacG 85] MacGregor R., ARIEL, a semantic front-end to relational DBMSs, , *Proc. 11th Int. Conf. on VLDB*, Stockholm, août 1985, pp. 305-315.

[McCa 80] McCarthy J., Circumscription: a form of non-monotonic reasoning, *dans* [Bobr 80], pp. 27-39.

[McLS 80] McLeod D. et Smith J., Abstraction in databases, *dans* [BroZ 80], pp. 19-25.

[McNB 83] McLeod D., Narayanaswamy K. et Bapa Rao K., An approach to information management for CAD/VLSI applications, *Databases for Engineering Applications, ACM Database Week*, mai 1983, pp. 39-50.

[MerA 87] Mermet J. et Ania I., On the consistency of data in integrated CAD systems, *in* F. Rammig (ed.), *Proc. IFIP WG 10.2 Workshop on Tool Integration and Design Environments*, Paderborn, RFA, novembre 1987 (North-Holland).

[Merm 83] Mermet J., Circuit and system computer aided design and engineering, *Proc. 1st Int. Conf. on Computer Applications in Production and Engineering*, Amsterdam, Pays-Bas, avril 1983, pp. 245-262.

[Merm 85] Mermet J., Several steps towards a circuits integrated CAD system: CASCADE, *Proc. 7th Int. Conf. CHDL*, Tokyo, Japon, août 1985 (North-Holland).

[Meyr 86] Meyrowitz N. (ed.), Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications, Portland, OR, septembre 1986, *SIGPLAN Notices*, Vol. 21, n° 11 (novembre 1986).

[Mins 75] Minsky M., A framework for representing knowledge, *dans* P. Winston (ed.), *The psychology of computer vision*, McGraw-Hill, New York, NY, 1975, pp. 211-277.

[Moon 86] Moon D., Object-oriented programming with Flavors, *dans* [Meyr 86], pp. 1-8.

[MSOP 86] Maier D., Stein J., Otis A. et Purdy A., Development of an object-oriented DBMS, *dans* [Meyr 86], pp. 472-482.

[MyBW 80] Mylopoulos J., Bernstein P. et Wong H., A language facility for designing database-intensive applications, *ACM TODS*, Vol. 5, n° 2 (juin 1980), pp. 185-207.

[Mylo 80] Mylopoulos J., An overview of knowledge representation, dans [BroZ 80], pp. 5-12.

[NewS 86] Newton A. et Sangiovanni-Vincentelli A., Computer-aided design for VLSI circuits, *Computer*, avril 1986, pp. 38-60.

[Nils 82] Nilsson N., *Principles of artificial intelligence*, Tioga Publishing Co., Palo Alto, CA, 1980.

[NoRR 82] Noon W., Robbins K. et Roberts M., A design system approach to data integrity, *Proc. 19th DAC*, Las Vegas, NV, juin 1982, pp. 699-705.

[Orga 76] Organick, E. (chrm.), Proc. ACM SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition, and Structure, Salt Lake City, UT, mars 1976, *SIGPLAN Notices*, Vol. 11, n° 2 (novembre 1976).

[Parn 72] Parnas D., A technique for software module specification with examples, *CACM*, Vol. 15, n° 5 (mai 1972), pp. 330-336.

[Polo 87] Polo A., Infrastructure de stockage pour des bases de données CAO centrées-objet, *Rapport de DEA*, ENSIMA, Grenoble, France, juin 1987.

[Prei 83] Preiss K., Future CAD systems, *CAD*, Vol. 15, n° 4 (juillet 1983), pp. 223-227.

[Quil 68] Quillian M., Semantic memory, *dans* M. Minsky (ed.), *Semantic information processing*, MIT Press, Cambridge, MA, 1968, pp. 227-270.

[Rech 85] Rechenmann F., Shirka: mécanismes d'inférence sur une base de connaissances centrée-objet, *5ième Congrès AFCET-ADI-INRIA, Reconnaissance des Formes et Intelligence Artificielle*, Grenoble, France, novembre 1985.

[Reit 80] Reiter R., A logic for default reasoning, *dans* [Bobr 80], pp. 81-132.

[RieN 86] Rieu D. et Nguyen G., Semantics of CAD objects for generalized databases, *Proc. 23rd DAC*, Las Vegas, NV, juin 1986, pp. 34-40.

[Rieu 85] Rieu D., Modèle et fonctionnalités d'un SGBD pour les applications CAO, thèse de doctorat, INPG, Grenoble, France, juillet 1985.

[RobG 77] Roberts R. et Goldstein I., The FRL Primer, *Mémorandum* n° 408, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, juillet 1977.

[ShWL 77] Shaw M., Wulf W. et London R., Abstraction and verification in Alphard: defining and specifying iteration and generators, *CACM*, Vol. 20, n° 8 (septembre 1977), pp. 553-564.

[SmiS 77] Smith J. et Smith D., Database abstractions: aggregation and generalization, *ACM TODS*, Vol. 2, n° 2 (juin 1977), pp. 105-133.

[StaA 86] Staley S. et Anderson D., Functional specifications for CAD databases, *CAD*, Vol. 18, n° 3 (avril 1986), pp. 132-138.

[SteB 86] Stefik M. et Bobrow D., Object-oriented programming: themes and variations, *The AI Magazine*, Vol. 6, n° 4 (hiver 1986), pp. 40-62.

[Ston 84] Stonebraker M., Adding semantic knowledge to a relational database system, *dans* [BrMS 84], pp. 333-356.

[SucW 79] Sucher D. et Wann D., A design aids database for physical components, *Proc. 16th DAC*, San Diego, CA, juin 1979, pp. 414-420.

[Thon 87] Thonemann H., AIDA: advaced integrated circuits design aids. Aims and progress towards a new generation of VLSI tools, *Proc. ECIP Workshop, ESPRIT Conference*, Bruxelles, septembre 1987, pp.279-286.

[Wass 78] Wasserman A., A software engineering view of data base management, *Proc. 4th Int. Conf. on VLDB*, Berlin Ouest, RFA, septembre 1978, pp. 23-35.

[Webe 78] Weber H., A software engineering view of data base systems, *Proc. 4th Int. Conf. on VLDB*, Berlin Ouest, RFA, septembre 1978, pp. 36-51.

[Wegb 77] Wegbreit B. (ed.), Selected papers from the ACM SIGPLAN/SIGMOD Conference on Data: Abstraction, Definition, and Structure, Salt Lake City, UT, mars 1976, *CACM*, Vol. 20, n° 6 (juin 1977), pp. 382-420.

[Wilm 79] Wilmore J., The design of an efficient data base to support an interactive LSI layout system, *Proc. 16th DAC*, San Diego, CA, juin 1979, pp. 445-451.

[WinH 84] Winston P. et Horn B., *Lisp*, 2ième édition, Addison-Wesley, Reading, MA, 1984.

[Wins 84] Winston P., *Artificial intelligence*, 2ième édition, Addison-Wesley, Reading, MA, 1984.

[Wirt 71] Wirth N., Program development by stepwise refinement, *CACM*, Vol. 14, n° 4 (avril 1971), pp. 221-227.

[WonM 77] Wong H. et Mylopoulos J., Two views of data semantics: a survey of data models in artificial intelligence and database management, *Informatics*, Vol. 15, n° 3 (octobre 1977), pp. 344-383.

[Zani 83] Zaniolo C., The database language GEM, *dans* D. DeWitt et G. Gardarin (eds.), Proc. SIGMOD *'83, Database week*, San Jose, CA, mai 1983, *ACM SIGMOD Record*, Vol. 13, n° 4, pp. 207-218.

[Zdon 84] Zdonik S., Object management system concepts, *Proc. 2nd ACM-SIGOA Conf. on Office Information Systems*, Toronto, Canada, juin 1984, pp. 13-19.

ANNEXES

Annexe A

Contraintes d'intégrité sur les associations entre objets

A.1. Associations entre objets agrégats et objets agrégés

aa1)	L'association composant-de, appelée aussi agrégé-de, n'est définie qu'entre deux objets instance ou entre deux objets classe.
aa2)	Si A et B sont des objets classe, A est un composant de B lorsque celui-ci a des descripteurs qui indiquent que les instances de B doivent avoir un ou plusieurs attributs dont les valeurs sont des instances de A.
aa3)	Si un objet X est composant d'un objet Y, alors Y est un agrégat de X.
aa4)	Tout objet peut avoir un nombre quelconque de composants et être composant d'un nombre quelconque d'agrégats.
aa5)	Si un objet X est composant d'un objet Y, alors tous les composants de X sont aussi des composants de Y (transitivité).
aa6)	Tout objet est composant et agrégat de lui-même (réflexivité).
aa7)	Toute instance hérite des caractéristiques et, en général, de tous les attributs de ses composantes. Ces caractéristiques et ces attributs sont appelés caractéristiques et attributs internes.
aa8)	Aucun objet ne peut être détruit s'il est composant d'un autre objet (Voir aussi la règle aa9).
aa9)	La destruction d'un objet X (classe ou instance) entraîne la destruction préalable de tous ses composants subordonnés qui n'ont pas un autre superordonné. Ainsi, si en accord avec les règles aa8, cc8, ci6 et gr13 la destruction de ces composants ne peut pas être réalisée, alors l'objet X ne peut pas être détruit non plus. (Voir aussi les règles cc9 et gr14.)

A.2. Associations entre bases et classes

bc1)	Deux classes associées à la même base ne peuvent pas avoir le même nom.
bc2)	La suppression d'une association entre une base et une classe entraîne la destruction préalable de cette classe. (Voir la règle aa8.)

A.3. Associations entre bases et instances

bi1)	La suppression d'une association entre une base et une instance entraîne la destruction préalable de cette instance. (Voir la règle aa8.)

A.4. Associations entre objets classe

cc1)	L'association binaire "spécialisation-de" est aussi appelée sous-classe-de et n'est définie qu'entre objets classe.
cc2)	Si un objet A est une sous-classe d'un objet B, alors B est une super-classe de A.
cc3)	Toute classe peut avoir un nombre quelconque de sous-classes et de super-classes.
cc4)	Les sous-classes de chaque classe sont aussi des sous-classes de toutes ses super-classes (transitivité).
cc5)	Aucune classe A ne peut être spécialisation d'une classe B si celle-ci est une spécialisation de A (asymétrie).
cc6)	Toute classe est sous-classe et super-classe d'elle-même (réflexivité).
cc7)	Chaque classe hérite de tous les attributs de ses super-classes et, par conséquent, elle ne peut ni redéfinir ni spécialiser ces attributs; pourtant, elle peut avoir d'autres attributs additionnels. (Voir aussi la règle ci4.)
cc8)	La destruction d'une classe entraîne la destruction préalable de toutes ses sous-classes qui n'ont pas une autre super-classe. (Voir la règle aa8.)
cc9)	Avant d'être détruite, toute classe lègue ses super-classes, ses caractéristiques et ses subordonnées, à ses sous-classes qui ont d'autres super-classes et qui, par conséquent, ne peuvent pas être détruites.
cc10)	Aucune classe A ne peut être spécialisation d'une classe B si celle-ci est une classe générique ou représentative de la classe A, c'est-à-dire si les instances de B ont été définies comme des objets génériques ou représentatifs des instances de la classe A.
cc11)	Aucune classe A ne peut être définie comme spécialisation d'une classe B, si la classe générique de la classe B n'est pas égale à, ou spécialisation de, la classe générique de la classe A.

