# Langages et modèles à objets

## État des recherches et perspectives

Jérôme Euzenat, Amedeo Napoli,
Roland Ducournau, Gérald Masini

# Table des matières

# Préface

La tendance des scientifiques à conférer à des mots simples du langage courant des acceptions nouvelles très spécialisées, recouvrant des concepts peu accessibles, est bien connue. Des catastrophes et du chaos aux groupes et aux anneaux, en passant par les cordes et les charmes, les exemples s'accumulent. En promouvant le terme « objet », les informaticiens ont probablement poussé cette démarche à son extrême. Existe-t-il en effet, aux exceptions près de « chose » ou « entité », des termes plus vagues et génériques ? Et pourtant, passée une courte période euphorique où toute démarche, tout langage, tout formalisme suscitait l'intérêt, voire l'enthousiasme, dès lors qu'il ou elle était « orienté(e) objets », force est de constater qu'il existe bien, autour de ce terme banal, un ensemble de concepts et de techniques informatiques qui possède une certaine identité.

C'est à l'explicitation de cette identité que cet ouvrage participe. Tâche difficile, car l'objet est maintenant présent dans plusieurs domaines de l'informatique — systèmes opératoires, langages de programmation, génie logiciel, bases de données, représentation des connaissances — et, au delà de quelques caractéristiques partagées, telle que la spécialisation de classes, il est difficile de ne pas être frappé par la diversité des formalismes, des langages et des mécanismes. Mais la lecture des différents chapitres conforte progressivement l'idée que l'apport des objets se situerait fondamentalement dans leurs capacités de modélisation, à savoir la mise en relation d'unités de description et d'entités d'un domaine d'intérêt. C'est vraisemblablement de cette adéquation à la modélisation du réel que découlent les propriétés tant appréciées des objets, y compris aux niveaux les plus bas des systèmes informatiques.

Bien entendu, les capacités de modélisation attendues diffèrent grandement, entre les niveaux où un objet « réel », tel que la pile, archétype de l'objet informatique, se confond de fait avec son modèle, et les niveaux où les objets modèles résultent d'un processus délicat d'abstraction et de simplification dirigé par des objectifs d'analyse et de compréhension. Dès lors, la question est bien de savoir s'il convient ou non de distinguer effectivement plusieurs niveaux de modélisation et d'y associer des formalismes et des langages différents. À ce titre, il peut être judicieux de s'interroger sur le décalage croissant entre les notations proposées par les diverses méthodes d'analyse et de conception par objets, qui mettent en avant la notion très expressive de relation, et les langages de programmation à objets, y compris les plus récents, qui l'ignorent et perpétuent la tradition du « tout objet ».

Seul un ouvrage collectif est à même de montrer comment la diversité des approches traduit les différences de motivation. Encore faut-il que les auteurs aient eu de fréquentes occasions de confronter leurs idées et leurs travaux, afin qu'apparaisse simultanément ce qui confère au domaine, et donc à l'ouvrage qui en traite, son identité. C'est bien le premier mérite de ce livre que de satisfaire pleinement toutes ces exigences. Loin de l'exposé des variantes syntaxiques de tel ou tel dialecte, ou graphiques de telle ou telle méthode, cet ouvrage propose une vision articulée de l'approche par les objets, contribuant à faire reconnaître la pertinence de l'informatique pour la modélisation, et donc la compréhension, du réel.

François Rechenmann
Directeur de recherche
INRIA Rhône-Alpes

# Avant-propos

L'ouvrage sur les langages à objets paru en 1989 [Masini *et al.*, 1989] s'ouvrait sur la constatation que l'intérêt pour les langages à objets dépassait le cadre restreint des spécialistes et se terminait sur la certitude que les concepts d'objets allaient prendre une place de premier plan dans tous les domaines de l'informatique. Nous y sommes certainement. Gageons que l'identification quasi immédiate des objets informatiques avec leur contrepartie physique n'y est pas pour rien.

Le présent ouvrage fait le point sur la diffusion du « modèle objet » dans de nombreuses disciplines de l'informatique à un moment où les travaux de recherche se poursuivent intensément et où les résultats en termes de systèmes d'exploitation, de bases de données, de méthodologies et de représentations des connaissances par objets sont disponibles auprès des utilisateurs.

Ce livre est issu de l'école d'été « Langages et modèles à objets » co-organisée par le Centre International de Mathématiques Pures et Appliquées (CIMPA) et l'Institut National de Recherche en Informatique et en Automatique (INRIA) à Nice, au mois de juillet 1996. L'école a été l'occasion pour les auteurs de confronter leurs points de vue à ceux d'étudiants venus du monde entier. Ce livre est aussi le résultat des nombreuses discussions menées au sein des groupes « Classification et objets » du PRC-GDR « intelligence artificielle » et « Évolution des langages à objets » du GDR « programmation » ainsi qu'aux quatre conférences « Langages et modèles à objets » qui se sont tenues entre 1994 et 1997. Le but de l'école et du livre est de donner un aperçu de la diversité des travaux développés à l'heure actuelle autour de la notion d'objet. L'ouvrage n'est pas centré sur quelques langages mais présente plutôt des grands thèmes choisis de ces travaux. Ainsi, chaque chapitre a été rédigé par un spécialiste du sujet et tous sauf un ont été conçus spécifiquement pour cet ouvrage. Celui-ci peut donc être lu sans connaissance préalable de langages particuliers mais ne donnera pas de connaissance approfondie d'un langage ou de son histoire. On insiste en échange sur les grands principes de chacun des thèmes en les caractérisant les uns pas rapport aux autres.