A.5. Associations entre classes et instances

ci1)	L'association "x est instance-de A" est définie uniquement si x est un objet instance et A est un objet classe.
ci2)	Si x est une instance de A, alors A est une classe de x.
ci3)	Toute instance peut avoir un nombre quelconque de classes, et toute classe peut avoir un nombre quelconque d'instances.
ci4)	Si x est instance d'une classe A et celle-ci est sous-classe de B, alors x est aussi une instance de B. Autrement dit, les instances de toute classe sont aussi des instances de ses super-classes. (Voir aussi la règle gr7.)
ci5)	Si un objet x est une instance d'un objet A, alors x doit respecter toutes les contraintes d'intégrité définies par lès descripteurs de A. En particulier, une instance ne peut ni redéfinir ou spécialiser les propriétés définies par les descripteurs-c et par les descripteurs-p, ni appartenir à des classes ayant des représentatifs différents. Pourtant, toute instance peut avoir des caractéristiques qui ne sont pas définies par ses classes. (Voir aussi la règle gr7.)
ci6)	La destruction d'un objet classe entraîne la destruction préalable de toutes ses instances qui ne sont pas associées à une autre classe. (Voir la règle aa8.)

A.6. Associations entre objets génériques et objets représentatifs

gr1)	L'association représentatif-de n'est définie qu'entre deux objets instance ou entre deux objets classe.
gr2)	Si un objet X est représentatif d'un objet Y, alors Y est le générique de X.
gr3)	Tout objet (classe ou instance) peut avoir un nombre quelconque de représentatifs, mais aucun objet ne peut avoir plus d'un objet générique immédiat (*Cf.* règle gr8).
gr4)	La classe représentative d'une instance X est la classe définie par le descripteur de la propriété "représentatifs" de la classe de X, c'est-à-dire la classe à laquelle doivent appartenir tous les représentatifs de X.
gr5)	Si A est représentatif de B, et si A et B sont des objets classe, alors les instances de A ne peuvent avoir comme objets génériques que des instances de B. Autrement dit, une instance X ne peut être définie comme représentative d'une instance Y que si X appartient à la classe représentative de Y ou si celle-ci est nulle.
gr6)	Aucun objet ne peut être associé en même temps à deux classes définissant des représentatifs différents.

gr7)	Les objets représentatifs héritent de tous les attributs de leur objet générique et, en conséquence, *a)* ils ne peuvent ni redéfinir ni spécialiser ces attributs, mais ils peuvent avoir des attributs additionnels, et *b)* chacun de ces objets représentatifs est aussi une instance des classes auxquelles appartient son objet générique. En d'autres termes, les représentatifs de toute instance sont aussi des instances de toutes ses classes. Ceci implique que tout objet représentatif doit respecter aussi bien les contraintes d'intégrité définies par ses classes, que les contraintes d'intégrité définies par les classes de son objet générique. (Voir aussi la règle ci5.)
gr8)	Si un objet X est représentatif d'un objet Y, tous les représentatifs de X sont aussi des représentatifs de Y (transitivité).
gr9)	Aucun objet X ne peut être représentatif d'un objet Y si celui-ci est représentatif de X (asymétrie).
gr10)	Tout objet est représentatif et générique de lui-même (réflexivité).
gr11)	Aucun objet X ne peut être représentatif d'un objet Y si X et Y ont une classe commune.
gr12)	Aucune classe A ne peut être représentative d'une classe B si celle-ci est sous-classe ou super-classe de A, si le générique dont A hérité de ces super-classes n'est pas nul ou une super-classe (ancêtre) de A, ou si la définition d'attributs de A redéfinit ou raffine l'une des définitions d'attributs de B.
gr13)	Aucun objet ne peut être détruit s'il est représentatif d'un autre objet. (Voir aussi la règle gr14).
gr14)	La destruction d'un objet générique (classe ou instance) entraîne la destruction préalable de tous ses représentatifs. (Voir la règle aa8.)

Annexe B

Structure et opérateurs des objets

Le système expérimental OBMS que nous avons développé, au Laboratoire ARTEMIS-IMAG, comme véhicule de recherche pour valider les concepts et les techniques présentés dans cette thèse, est composé de deux processeurs (*voir* figure B.1): le *Processeur de Bases d'Objets Logiques* (PBOL), qui implémente tous les opérateurs nécessaires pour décrire et pour manipuler les objets au niveau conceptuel, et le *Processeur de Bases d'Objets Physiques* (PBOP), qui implémente les opérateurs nécessaires pour manipuler des objets persistants, c'est-à-dire ceux qui doivent être stockés sur disque.

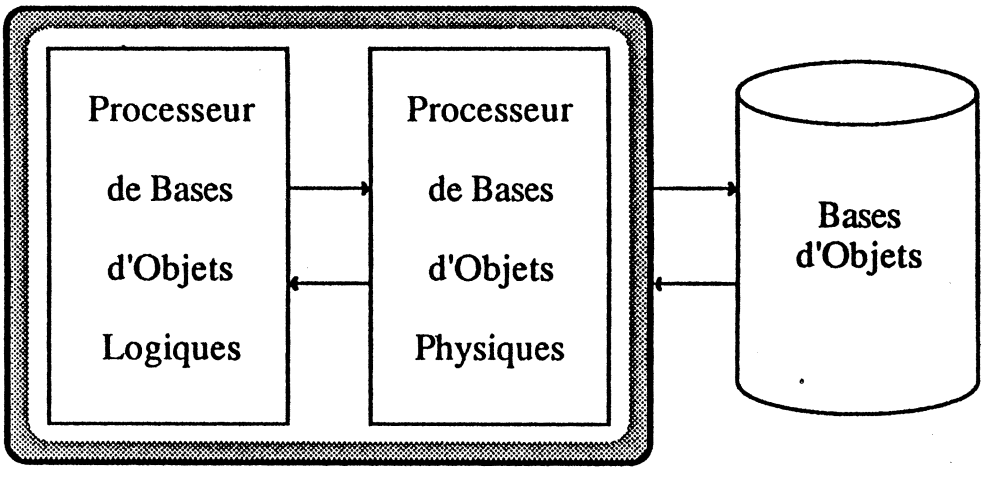


FIGURE B.1
Architecture du système expérimental OBMS

Ces deux processeurs identifient les objets par leur surrogate, mais leur structure est interprétée de manière différente. Pour le PBOL chaque objet est une agrégation de propriétés structurelles, tandis que pour le PBOP les objets ne sont que des suites d'octets de longueur variable pouvant être stockées dans des **pages** qui sont créées, sélectionnées, modifiées et supprimées en accord avec un mécanisme de hachage conçu par A. Polo [Polo 87] et basé sur les algorithmes proposés par W. Litwin [Litw 80].

Par ailleurs, l'espace de travail du PBOL se situe seulement dans la mémoire principale, tandis que celui du PBOP se situe plutôt dans la mémoire secondaire, c'est-à-dire le disque. En fait, la partie de la mémoire principale employée par le PBOP est une **mémoire cache** qui est utilisée uniquement pour diminuer le temps nécessaire pour accéder aux objets stockés sur

disque, et qui contient les N pages les plus récemment utilisées, où N est un entier positif (≥ 0) dont la valeur par défaut peut être modifiée par les utilisateurs d'OBMS, en fonction de leurs besoins et des ressources disponibles.

OBMS est un système portable qui a été entièrement développé en Le_Lisp [Chai 86]. Il comporte actuellement (décembre 1987) environ 300 fonctions qui incluent les opérateurs décrits aux chapitres 3 et 4 (*e.g.*, :+classe, :-instance et :$caractéristiques), les opérateurs du PBOP (*e.g.*, :taille-cache et :verrouiller-répertoire), des fonctions pour le traitement élémentaire d'erreurs et de cas exceptionnels (*e.g.*, :signaler-exceptions et :erreur-fatale), des extensions à Le_Lisp (*e.g.*, :date-courante et :union), et des opérateurs internes au système (*e.g.*, :générer-surrogate et :$descripteur).

Dans cette annexe nous n'avons inclus que les opérateurs pertinents à la modélisation conceptuelle des applications de CAO, c'est-à-dire que nous ne décrivons pas, par exemple, les opérateurs correspondant au stockage des objets sur disque, ou le comportement des opérateurs internes du système. En particulier, nous décrirons la structure et les opérateurs des objets qui peuvent être crées par les utilisateurs d'OBMS.

Tout de même, afin de mieux comprendre le fonctionnement global du système, nous devons mentionner que l'espace de travail du PBOL est un environnement standard de Le_Lisp, dans lequel on peut charger le système OBMS. La section B.1 décrit les opérateurs qui doivent être utilisés pour charger et pour enlever OBMS d'un tel environnement.

A partir du moment où OBMS a été chargé, les utilisateurs peuvent invoquer non seulement les opérateurs OBMS mais aussi les fonctions Lisp définies dans l'environnement en question. Ces opérateurs et ces fonctions peuvent être utilisés de manière indépendante ou combinée.

Le transfert d'objets entre la mémoire principale et la mémoire secondaire est réalisée par OBMS de manière transparente aux utilisateurs, en ce sens que ceux-ci n'ont pas besoin de demander explicitement le chargement des objets stockés sur disque.

En effet, lorsqu'un objet est référencé (à l'aide de son surrogate) pour être modifié ou analysé, OBMS charge cet objet dans la mémoire principale, si cela n'a pas encore été fait. Toutefois, OBMS ne met pas automatiquement à jour les informations du disque chaque fois qu'un objet est modifié dans la mémoire principale.

Cette fonctionnalité améliore les performances du système et permet de revenir en arrière lorsque l'on considère que les modifications réalisées sont erronées, mais requiert la demande explicite de la mise à jour du

disque[1] (voir l'opérateur :détruire, dans la section B.2, ainsi que les opérateurs :confirmer et :fermer, dans la section B.3).

Enfin, le transfert d'objets entre les mémoires principale et secondaire peut être décrit en termes des protocoles de lecture et d'écriture suivants:

> **Protocole de lecture.** Lorsque le PBOL a besoin d'un objet, il envoi au PBOP le surrogate correspondant. Le PBOP détermine (par hachage) l'identificateur de la page contenant l'objet en question, et envoi cet identificateur et le surrogate de l'objet au Lecteur d'OBMS.
>
> La première action du Lecteur consiste à chercher cette page dans la mémoire cache. Si la page n'y est pas, le Lecteur active le mécanisme qui amène ("fetch") des pages du disque, et recommence son travail; dans le cas contraire, le Lecteur cherche l'objet en question dans cette page, et l'«installe» dans l'espace de travail du PBOL.
>
> **Protocole d'écriture.** Lorsque le PBOL souhaite stocker un objet sur disque, il envoie au PBOP le surrogate correspondant, et le PBOP détermine par hachage l'identificateur de la page où cet objet doit être stocké. Si la page n'a pas encore été créée, le PBOP en crée une nouvelle. Dans tous les cas, ce processeur envoie l'identificateur de la page et le surrogate de l'objet au Ecrivain d'OBMS.
>
> La première action de l'Ecrivain consiste à chercher cette page dans la mémoire cache. Si la page n'y est pas, il active le mécanisme qui amène des pages du disque et recommence son travail; dans le cas contraire, il cherche l'objet dans la page en question.
>
> Si cette recherche échoue, l'Ecrivain insère dans cette page la représentation de l'objet situé dans l'espace de travail du PBOL et, ainsi, un nouveau objet a été créé; dans le cas contraire il remplace l'objet situé dans l'espace de travail du PBOL par l'objet étant dans la page de la mémoire cache et, ainsi, l'objet est mis à jour. Finalement, si dans l'espace de travail du PBOL il n'y a pas de représentation pour l'objet en question, ce processus implique la destruction d'un objet existant.

Dans la description d'opérateurs contenue dans cette annexe nous utilisons les conventions suivantes: les arguments entre crochets (*e.g.*, [arg]) sont facultatifs, les arguments précédés par un apostrophe (*e.g.*, 'arg) sont évalués par le système, le symbole **nil** est utilisé pour représenter tout

[1]Cette fonctionnalité est similaire à celle des logiciels pour le traitement de texte qui mettent à jour les fichiers uniquement lorsque l'utilisateur demande cette action de manière explicite.

argument dont la valeur est nulle (*e.g.*, '( ), ( ), ||, '|| et (not t)), et le symbole srgt-bc est utilisé pour représenter le surrogate de la base courante.

B.1. Opérateurs pour charger et pour enlever OBMS d'un environnement Lisp

(obms [nom-base])
(:obms [nom-base])

chargent OBMS, si cela n'a pas été fait, et appellent l'opérateur :ouvrir lorsque l'utilisateur fournit le nom d'une base d'objets. Il n'y a pas de différence entre obms et :obms.

(:obmsfin [nil])

enlève OBMS de l'environnement Lisp, après avoir mis à jour les informations du disque (si besoin est, et si l'utilisateur ne fournit pas l'argument facultatif). Cet opérateur n'implique pas la sortie de l'interprète Lisp.

(end [nil])
(:fin [nil])

impliquent la sortie de l'interprète Lisp, après avoir mis à jour les informations du disque si l'utilisateur ne fournit pas l'argument facultatif. end est la fonction standard de Lisp qui sous OBMS est précédée par l'opérateur :fermer (*voir* section B.3), si l'utilisateur fournit l'argument facultatif.

B.2. Les objets OBMS

Tout objet OBMS a au moins les propriétés suivantes:

- ◊ surrogate : identificateur généré par le système, dont la représentation externe a la forme #{ent+}, où ent+ est un entier positif[1].
- ◊ date1 : date de création de l'objet.
- ◊ date2 : date de sa dernière mise à jour.

[1]En vue de simplifier l'implémentation d'OBMS, le système est actuellement "monobase" en ce sens que les surrogates utilisés pour identifier les objets ne sont uniques que dans le contexte d'une base d'objets. Autrement dit, deux objets différents appartenant à des bases distinctes peuvent avoir le même surrogate et, en conséquence, le système ne permet pas le transfert d'objets entre les bases. Par ailleurs, le surrogate des objets base est toujours #{0}.

◊ commentaires : un nombre quelconque de commentaires sur l'objet.
◊ statut : nouveau, modifié, non-modifié, enlevé ou détruit.

Les opérateurs qui permettent l'accès à ces propriétés sont décrits dans cette section.

(:enlever 'srgt)

affecte la valeur «enlevé» à la propriété "statut" de l'objet identifié par le surrogate srgt, si cet objet n'est pas la base courante. Lorsque l'objet est la base courante, le comportement de cet opérateur est équivalent à celui de l'opérateur :fermer, dans l'expression (:fermer ( )) [*voir ci-après*].

La valeur «enlevé» dans la propriété "statut" d'un objet autorise OBMS à détruire cet objet lors de la prochaine opération de mise à jour du disque. Cela veut dire que la portée de l'opérateur :enlever n'inclut pas les objets du disque mais seulement leurs "copies" chargées dans la mémoire principale.

D'autre part, avant de modifier le statut d'un objet, :enlever vérifie que l'objet pourra être détruit (lors de la mise à jour du disque) sans violer les contraintes d'intégrité définies sur la base d'objets. Si le test échoue, :enlever ne modifie pas le statut de l'objet et signale un événement exceptionnel. Les vérifications et toutes les actions réalisées par :enlever lorsque l'objet est une classe, une instance, ou un opérateur, sont décrites dans la section B.3 (*voir* :-classe, :-instance, et :-opérateur).

(:restaurer 'srgt) *;non-implémenté.*

restaure l'objet identifié par srgt, en utilisant les informations les plus récemment confirmées (*voir* :confirmer).

(:détruire 'srgt)

détruit l'objet identifié par srgt. Pour maintenir l'intégrité sémantique de la base d'objets, cet opérateur est automatiquement précédé par la fonction :enlever. Avertissement: cet opérateur met immédiatement à jour les informations du disque et, ainsi, sa portée inclut aussi bien les objets de la mémoire principale que ceux de la mémoire secondaire (c'est-à-dire du disque); par conséquent, un objet détruit ne peut plus jamais être restauré.

Du fait que dans l'implémentation actuelle d'OBMS les objets base ont toujours le surrogate #{0}, et que l'opérateur :enlever ferme la base courante lorsqu'il reçoit comme argument le surrogate de cette base, l'opérateur :détruire ne peut pas être utilisé pour détruire des objets base (*voir* :détruire-base).

(:$date1 'srgt)

retourne la date de création de l'objet.

(:$date2 'srgt)

retourne la date de la dernière mise à jour de l'objet.

(:+commentaire 'srgt '(cmt))

ajoute un commentaire sur l'objet identifié par srgt.

(:-?commentaire 'srgt 'index)

enlève le index^{ième} commentaire de l'objet identifié par srgt. index doit être un entier plus grand que zéro, et plus petit ou égal au nombre maximal de commentaires de l'objet.

(:commentaires 'srgt [nil])

retrouve tous les commentaires sur l'objet identifié par srgt, ou les enlève tous si l'argument facultatif est fourni.

(:$statut 'srgt)

montre le statut de l'objet identifié par srgt. Si l'objet n'a pas été modifié, la valeur retournée par cet opérateur est "( )". Si l'objet n'a pas été défini, cet opérateur retourne "non-défini". Pour connaître, par exemple, le statut de la base courante, on peut utiliser l'expression (:statut (:base-courante?)).

B.3. Les objets BASE

Outre les propriétés communes à tous les objets OBMS, tout objet BASE a les propriétés suivantes:

◊ nom.
◊ classes (qui n'ont pas au moins une super-classe).
◊ instances (qui ne sont pas associées à un objet classe).

◊ opérateurs (définis par les utilisateurs).

Les opérateurs correspondant à ces propriétés sont décrits dans cette section.

(:+base nom-base)
(:créer-base nom-base)

créent une base d'objets ayant le nom spécifié par l'argument nom-base, uniquement si l'utilisateur n'a pas créé une autre base avec ce nom. La seule différence entre ces opérateurs est que le premier retourne le surrogate de la nouvelle base, tandis que le deuxième retourne son nom.

(:ouvrir nom-base)

ouvre une base d'objets ayant le nom spécifié par l'argument nom-base, ou en crée une nouvelle (en appelant l'opérateur :+base) si l'utilisateur n'a pas créé une base avec ce nom. Dans ce dernier cas, OBMS envoi à l'utilisateur le message "AVERTISSEMENT... c'est une nouvelle base...". Dans tous les cas, cette base devient la base courante.

(:fermer [nil])
(:enlever-base)

Le premier opérateur ferme la base courante, s'il y en a une, après avoir mis à jour les informations du disque (si besoin est et si le deuxième argument n'est pas fourni). Le deuxième opérateur est équivalent à (:fermer ( )), et tous les deux sont équivalents à (:enlever (:base-courante?)).

(:détruire-base nom-base)

détruit la base d'objets ayant le nom spécifié par l'argument nom-base, si cette base existe et si elle n'est pas la base courante.

(:base-courante?)

retourne le surrogate de la base courante, ou ( ) s'il n'y en a pas.

(:$nom 'srgt-bc) ;(*voir* sections B.4 et B.6).

est un opérateur polymorphique qui peut être utilisé pour connaître le nom de la base courante.

(:base? 'nom-base)

est un opérateur qui dans l'implémentation actuelle d'OBMS retourne *a*) le surrogate #{0}, lorsque la base d'objets ayant le nom spécifié par l'argument nom-base est la base courante, *b*) la valeur T (vrai), lorsque la base existe mais ce n'est pas la base courante, et *c*) la valeur ( ), lorsque l'utilisateur n'a pas créé une base d'objets ayant ce nom.

(:statut-base)

est un opérateur dont le résultat est égal à celui de l'expression (or (:statut (:base-courante?)) 'non-modifié), c'est-à-dire, il retourne le symbole «non-modifié» au lieu de la liste vide retournée par l'opérateur :statut, lorsque la base courante n'a pas été modifiée.

(:décrire-base)

est un opérateur équivalent à (eval (:base-courante?)), qui montre les propriétés "les plus importantes" de la base courante.

(:confirmer)

enregistre sur disque toute modification qui a été effectuée dans la base courante. Cette opération implique la destruction de tous les objets dont le statut est «enlevé». Pour cette raison, ils ne peuvent plus jamais être restaurés.

(:+classe ['srgt-bc] 'nom-classe ['cars-f]) ;(*voir* section B.5).
(:créer-classe nom-classe [cars-f])

créent une nouvelle classe, si dans la base courante il n'y a pas une autre classe ayant le nom spécifié par l'argument nom-classe. Eventuellement, l'utilisateur peut spécifier des caractéristiques filtrées (*Cf.* section 5 du chapitre 3) de cette classe. Cette spécification peut être constituée par un nombre quelconque de couples 'nom-car 'filtre, où nom-car est le nom d'une caractéristique-f et filtre est le nom d'une fonction ou d'un opérateur défini auparavant, ou une liste dont le premier élément est ce nom ou une "lambda expression" (Lisp), et les autres éléments sont d'éventuels arguments nécessaires pour invoquer la fonction, l'opérateur ou la lambda expression (*voir* :+caractéristique-f). :créer-classe est un opérateur équivalent qui n'évalue pas ses arguments.

(:-classe ['srgt-bc] 'srgt-cl) ;(*voir* section B.5).
(:enlever-classe nom-classe)

Le premier opérateur enlève la classe identifiée par srgt-cl si elle n'a pas d'objets génériques, superordonnés (tous les deux étant peut-être hérités des super-classes), ou agrégats. Cet opérateur enlève aussi les sous-classes, les instances et les subordonnés de srgt-cl qui ne sont pas partagés avec d'autres classes. Lorsqu'une sous-classe est partagée avec d'autres classes, srgt-cl lui lègue ses super-classes, ses caractéristiques et ses subordonnés. Par ailleurs, le comportement de cet opérateur est celui de l'opérateur :enlever dans l'expression (:enlever 'srgt-cl). :enlever-classe est un opérateur équivalent qui n'évalue pas son argument.

(:$classes ['srgt-bc] [nil]) ;(*voir* section B.5).
(:retrouver-classes)

Si le deuxième argument est fourni par l'utilisateur, le premier opérateur retourne la valeur de la propriété "classes" de la base courante, c'est-à-dire une liste contenant le surrogate de toutes les classes qui n'ont pas des super-classes. Par contre, lorsque le deuxième argument n'est pas fourni, OBMS utilise des mécanismes d'inférence en vue d'ajouter à cette liste les surrogates des sous-classes de ces classes; autrement dit, il retourne le surrogate de toutes les classes définies dans la base courante. Le deuxième opérateur est équivalent à l'expression Lisp (sortl (mapcar ':$nom (:$classes))), car il retourne dans l'ordre alphabétique les noms de toutes les classes définies dans la base courante.

(:classe? 'nom-classe)

Si dans la base courante il y a une classe ayant le nom spécifié par l'argument, cet opérateur retourne son surrogate; dans le cas contraire il retourne ( ).

(:+instance ['srgt-bc] ['cars]) ;(*voir* section B.4).

crée un nouvel objet instance. Eventuellement, l'utilisateur peut spécifier un nombre quelconque de caractéristiques de cette instance, à l'aide de l'argument cars qui doit être constitué par des couples 'nom-car 'valeur-car, où nom-car est le nom d'une caractéristique et valeur-car est une expression OBMS (donc une expression Lisp) dont l'évaluation retourne la valeur de cette caractéristique.

(:-instance ['srgt-bc] 'srgt-i) ;(*voir* section B.4).

enlève l'instance identifiée par le surrogate srgt-i si elle n'est pas représentative, agrégée ou subordonnée d'un autre objet. Cet opérateur enlève aussi tous les représentatifs ainsi que les subordonnés qui ne sont pas partagés avec d'autres objets. Par ailleurs, le comportement de cet opérateur est celui de l'opérateur :enlever dans l'expression (:enlever 'srgt-i).

(:$instances ['srgt-bc] [nil]) ;(*voir* section B.4).

Si le deuxième argument est fourni par l'utilisateur, cet opérateur retourne la valeur de la propriété "instances" de la base courante, c'est-à-dire une liste contenant le surrogate de toutes les instances qui n'ont été associées à aucune classe. Par contre, lorsque le deuxième argument n'est pas fourni, OBMS utilise des mécanismes d'inférence en vue d'ajouter à cette liste les surrogates des instances qui ont été associées aux objets classe; autrement dit, il retourne le surrogate de toutes les instances définies dans la base courante.

(:instance? ['srgt-bc] ['liste-atts]) ;(*voir* section B.4)

retourne le surrogate de toutes les instances qui n'ont pas été classifiées et qui ont les attributs définis à l'aide de l'argument liste-atts. Le comportement de cet opérateur polymorphique est complètement défini dans la section B.4.

(:+opérateur 'nom-op ['nom-fctn ['args-x]])
(:créer-opérateur nom-op [nom-fctn [args-x]])

créent un nouvel objet opérateur si dans la base courante il n'y a pas un autre opérateur défini par OBMS, ou par les utilisateurs, ayant le nom spécifié par le premier argument. Si l'argument args-x (décrit ci-après) n'est pas fourni, OBMS cherche une fonction Lisp ayant le nom spécifié par l'argument nom-fctn. Si cette recherche réussit, la propriété "source" du nouvel opérateur est affectée avec l'expression Lisp associée à cette fonction; dans le cas contraire, OBMS signale un événement exceptionnel. Si l'argument nom-fctn n'est pas fourni, OBMS utilise par défaut l'argument nom-op.

Si l'argument args-x est fourni, OBMS évalue le prédicat :externes?, défini ci-dessous, afin de vérifier que l'emploi de fonctions externes (c'est-à-dire les fonctions qui ne sont pas écrites en Le_Lisp) est autorisé. Si l'évaluation de ce prédicat ne retourne pas la valeur T (vrai), OBMS signale un événement exceptionnel; dans le cas contraire il essaie d'intégrer dynamiquement à l'environnement Lisp une

fonction externe ayant le nom spécifié par l'argument nom-fctn (cet argument est obligatoire lorsque l'argument args-x est fourni). Si cette intégration peut se réaliser, le nom de la fonction est affectée à la propriété "source" du nouvel opérateur.

La seule différence entre les opérateurs :+opérateur et :créer-opérateur est que le premier évalue ses arguments et retourne le surrogate du nouvel opérateur, tandis que le deuxième n'évalue pas ses arguments et retourne l'argument nom-op.

Remarques:

a) Le nom spécifié par l'argument nom-op doit être une variable du package || de l'environnement Lisp.
b) Lorsque la fonction spécifié par l'argument nom-fctn est interne (c'est-à-dire une fonction Lisp), elle ne doit pas être compilée. D'autre part, lorsque cette fonction est externe, elle doit être un sous-programme ayant le nom spécifié par l'argument nom-fctn et étant stocké dans un fichier (du même nom) du répertoire courant de l'utilisateur[1].
c) args-x doivent être les arguments requis par la fonction DEFEXTERN de Le_Lisp, mais le nom de la fonction externe doit être une variable du package ||, non précédée par le signe "_".
d) Si l'une des conditions précédentes n'est pas satisfaite, OBMS signale un événement exceptionnel.

(:externes?)

retourne T (vrai) si l'éditeur de liens du système d'exploitation permet d'intégrer dynamiquement au système Lisp des modules écrits dans d'autres langages de programmation.

(:-opérateur 'srgt-op)
(:enlever-opérateur nom-op)

enlèvent l'objet opérateur identifié par srgt-op. Leur résultat est égal à celui de l'expression (:enlever 'srgt-op). :enlever-opérateur n'évalue pas son argument (c'est-à-dire le nom de l'opérateur à enlever), et peut être défini comme (:-opérateur (:opérateur? 'nom-op)) ou comme (:enlever (:opérateur? 'nom-op)).

[1]Dans les systèmes UNIX, le répertoire courant est par défaut celui spécifié par la variable $HOME, tandis que dans le système Macintosh ce répertoire est par défaut celui où le système OBMS est stocké.

(:$opérateurs)
(:retrouver-opérateurs)

Le premier opérateur montre le surrogate de tous les objets opérateur définis par les utilisateurs dans la base courante, tandis que le deuxième montre leur nom dans l'ordre alphabétique.

(:opérateur? 'nom-op)

Si dans la base courante les utilisateurs ont défini un opérateur ayant le nom spécifié par l'argument nom-op, :opérateur? retourne son surrogate; dans le cas contraire il retourne ( ).

B.4. Les objets CLASSE

Outre les propriétés communes à tous les objets OBMS, tout objet CLASSE a les propriétés suivantes, dont les opérateurs correspondant sont décrits dans cette section:

- ◊ nom
- ◊ super-classes
- ◊ sous-classes
- ◊ générique
- ◊ représentatifs
- ◊ superordonnés
- ◊ subordonnés
- ◊ instances
- ◊ caractéristiques
- ◊ assertions

(:$nom 'srgt-cl) ;(*voir* sections B.3 et B.6)

montre le nom de l'objet classe identifié par srgt-cl.

(:&nom-classe 'srgt-cl 'nouveau-nom)
(:renommer-classe ancien-nom nouveau-nom)

changent le nom d'un objet classe défini auparavant, si dans la base courante il n'y a pas une autre classe ayant le nom spécifié par le deuxième argument.

(:statut-classe nom-cl)

est un opérateur équivalent à (or (:$statut (:classe? 'nom-cl)) 'non-modifié), c'est-à-dire qu'il retourne «non-modifié» au lieu de la valeur

( ) retournée par :statut lorsque la classe identifiée par srgt-cl n'a pas été modifiée.

(:+super-classe 'srgt-cl 'srgt-sp)
(:ajouter-super-classe nom-classe nom-spcl)

définissent la classe identifiée par srgt-sp comme une super-classe de la classe identifiée par srgt-cl. A l'exception de la valeur retournée, ces opérateurs sont équivalents à (:+sous-classe 'srgt-sp 'srgt-cl). Ils vérifient ainsi le respect des mêmes contraintes d'intégrité qui sont vérifiées par l'opérateur :+sous-classe. La valeur qu'ils retournent est une liste contenant les surrogates (ou les noms, si le deuxième opérateur est utilisé) de toutes les super-classes immédiates de la classe srgt-cl.

(:-super-classe 'srgt-cl 'srgt-sp)
(:enlever-super-classe nom-classe nom-spcl)

enlèvent de la base d'objets courante toute information définissant srgt-sp comme super-classe de srgt-cl. A l'exception de la valeur retournée, ces opérateurs sont équivalents à (:-sous-classe 'srgt-sp 'srgt-cl). Ils vérifient ainsi le respect des mêmes contraintes d'intégrité qui sont vérifiées par l'opérateur :-sous-classe. La valeur qu'ils retournent est une liste contenant les surrogates (ou les noms, si le deuxième opérateur est utilisé) de toutes les super-classes immédiates de srgt-cl.

(:$super-classes 'srgt-cl [nil])
(:retrouver-super-classes nom-classe)
(:retrouver-ancêtres nom-classe)

Si le deuxième argument est fourni, :$super-classes retourne une liste contenant les surrogates des super-classes immédiates de la classe identifiée par srgt-cl. Par contre, lorsque le deuxième argument n'est pas fourni, OBMS utilise des mécanismes d'inférence en vue d'ajouter à cette liste les surrogates des super-classes de ces classes, les surrogates des super-classes de ces super-classes, etc.; autrement dit, il retourne le surrogate de tous les ancêtres de la classe srgt-cl. Le deuxième opérateur est équivalent à l'expression Lisp (sortl (mapcar ':$nom (:$super-classes ( )))), car il retourne dans l'ordre alphabétique les noms de toutes les super-classes immédiates de srgt-cl, tandis que le troisième opérateur est équivalent à l'expression (sortl (mapcar ':$nom (:$super-classes))), car il retourne dans l'ordre alphabétique les noms de toutes les ancêtres de srgt-cl.

(:super-classe? 'srgt-sp 'srgt-cl [nil])
(:ancêtre? nom-anc nom-classe)

Lorsque l'argument facultatif n'est pas fourni par l'utilisateur, l'opérateur :super-classe? retourne soit une liste vide, c'est-à-dire ( ), si la classe identifiée par srgt-sp n'a pas été définie comme une super-classe (immédiate ou pas) de la classe srgt-cl, soit une liste de surrogates qui représente le "chemin d'accès" qui permet de retrouver srgt-sp en analysant la propriété "super-classes" de srgt-cl, de ses super-classes, des super-classes de ces super-classes, etc., ou srgt-cl en analysant la propriété "sous-classes" de srgt-sp, de ses sous-classes, des sous-classes de ces sous-classes, etc.

Par contre, lorsque l'argument facultatif est fourni, OBMS n'utilise pas des mécanismes d'inférence et, en conséquence, l'opérateur :super-classe? retourne une liste vide si srgt-sp n'est pas une sous-classe immédiate de srgt-cl ou, dans le cas contraire, une liste contenant un seul élément: le surrogate srgt-sp.

L'opérateur :ancêtre est un opérateur similaire qui retourne des listes contenant des noms au lieu des surrogates, et qui peut être défini comme (mapcar ':$nom (:super-classe? (:classe? 'nom-anc) (:classe? 'nom-classe))). Remarque: En accord avec la règle d'intégrité cc6, (:super-classe? A A) = (A).

(:+sous-classe 'srgt-cl 'srgt-ss)
(:ajouter-sous-classe nom-classe nom-sscl)

définissent srgt-ss comme une sous-classe de srgt-cl, uniquement si

a) srgt-ss n'est pas un ancêtre (c'est-à-dire une super-classe immédiate ou pas), une classe générique, ou une classe représentative de srgt-cl,
b) l'objet générique de srgt-ss est égal à, ou descendant de, l'objet générique de srgt-cl.

srgt-ss hérite de toutes les propriétés de srgt-cl; par conséquent, srgt-ss ne doit pas avoir des caractéristiques définies dans srgt-cl. Par ailleurs, si l'objet générique de ces classes est le même objet, OBMS enlève de srgt-ss l'information redondante; si srgt-ss a été définie comme sous-classe d'un ancêtre de srgt-cl, OBMS enlève aussi cette information redondante (si srgt-ss est sous-classe de srgt-cl, srgt-ss est aussi sous-classe de tous les ancêtres de srgt-cl). La valeur retournée par ces opérateurs est une liste contenant les surrogates (ou les nom, dans le cas du deuxième opérateur) de toutes les sous-classes immédiates de srgt-cl.

(:-sous-classe 'srgt-cl 'srgt-ss)
(:enlever-sous-classe nom-classe nom-sscl)

enlèvent de la base d'objets courante toute information définissant srgt-ss comme sous-classe de srgt-cl. La valeur retournée est une liste contenant les surrogates (ou les noms, si le deuxième opérateur est utilisé) de toutes les sous-classes immédiates de srgt-cl.

(:$sous-classes 'srgt-cl [nil])
(:retrouver-sous-classes nom-classe)
(:retrouver-descendants nom-classe)

Si le deuxième argument est fourni, :$sous-classes retourne une liste contenant les surrogates des sous-classes immédiates de la classe identifiée par srgt-cl. Par contre, lorsque le deuxième argument n'est pas fourni, OBMS utilise des mécanismes d'inférence en vue d'ajouter à cette liste les surrogates des sous-classes de ces classes, les surrogates des sous-classes de ces sous-classes, etc.; autrement dit, il retourne le surrogate de tous les descendants de la classe srgt-cl. Le deuxième opérateur est équivalent à l'expression Lisp (sortl (mapcar ':$nom (:$sous-classes ( )))), car il retourne dans l'ordre alphabétique les noms de toutes les sous-classes immédiates de srgt-cl, tandis que le troisième opérateur est équivalent à l'expression (sortl (mapcar ':$nom (:$sous-classes))), car il retourne dans l'ordre alphabétique les noms de toutes les descendants de srgt-cl.

(:sous-classe? 'srgt-ss 'srgt-cl [nil])
(:descendant? nom-desc nom-classe)

les résultats retournés par ces opérateurs sont égaux à ceux retournés par les opérateurs :super-classe? et :ancêtre? dans les expression (:super-classe? 'srgt-cl 'srgt-ss [nil]) et (:ancêtre? nom-classe nom-desc), respectivement.

(:généalogie nom-classe)

retourne le nom des ancêtres et des descendants de l'objet classe ayant le nom spécifié par l'argument.

(:+générique 'srgt-cl 'srgt-g) ;(*voir* section B.5)
(:ajoute-générique nom-classe nom-gén)

définissent srgt-g comme objet générique de srgt-cl. Les contraintes d'intégrité vérifiées par ces opérateurs sont celles vérifiées lorsque l'on utilise l'expression (:+représentatif 'srgt-g 'srgt-cl). Par ailleurs,

les deux opérateurs retournent son première argument comme résultat.

(:-générique 'srgt-cl) ;(*voir* section B.5)
(:enlever-générique nom-classe)

enlèvent de la base d'objets l'association entre la classe identifiée par srgt-cl et son objet générique immédiat, si cette classe en a un.

(:$générique 'srgt-cl [nil]) ;(*voir* section B.5)
(:retrouver-générique nom-classe [( )])

Si le deuxième argument est fourni par l'utilisateur, ces opérateurs retournent seulement le surrogate (ou le nom, si :retrouver-générique est utilisé) de l'objet générique immédiat de la classe identifiée par srgt-cl, si elle en a un. Lorsqu'il n'y a qu'un argument, OBMS utilise des mécanismes d'inférence pour ajouter a ce résultat le surrogate (ou le nom) des génériques de tous les ancêtres de srgt-cl, et celui des génériques de ces génériques. Le deuxième opérateur retourne la liste dans l'ordre alphabétique.

(:+représentatif 'srgt-cl 'srgt-r) ;(*voir* section B.5)
(:ajouter-représentatif nom-classe nom-rep)

définissent srgt-r comme une classe représentative de srgt-cl, uniquement si

a) srgt-r n'est ni ancêtre ni descendant de srgt-cl,
b) srgt-r n'a pas d'objet générique immédiat, et son générique non immédiat est nul ou un ancêtre de srgt-cl,
c) srgt-r n'a pas d'attributs définis aussi dans la classe srgt-cl.

(:-représentatif 'srgt-cl 'srgt-r) ;(*voir* section B.5)
(:enlever-représentatif nom-classe nom-rep)

enlèvent de la base d'objets toute information définissant srgt-r comme un représentatif de srgt-cl, et retournent une liste avec les surrogates (ou les noms, dans le cas du deuxième opérateur) des représentatifs immédiats de srgt-cl.

(:$représentatifs 'srgt-cl [nil]) ;(*voir* section B.5)
(:retrouver-représentatifs nom-classe [( )])

Si le deuxième argument est fourni par l'utilisateur, ces opérateurs retournent une liste contenant le surrogate (ou le nom, si :retrouver-représentatifs est utilisé) des objets représentatifs immédiats de la

classe identifiée par srgt-cl. Lorsqu'il n'y a qu'un argument, OBMS utilise des mécanismes d'inférence pour ajouter a ce résultat le surrogate (ou le nom) des représentatifs de tous les descendants de srgt-cl, et celui des représentatifs de ces représentatifs. Le deuxième opérateur retourne la liste dans l'ordre alphabétique.

(:+**superordonné** 'srgt-cl 'srgt-sp)
(:**ajouter-superordonné** nom-classe nom-sp)

définissent srgt-sp comme un objet superordonné de srgt-cl. Le comportement de ces opérateurs est égal à celui de l'opérateur :+subordonné, dans l'expression (:+subordonné 'srgt-sp 'srgt-cl), mais la valeur qu'ils retournent est une liste contenant les surrogates (ou les noms, si le deuxième opérateur est utilisé) de tous les superordonnés immédiats de srgt-cl.

(:-**superordonné** 'srgt-cl 'srgt-sp)
(:**enlever-superordonné** nom-classe nom-sp)

enlèvent de la base d'objets courante toute information définissant srgt-sp comme un superordonné .de srgt-cl. Le comportement de ces opérateurs est égal à celui de l'opérateur :-subordonné, dans l'expression (:-subordonné 'srgt-sp 'srgt-cl), mais la valeur qu'ils retournent est une liste contenant les surrogates (ou les noms, si le deuxième opérateur est utilisé) de toutes les superordonnés immédiats de srgt-cl.

(:$**superordonnés** 'srgt-cl [nil]) ;(*voir* section B.5)
(:**retrouver-superordonnés** nom-classe [( )])

Si le deuxième argument est fourni, ces opérateurs retournent une liste contenant les surrogates (ou les noms, si le deuxième opérateur est utilisé) des superordonnés immédiats de srgt-cl. Par contre, lorsque le deuxième argument n'est pas fourni, OBMS utilise des mécanismes d'inférence en vue d'ajouter à cette liste les surrogates (ou les noms) des superordonnés des ancêtres de srgt-cl, et des superordonnés de tous ces superordonnés. Le deuxième opérateur retourne la liste dans l'ordre alphabétique.

(:+**subordonné** 'srgt-cl 'srgt-sb)
(:**ajouter-subordonné** nom-classe nom-sb)

définissent srgt-sb comme un objet subordonné de srgt-cl, et retournent une liste contenant le surrogate (ou le nom, dans le cas du deuxième opérateur) des subordonnés immédiats de srgt-cl.

(:-subordonné 'srgt-cl 'srgt-sb)
(:enlever-subordonné nom-classe nom-sb)

enlèvent de la base d'objets toute information définissant srgt-sb comme un subordonné de srgt-cl, et retournent une liste contenant le surrogate (ou le nom, dans le cas du deuxième opérateur) des subordonnés immédiats de srgt-cl.

(:$subordonnés 'srgt-cl [nil]) ;(*voir* section B.5)
(:retrouver-subordonnés nom-classe [( )])

Si le deuxième argument est fourni, ces opérateurs retournent une liste contenant les surrogates (ou les noms, si le deuxième opérateur est utilisé) des subordonnés immédiats de srgt-cl. Par contre, lorsque le deuxième argument n'est pas fourni, OBMS utilise des mécanismes d'inférence en vue d'ajouter à cette liste les surrogates (ou les noms) des subordonnés des descendants de srgt-cl, et des subordonnés de tous ces subordonnés. Le deuxième opérateur retourne la liste dans l'ordre alphabétique.

(:+instance 'srgt-cl 'srgt-i) ;(*voir* section B.3)

définit srgt-i comme une instance de srgt-cl, uniquement si les caractéristiques de srgt-i vérifient toutes les contraintes définies par les descripteurs de srgt-cl, et si srgt-i n'hérite pas (de ses autres classes) des attributs qui sont déjà définis par srgt-cl. De plus, si srgt-i a été définie comme une instance d'un ancêtre de srgt-cl, OBMS enlève cette information redondante, du fait que toutes les instances de srgt-cl sont aussi des instances de ses ancêtres. Le résultat retourné par cet opérateur est la liste des objets classifiés dans la classe srgt-cl.

(:@instance 'srgt-cl ['cars])
(:créer-instance nom-classe [cars])

Lorsqu'il n'y a pas d'événements exceptionnels, le comportement de ces opérateurs est équivalent à (:+instance 'srgt-cl (:+instance ['cars])); c'est-à-dire, ils créent un objet instance et essaient de le classifier dans srgt-cl. Toutefois, si l'utilisateur fournit le deuxième argument afin de définir des caractéristiques de la nouvelle instance, et si ces caractéristiques ne satisfont pas toutes les contraintes définies par srgt-cl, la nouvelle instance est détruite et un événement exceptionnel est signalé. Dans le cas contraire, le résultat retourné par ces opérateurs

est le surrogate (ou la description[1], dans le cas du deuxième opérateur) de la nouvelle instance.

(:instance-valide? 'srgt-i 'srgt-cl)

retourne srgt-i si l'objet identifié par ce surrogate a été classifié dans srgt-cl, ou si l'utilisateur peut le faire sans corrompre l'intégrité sémantique de la base d'objets.

(:-instance 'srgt-cl 'srgt-i) ;(*voir* section B.3)

enlève de la base d'objets toute information définissant srgt-i comme une instance de srgt-cl, et retourne comme résultat la liste des objets classifiés dans la classe srgt-cl.

(:$instances 'srgt-cl [nil]) ;(*voir* section B.3)
(:retrouver-instances nom-classe [( )])

Lorsque le deuxième argument est fourni par l'utilisateur, ces opérateurs retournent une liste contenant le surrogate (ou les descriptions, dans le cas du deuxième opérateur) de toutes les instances immédiates de srgt-cl. Par contre, lorsqu'ils son invoqués avec un seul argument, OBMS utilise des mécanismes d'inférence pour ajouter à cette liste le surrogate (ou la description) des instances des descendants de srgt-cl, et des représentatifs de toutes ces instances.

(:est-un? 'srgt-i 'srgt-cl)

retourne ( ) si srgt-i n'est ni instance de la classe srgt-cl (dans le cas contraire il retourne l'argument srgt-cl), ni instance de l'une des sous-classes ou des représentatifs de srgt-cl (autrement il retourne le surrogate de cette classe).

(:est-un-XX? 'srgt-i)

représente un ensemble de prédicats définis de manière automatique par OBMS lors de la création d'objets classe. XX représente le nom d'une classe. Si l'utilisateur crée, par exemple, la classe Projet, OBMS défini le prédicat ":est-un-Projet?", tel que (:est-un-Projet? 'srgt-i) est équivalent à (:est-un? 'srgt-i (:classe? 'Projet)). Cette fonctionnalité facilite la définition de descripteurs-f (*voir ci-après*).

---

[1]La description d'un objet instance contient le nom et la valeur de toutes ses caractéristiques, et peut être obtenue en utilisant l'expression Lisp (eval 'srgt-i).

(:instance? 'srgt-cl ['liste-atts]) ;(*voir* section B.3)

retourne le surrogate de toutes les instances qui ont été classifiées dans srgt-cl et qui ont les attributs spécifiés à travers l'argument liste-atts. Cet argument doit être une collection de listes de la forme (att ['op] 'val). Par exemple, l'expression (:instance? (:classe? 'Ordinateur) '(syst_exploit '= "UNIX") '(périphériques ':contient 'imprimante)) retourne le surrogate de tous les ordinateurs ayant le système d'exploitation UNIX et une imprimante parmi leurs périphériques.

L'élément "op" de ces listes est facultatif, et sa valeur par défaut est "=". Ainsi, (syst_exploit '= "UNIX") est identique à (syst_exploit "UNIX"). Tout de même, lorsqu'il est fourni, il est évalué et son évaluation doit retourner le nom d'un opérateur binaire ou une lambda expression (Lisp) équivalente, qui aura att et val comme arguments. Dans notre exemple précédent, la liste (périphériques ':contient 'imprimante) est équivalente à la liste (périphériques '(lambda (x y) (car (member y x))) 'imprimante).

L'élément "val" est aussi évalué. Par conséquent, l'utilisateur peut employer des expressions telles que (salaire '= (* 30 (+ 527 234))). D'autre part, l'élément att peut être le nom d'un attribut interne (*voir* section B.5). Pour cette raison, l'expression (:instance? (:classe? 'Ordinateur) '(#:nom:responsable:projet "Paul")) peut être utilisée pour retrouver les ordinateurs conçus dans le Projet dont le nom du responsable est Paul.

Finalement, si l'argument liste-atts n'est pas fourni, l'opérateur :instance? retourne le surrogate des instances de srgt-cl qui n'ont pas encore d'attributs définis.

(:+caractéristique-f 'srgt-cl 'nom-car 'filtre)
(:ajouter-caractéristique-f nom-classe nom-car filtre)

définissent une nouvelle caractéristique filtrée (*Cf.* section 5 du chapitre 3) de l'objet classe identifié par srgt-cl, si le filtre est le nom d'une fonction ou d'un opérateur unaire défini auparavant, ou une liste dont le premier élément est une lambda expression ou le nom d'une fonction ou d'un opérateur n-aire, et les autres éléments sont des éventuels arguments nécessaires pour invoquer la fonction, l'opérateur ou la lambda expression.

Lors du processus de filtrage des caractéristiques des instances de srgt-cl, ce filtre (ou son premier élément, dans le cas d'une liste) sera invoqué en utilisant comme premier argument chacune des éléments constituant la valeur de la caractéristique dénotée par nom-car, et

comme deuxième, troisième,... et n-ième argument tous les autres éléments du filtre, qui seront évalués au moment de cette invocation.

Les expressions suivantes sont équivalentes et illustrent l'emploi du prédicat :est-un?, décrit ci-dessus, et toutes les deux pourraient être utilisée en vue de définir, par exemple, la caractéristique "concepteur" de la classe Ordinateur:

a) (:ajouter-caractéristique-f Ordinateur
concepteur :est-un-Ingénieur?)
b) (:ajouter-caractéristique-f Ordinateur concepteur
(lambda (ing) (:est-un? ing (:classe? 'Ingénieur))))

(:+caractéristique-c 'srgt-cl 'nom-car 'fonction)
(:ajouter-caractéristique-c nom-classe nom-car fonction)

définissent une nouvelle caractéristique calculée (*Cf.* section 7 du chapitre 3) de l'objet classe identifié par srgt-cl, si fonction est le nom d'une fonction ou d'un opérateur unaire défini auparavant, ou une liste dont le premier élément est une lambda expression ou le nom d'une fonction ou d'un opérateur n-aire, et les autres éléments sont les n arguments nécessaires pour invoquer la fonction, l'opérateur ou la lambda expression. Ces arguments seront évalués au moment de cette invocation.

(:+caractéristique-p 'srgt-cl 'nom-car 'valeur)
(:ajouter-caractéristique-p nom-classe nom-car valeur)

définissent une nouvelle caractéristique préétablie (*Cf.* section 7 du chapitre 3) de l'objet classe identifié par srgt-cl. Le résultat de ces opérateurs est équivalent au résultat de (:+caractéristique-c 'srgt-cl 'nom-car '(:identité 'valeur)).

(:-caractéristique 'srgt-cl 'nom-car) ;(*voir* section B.5)
(:enlever-caractéristique nom-classe nom-car)

enlèvent de la classe identifiée par srgt-cl la caractéristique spécifiée par l'argument nom-car.

(:$caractéristiques 'srgt-cl ['noms-cars]) ;(*voir* section B.5)
(:retrouver-caractéristiques nom-classe [noms-cars])

Si l'argument facultatif n'est pas fourni pas l'utilisateur, ces opérateurs retrouvent toutes les caractéristiques de l'objet classe identifié par srgt-cl. Dans le cas contraire, ils ne retournent que les caractéristiques

spécifiées à l'aide de l'argument facultatif noms-cars, qui représente un nombre quelconque de noms de caractéristiques.

(:$attributs 'srgt-cl ['noms-atts]) ;(*voir* section B.5)
(:retrouver-attributs nom-classe [noms-atts])

Lorsque ces opérateurs sont invoqués avec un seul argument, ils retrouvent tous les attributs de l'objet classe identifié par srgt-cl, y compris ceux dont cette classe hérite de ses ancêtres et de son objet générique. Dans le cas contraire, ces opérateurs retournent uniquement les attributs spécifiés par l'argument noms-atts qui représente un nombre quelconque de noms d'attributs.

(:filtre 'srgt-cl 'nom-car ['préd])

redéfinit (si le troisième argument est fourni) ou enlève le prédicat qui doit être utilisé par OBMS pour filtrer la caractéristique nom-car des instances de la classe srgt-cl.

(:cardinalités 'srgt-cl 'nom-car ['(min max)])

redéfinit (si l'argument facultatif est fourni) ou montre les contraintes sur le nombre minimal et maximal d'éléments qui doivent constituer la valeur de la caractéristique nom-car des instances de la classe srgt-cl.

Les valeurs possibles des éléments min et max sont ( ), *, ou des entiers positifs (max > 0). ( ), *, et 0 signifient qu'il n'y a pas de contraintes. D'autre part, si min et max sont tous les deux des entiers, min doit être plus petit ou égal à max.

Des exemples des valeurs possibles pour cet argument sont '(( ) ( )), '(* ( )), '(* *), '(* ( )), '(0 *), '(1 1), et '(0 234). Les quatre premières valeurs sont identiques, la cinquième spécification indique que la caractéristique en question doit être monovaluée et que sa valeur ne doit pas être nulle. La dernière spécification indique que la valeur de la caractéristique peut être nulle ou peut contenir jusqu'à 234 éléments.

(:ensemble 'srgt-cl 'nom-car ['non-dupliqués?])

Si l'argument facultatif n'est pas fourni, cet opérateur montre la propriété "ensemble" du descripteur de la caractéristique nom-car de la classe srgt-cl (*Cf.* section 5 du chapitre 3); autrement, OBMS essaie de affecter à la valeur de cette propriété la valeur booléen (T ou ( )) de l'argument "non-dupliqués?". Le processus est avorté seulement lorsque cette valeur est T et il y a au moins une instance de srgt-cl ayant des éléments dupliqués dans la valeur de sa propriété nom-car.

(:+assertion 'srgt-cl 'filtre)

définit une nouvelle assertion sur la classe srgt-cl, si le filtre est le nom d'une fonction ou d'un opérateur unaire défini auparavant, ou une liste dont le premier élément est une lambda expression ou le nom d'une fonction ou d'un opérateur n-aire, et les autres éléments sont des éventuels arguments nécessaires pour invoquer la fonction, l'opérateur ou la lambda expression.

Lors de la classification d'une nouvelle instance dans la classe srgt-cl, ou la modification d'une propriété quelconque des instances de cette classe, ce filtre (ou son premier élément, s'il est une liste) sera invoqué en utilisant comme premier argument le surrogate srgt-cl, et comme deuxième, troisième,... et n-ième argument tous les autres éléments du filtre, qui seront évalués au moment de cette invocation.

Par exemple, l'expression (:+assertion (:classe? 'Ordinateur) '((lambda (cl) (<= 100 (length (:$instances cl)))))) peut être utilisée pour spécifier que la classe Ordinateur ne doit jamais avoir plus de 100 instances.

(:-?assertion 'srgt-cl 'index)

enlève la index^{ième} assertion définie sur la classe identifiée par srgt-cl. index doit être un entier plus grand que zéro, et plus petit au égal au nombre maximal d'assertions dans la classe en question.

(:$assertions 'srgt-cl [nil])
(:retrouver-assertions nom-classe [( )])

Si le deuxième opérateur est fourni par l'utilisateur, ces opérateurs retournent une liste contenant toutes les assertions qui ont été directement définies sur la classe srgt-cl. Par contre, lorsqu'ils sont invoqués avec un seul argument, OBMS utilise des mécanismes d'inférence pour ajouter à cette liste toutes les assertions définies sur les ancêtres et les objets génériques de srgt-cl.

(:tester-assertion 'prédicat 'srgt-cl ['srgt-i])

retourne ( ) si la classe srgt-cl ne satisfait pas le prédicat dénoté par le premier argument. Lorsque l'argument facultatif (qui doit être le surrogate d'un objet instance) est fourni, le test est réalisé en classifiant (temporairement) l'instance srgt-i dans la classe srgt-cl. Cet opérateur peut être utilisé pour tester si la classification d'un objet dans une classe donnée peut se réaliser sans corrompre l'intégrité sémantique de la

base d'objets. En fait, OBMS utilise cet opérateur avant de classifier n'importe quelle instance.

(:clé? 'srgt-cl ['noms-cars])

retourne T uniquement si nom-cars sont les propriétés clé de la classe srgt-cl, c'est-à-dire si les listes de valeurs correspondant aux caractéristiques nom-cars des instances de la classe srgt-cl constituent un ensemble. Par exemple, l'expression (:+assertion (:classe? 'Projet) '(:clé? 'nom)) spécifie que la propriété "nom" de la classe Projet est une propriété clé et, par conséquent, deux projets distincts ne doivent pas avoir le même nom.

(:décrire-classe nom-classe)

est un opérateur équivalent à (eval (:classe? 'nom-classe)), qui montre les propriétés "les plus importantes" de la classe ayant le nom spécifié par l'argument.

(:classep 'srgt)

retourne son argument s'il est un surrogate généré par le système pour identifier un objet classe.

B.5. Les objets INSTANCE

Outre les propriétés communes à tous les objets OBMS, tout objet INSTANCE a les propriétés suivantes, dont les opérateurs correspondant sont décrits dans cette section:

- ◊ classes (où cette instance a été classifiée)
- ◊ générique
- ◊ représentatifs
- ◊ superordonnés
- ◊ subordonnés
- ◊ agrégats
- ◊ caractéristiques

(:+classe 'srgt-i 'srgt-cl) ;(*voir* section B.3)

ajoute la classe srgt-cl a la propriété "classes" de l'instance srgt-i. Le comportement de cet opérateur est équivalent à celui de l'opérateur :+instance dans l'expression (:+instance 'srgt-cl 'srgt-i), mais la valeur retournée est la liste de classes de srgt-i.

(:-classe 'srgt-i 'srgt-cl) ;(*voir* section B.3)

enlève la classe srgt-cl de la propriété "classes" de l'instance srgt-i. Le comportement de cet opérateur est équivalent à celui de :-instance dans l'expression (:-instance 'srgt-cl 'srgt-i), mais la valeur retournée est la liste de classes de srgt-i.

(:$classes 'srgt-i [nil]) ;(*voir* section B.3)

Si le deuxième argument est fourni, cet opérateur retourne une liste contenant les surrogates de toute les classe immédiates de srgt-i. Par contre, s'il est invoque avec un seul argument, OBMS utilise des mécanismes d'inférence pour ajouter à cette liste le surrogate des ancêtres de srgt-cl, et celui des objets génériques de toutes ces classes.

(:$agrégats 'srgt-i [nil])

Si l'argument facultatif est fourni par l'utilisateur, cet opérateur retourne une liste contenant les surrogates de tous les agrégats immédiats de srgt-i, c'est-à-dire des objets ayant srgt-i comme un élément de la valeur de l'une de leurs caractéristiques. Lorsque l'opérateur est invoqué avec un seul argument, OBMS utilise des mécanismes d'inférence pour ajouter à cette liste le surrogate des agrégats des objets génériques de srgt-i (y compris les génériques non immédiats), et celui des agrégats de tous ces agrégats.

(:$superordonnés 'srgt-i [nil]) ;(*voir* section B.4)

Si le deuxième argument est fourni par l'utilisateur, cet opérateur retourne une liste contenant les surrogates de tous les superordonnés immédiats de srgt-i. Lorsque l'opérateur est invoqué avec un seul argument, OBMS utilise des mécanismes d'inférence pour ajouter à cette liste le surrogate des superordonnés des objets génériques de srgt-i (y compris les génériques non immédiats), et celui des superordonnés de tous ces superordonnés.

(:$subordonnés 'srgt [nil]) ;(*voir* section B.4)

Si l'argument facultatif est fourni, cet opérateur retourne une liste contenant les surrogates de tous les subordonnés immédiats de srgt-i. Lorsque l'opérateur est invoqué avec un seul argument, OBMS utilise des mécanismes d'inférence pour ajouter à cette liste le surrogate des subordonnés des objets représentatifs de srgt-i (y compris les représentatifs non immédiats), et celui des subordonnés de tous ces subordonnés.

(:+générique 'srgt-i 'srgt-r) ;(*voir* section B.4)

définit srgt-g comme un objet générique de srgt-i. Le comportement de cet opérateur est équivalent à celui de l'opérateur :+représentatif dans l'expression (:+représentatif 'srgt-g 'srgt-i), mais la valeur retournée est srgt-r.

(:-générique 'srgt-i) ;(*voir* section B.4)

enlève de la base d'objets l'association entre l'objet identifié par srgt-i et son objet générique immédiat, si srgt-i en a un. Lorsque srgt-i a un générique immédiat, le comportement de cet opérateur est équivalent à celui de l'opérateur :-représentatif dans l'expression (:-représentatif (:$générique 'srgt-i ( )) 'srgt-i), mais la valeur retournée est le surrogate de l'objet générique de srgt-i.

(:$génériques 'srgt-i [nil]) ;(*voir* section B.4)

Lorsque l'argument facultatif est fourni par l'utilisateur, cet opérateur retourne une liste contenant le surrogate de l'objet générique immédiat de srgt-i, si celui-ci en a un. Par contre, si l'opérateur est invoqué avec un seul argument, OBMS utilise des mécanismes d'inférence pour ajouter à cette liste le surrogate de l'objet générique immédiat du générique de srgt-i, et ainsi de suite.

(:+représentatif 'srgt-i 'srgt-r) ;(*voir* section B.4)

définit srgt-r comme un représentatif de srgt-i, seulement si srgt-r n'a ni un objet générique ou superordonné, ni d'attributs déjà définis dans l'objet srgt-i.

(:représentatif-valide? 'srgt-i 'srgt-r)

retourne T seulement si srgt-r est une instance d'une classe représentative de srgt-i (c'est-à-dire une classe étant l'objet représentatif de la classe ou des classes de srgt-i), et si srgt-r n'a pas d'attributs déjà définis dans l'objet srgt-i.

(:-représentatif 'srgt-i 'srgt-r) ;(*voir* section B.4)

indique au système que l'objet srgt-r n'est plus représentatif de l'objet srgt-i. Le résultat retourné par cet opérateur est la liste des représentatifs de srgt-i.

(:$représentatifs 'srgt-i [nil]) ;(*voir* section B.4)

Lorsque le deuxième argument est fourni, cet opérateur retourne une liste contenant le surrogate des représentatifs immédiats de srgt-i. D'autre part, si l'opérateur est invoqué avec un seul argument, OBMS utilise des mécanismes d'inférence pour ajouter à cette liste le surrogate de tous les représentatifs immédiats de ces représentatifs, et ainsi de suite.

(:+caractéristique 'srgt-i 'nom-car 'valeur)

définit une nouvelle caractéristique de srgt-i s'il n'y a pas de contraintes sur cette caractéristique ou si elle les satisfait toutes.

(:-caractéristique 'srgt-i 'nom-car) ;(*voir* section B.4)

enlève la caractéristique nom-car de l'objet srgt-i si dans cet état l'objet satisfait toutes les contraintes d'intégrité définies par les descripteurs de toutes ses classes.

(:&caractéristique 'srgt-i 'nom-car ['val])

Si nom-car dénote une caractéristique de srgt-i ayant une valeur non nulle, cet opérateur met à jour cette valeur en utilisant la valeur de l'argument val. Si nom-car n'a pas été définie (c'est-à-dire si sa valeur est nulle), cette opérateur invoque l'opérateur :+caractéristique en utilisant les trois arguments. Si l'argument val n'est pas fourni pas l'utilisateur, alors l'opérateur :&caractéristique invoque l'opérateur :-caractéristique en utilisant les deux arguments fournis. Cet opérateur peut donc ajouter, enlever et mettre à jour les caractéristiques des objets instance. Tout de même, son principale utilisation est celle de la mise à jour, particulièrement dans le cas des caractéristiques dont les cardinalités sont (1 1).

(:$caractéristiques 'srgt-i ['noms-cars]) ;(*voir* section B.4)

Si l'argument facultatif n'est pas fourni pas l'utilisateur, cet opérateur retrouve le nom et la valeur de toutes les caractéristiques de l'objet identifié par srgt-i. Dans le cas contraire, il ne retourne que les caractéristiques spécifiées à l'aide de l'argument facultatif noms-cars, qui représente un nombre quelconque de noms de caractéristiques.

(:$caractéristique-interne 'nom-interne 'srgts-i)

retourne la valeur de la caractéristique dénotée par l'argument nom-interne dans toutes les instances spécifiées par l'argument srgts-i, qui

représente un nombre quelconque de surrogates. D'autre part, l'argument nom-interne peut être structuré (*Cf.* section 8 du chapitre 3).

Dans l'implémentation actuelle d'OBMS les argument non structurés doivent être des variables Le_Lisp appartenant au package ||, tandis que les arguments structurés sont représentés par des variables des autres packages. Par exemple, abc, x et r2-NN son des arguments non structurés, tandis que #:n:abc et #:nsr:abc:xyz:r2-NN sont structurés. Une expression telle que (:caractéristique-interne '#:x:y '#{55} '#{89}) retournera la valeur de la caractéristique "x" de tous les objets constituant la valeur de la caractéristique "y" des objets identifiés par les surrogates #{55} et #{89}.

(:$attributs 'srgt-i [noms-atts]) ;(*voir* section B.4)

Lorsque cet opérateur est invoqué avec un seul argument, il retrouve le nom et la valeur de tous les attributs de l'objet identifié par srgt-i, y compris ceux dont cet objet hérite de ses classes et de son objet générique. Dans le cas contraire, cet opérateur retourne uniquement les attributs spécifiés par l'argument noms-atts qui représente un nombre quelconque de noms d'attributs.

(:$attribut-interne 'nom-interne 'srgts-i)

est un opérateur similaire à :$caractéristique-interne, mais la recherche inclut l'analyse des propriétés dont srgt-i hérité de ses classes et de son objet génériques, et des propriétés dont les composants de srgt-i héritent des leurs classes et de leurs objets génériques, dans le cas où l'argument nom-interne soit structure.

(:cyclique? 'srgt1 'nom-att 'srgt2)

retourne T (vrai) si srgt1 et srgt2 sont égaux, ou si srgt1 est inclus dans la valeur de l'attribut nom-att de l'objet srgt2, ou de l'attribut nom-att de n'importe quel objet constituant la valeur de l'attribut nom-att de l'objet srgt2, etc. Cet opérateur peut donc être utilisé pour contraindre aussi bien les spécifications cycliques que les spécifications récursives (*Cf.* section 5 du chapitre 4).

(:retrouver 'atts 'srgt ['liste-atts])

combine certaines caractéristiques des opérateurs :instance? et :$attribut-interne. Il retourne le surrogate et les attributs (spécifiés par l'argument atts) de toutes les instances ayant les attributs spécifiés par l'argument liste-atts.

L'évaluation de l'argument atts doit retourner soit un seul élément soit une liste d'éléments. Dans le deux cas ces éléments peuvent être structurés et, ainsi, la recherche inclut l'analyse d'attributs internes. Par ailleurs, cette liste peut être vide et, dans ce cas, l'opérateur :retrouver ne retourne que le surrogate des objets satisfaisant les spécifications de l'argument liste-atts-internes.

D'autre part, l'argument srgt doit être le surrogate de la base courante ou celui d'un objet classe. Dans le premier cas, la recherche est réalisée dans toutes les instances qui ne sont associées à aucune classe; dans le deuxième, la recherche est réalisée dans toutes les instances (non nécessairement immédiates) de la classe identifiée par ce surrogate.

Finalement, si le troisième argument n'est pas fourni, cet opérateur (comme l'opérateur :instance?) retourne le surrogate des instances qui n'ont pas encore d'attributs définis.

(:lui-même)

est un opérateur spécial qui retourne le surrogate de l'instance qui est en train d'être analysée par OBMS lors de la recherche de réponses aux requêtes, ou lors de la vérification des contraintes d'intégrité définies par les descripteurs des classes. Par conséquent, cet opérateur peut être utilisé dans la spécification de telles requêtes et de ces contraintes, ainsi que dans la spécification de caractéristiques calculées.

Dans la section 5 du chapitre 4 nous avons présenté un exemple dans lequel l'opérateur :lui-même est utilisé pour spécifier une contrainte d'intégrité indiquant que les systèmes électroniques ne peuvent pas être leurs propres composants. Des exemples de l'emploi de cet opérateur pour la formulation de requêtes, et pour la spécification de caractéristiques calculées, sont les suivants:

a) Retrouver tous les ordinateurs dont le concepteur est en même temps le responsable du projet auquel l'ordinateur en question est associé (*Cf.* figures 1 et 2 du chapitre 2):

```
(:retrouver, ( ) (:classe? 'Ordinateur)
   '(concepteur  '=
      (:$caractéristique-interne
         '#:responsable:projet (:lui-même))) )
```

b) La structure des circuits en cours de conception peut être définie comme une caractéristique calculée dont la fonction de son descripteur-c retourne soit le nom du circuit, lorsque celui-ci n'a

pas de composants, soit une liste contenant les noms de ses sous-composants les plus internes:

```
(:ajouter-caractéristique-c  Circuit structure
   (let ((comp (cadar (:$caractéristiques
                                (:lui-même) 'composants))))
      (ifn comp
         (cadar (:$caractéristiques (:lui-même) 'nom)
         (mapcan
            '(lambda (c) (:$attributs c 'structure))
            comp))) )
```

(:instancep 'srgt)

retourne son argument s'il est un surrogate généré par le système pour identifier un objet instance.

B.6. Les objets OPERATEUR

Outre les propriétés communes à tous les objets OBMS, tout objet OPERATEUR a les propriétés suivantes:

- ◊ nom
- ◊ préconditions
- ◊ source
- ◊ code

Les opérateurs OBMS qui permettent l'accès à ces propriétés sont décrits dans cette section.

(:$nom 'srgt-op) ;(*voir* sections B.3 et B.4)

montre le nom de l'objet opérateur identifié par srgt-op.

(:$type 'srgt-op)
(:retrouver-type nom-op)

retournent le type de l'opérateur identifié par srgt-op. Ce type est «interne», si le source de l'opérateur est une fonction Lisp, ou «externe» dans le cas contraire.

(:+précondition 'srgt-op 'préd)

définit une nouvelle précondition concernant l'exécution de l'opérateur srgt-op, si préd est le nom d'une fonction ou d'un

opérateur unaire défini auparavant, ou une liste dont le premier élément est une lambda expression, une fonction ou un opérateur n-aire, et les autres éléments sont les n arguments nécessaires pour invoquer la fonction, l'opérateur ou la lambda expression.

(:-?précondition 'srgt-op 'index)

enlève la index^{ième} précondition définie sur l'opérateur srgt-op. index doit être un entier plus grand que zéro, et plus petit ou égal au nombre maximal de préconditions de cet opérateur.

(:préconditions 'srgt-op [nil])

montre toutes les préconditions de l'opérateur identifié par le surrogate srgt-op, ou les enlève toutes si l'argument facultatif est fourni.

(:$source 'srgt-op)
(:retrouver-source nom-op)

Si srgt-op est un opérateur dont le type est «interne», ces opérateurs retournent l'expression Lisp constituant la valeur de la propriété "source" de srgt-op; autrement, ils retournent le nom de la fonction externe correspondant.

(:$code 'srgt-op)
(:retrouver-code nom-op)

Si srgt-op est un opérateur dont le type est «interne», et si cet opérateur a été compilé, les opérateurs :$code et :retrouver-code retournent le code LAP résultat de cette compilation (*Cf.* [Chai 86]).

(:décrire-opérateur nom-op)

est un opérateur équivalent à (eval (:opérateur? 'nom-op)), qui montre les propriétés "les plus importantes" de l'opérateur ayant le nom spécifié par l'argument.

(:opérateurp 'srgt)

retourne son argument s'il est un surrogate généré par le système pour identifier un objet opérateur.

(:comp 'srgt-op)
(:compile nom-op)

compilent le source de l'opérateur identifié par srgt-op, si cela n'a pas été fait et si le type de l'opérateur est «interne».

(:compilép 'srgt-op)

retourne son argument si srgt-op est un opérateur dont le type est «interne», et si son source a été compilé.

(:exécuter 'srgt-op ['arguments])

invoque l'opérateur identifié par srgt-op, avec les arguments fournis par l'utilisateur, seulement si toutes les préconditions de srgt-op ont été vérifiées. Remarque: OBMS permet aux utilisateurs d'invoquer les objets opérateur de la même manière qu'ils peuvent invoquer tous les opérateurs OBMS, c'est-à-dire en construisant une liste dont le premier élément est le nom de l'opérateur en question (précédé par le signe ":"), et les autres éléments sont les arguments de cet opérateur. Ainsi, l'expression (:abc 'x 'y) est tout à fait équivalente à l'expression (:exécuter (:opérateur? 'abc) 'x 'y).

INDEX

Abstraction
    des données, 12
    mécanismes d', 7
agrégation, 41
assertions, 79
associations
    composant-de, 41
    instance-de, 43
    représentatif-de, 92
    spécialisation-de, 43
attachement procédural, 9, 67
attributs, 41

Base
    de connaissances, 8, 11
    de données, 8
    d'objets, 14, 38

Caractéristiques, 44
    calculées, 69
    filtrées, 65
    préétablies, 69
classe, 13, 42
    sous-, 43
    super-, 43
contraintes d'intégrité, 28

Descripteurs, 67
données
    base de, 8
    indépendance des, 7

Filtrage, 66

Généralisation, 42

Instance, 13, 42

Mécanismes
    d'abstraction, 7
    de raisonnement, 11
messages, envoi de, 13
méthodes, 13
modèle(s)
    conceptuel, 8
    de données, 7
    sémantiques, 5
modélisation conceptuelle, 6

Objet(s)
    agrégats, 41
    agrégés, 41
    base d', 14, 38
    catégories d', 47
    classe, 13, 42
    comportement d'un, 38
    élémentaires, 38
    état d'un, 38
    génériques, 89
    instance, 13, 42
    non-élémentaires, 38
    opérateur, 82
    représentatifs, 89
    structure d'un, 38
    subordonnées, 77
    superordonnés, 78
opérateur(s), 82
    polymorphes 53, 62

Propriétés
    clés, 27
    opératoires, 30, 38
    prédéfinies, 57
    principe d'héritage de, 44
    principe d'induction de, 43
    structurelles, 38

Raisonnement
    mécanismes de 11
    par abstraction 75
réaction, 31, 86
réseau sémantique, 9

Schéma, 9
    conceptuel, 8
    de représentation, 8, 9

surrogates, 40

Types abstraits, 12

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974

VU les rapports de présentation de Messieurs
. C. JULLIEN, Ingénieur
. J. MERMET, Directeur de recherche

Monsieur Ignacio de Jesus ANIA BRISENO

est autorisé à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR-INGENIEUR, spécialité "Informatique".

Fait à Grenoble, le 7 juin 1988

Georges LESPINARD
Président
de l'Institut National Polytechnique
de Grenoble