L'ouvrage commence par un chapitre prenant à revers les analyses globales actuelles sur les objets afin de cerner, en partant de l'implémentation, la nature de ces objets informatiques. En revenant à la source de l'objet informatique, il permet d'ancrer la suite de l'ouvrage dans le temps et dans la mémoire de l'ordinateur. L'ouvrage est ensuite découpé en quatre parties : génie logiciel et objets, développements avancés de la notion d'objet, objets pour la représentation des connaissances et application des objets.

***Génie logiciel et objets.*** Le succès actuel des objets vient d'abord de leur utilisation dans la pratique de l'industrie du logiciel autant que dans les théories et les méthodes du génie logiciel. La première partie est donc consacrée aux aspects logiciels. Le chapitre 2 détaille ce qu'un langage et un environnement de programmation par objets (en l'occurrence Eiffel) peuvent apporter au développement de logiciels importants. Cette première contribution est mise en perspective dans le chapitre 3 où sont comparés les langages EIFFEL et C++. Mais les objets ont fini par prendre, ces dernières années, leur place attendue dans deux domaines : il s'agit des méthodologies de conception de programmes et des bases des données. C'est à un état de l'art et un panorama des recherches en cours dans ces deux domaines que sont consacrés les chapitres 4 et 5.

***Développements avancés de la notion d'objet.*** S'il est des domaines dans lesquels les objets occupent maintenant une place prépondérante, il en est d'autres pour lesquels on peut conjecturer que ce sera le cas à moyen terme. La seconde partie de ce livre leur est consacré. Le chapitre 6 propose une classification des approches objets de la programmation parallèle et répartie. Il semble clair que le développement des réseaux informatiques et des ordinateurs parallèles renforcera le développement de ce type d'approche. Le chapitre 7 est consacré à l'une d'entre elles. Dans la présentation d'EIFFEL//, c'est l'intégration des principes de programmation par objets (réutilisabilité, abstraction) à la programmation concurrente qui est présentée. Les prototypes (des objets sans classes) permettent plus de souplesse dans le développement incrémental des applications. Les principes des langages à prototypes sont présentés dans le chapitre 8. Bien que contraire à ceux du genie logiciel, ils permettent d'analyser plus finement la véritable nature des objets. Enfin, la programmation par contraintes et la programmation par objets sont déjà le sujet de nombreux travaux. Leur union prometteuse, présentée au chapitre 9, sera féconde lorsque certains problèmes liés à l'interaction des deux approches auront trouvé des solutions harmonieuses.

***Objets pour la représentation des connaissances.*** Depuis le début des objets, les travaux sur la représentation des connaissances ont été le terrain de nombreux développements et expérimentations. Comme en programmation, la capacité des objets informatiques à représenter les objets modélisés est certainement l'un des aspects les plus déterminants dans cette adoption. Mais, à l'instar des langages de programmation, un consensus a du mal à voir le jour. Le chapitre 10 expose les traits caractéristiques qui gouvernent la représentation des connaissances par objets et introduit certains développements actuels en s'appuyant sur le système TROEPS. L'un des développements les plus aboutis de la représentation des connaissances est une famille de logiques, les logiques de descriptions, détaillée dans le chapitre 11. Les logiques de descriptions sont le résultat d'un développement théorique donnant lieu à leur classification sous le jour de l'expressivité et de la complexité de la déduction. Le même type de démarche est présenté dans le chapitre 12 qui concerne le noyau commun aux représentations des connaissances et aux langages de programmation par objets : les systèmes classificatoires. Ce travail aboutit à une analyse rigoureuse des mécanismes de classification tels qu'on les trouve dans certains systèmes de représentation des connaissances par objets.

***Exigences des applications.*** Enfin, la dernière partie est destinée à motiver les travaux sur les objets par l'entremise de domaines d'application exigeants. Si les objets semblent résoudre de nombreux problèmes, ils en posent aussi certains dont la résolution est cruciale. Le chapitre 13 est consacré à RESYN, une application des représentations des connaissances par objets à la synthèse de molécules en chimie organique. Ce type d'application pose le problème de la représentation de graphes (que l'on retrouve dans les applications à la gestion de réseau, par exemple). Il montre aussi qu'une telle application fait appel à de nombreuses techniques (dont la classification, objet des chapitres 11 et 12). Les difficultés de traduire, en terme d'objets, les notions d'un domaine ayant ses formalisations propres, tel que la musique, est ensuite présenté au chapitre 14. Une telle traduction est contrainte par la pauvreté relative du modèle objet. Mais lorsqu'elle est réussie, elle est source d'une meilleure appréhension des notions modélisées. Le chapitre 15 montre enfin que les classifications des objets ne correspondent pas, dans leurs fondements, aux classifications usuelles en sciences naturelles, qu'il s'agisse de classification phylogénétiques – puisqu'elles n'offrent aucune perspective historique – ou de clés d'identification – parce que le langage n'est pas adéquat. Ce dernier chapitre apporte, s'il en était besoin, du grain à moudre aux travaux futurs dans le domaine de la classification.

Cette introduction ne serait pas complète sans remercier ceux à qui ce livre doit son existence : tout d'abord les auteurs dont on trouvera les coordonnées page xii. Les organisateurs de l'école d'été du CIMPA (Agnès Gomez, Jean-François Prothé et le directeur du CIMPA, Claude Lobry) nous ont permis de confronter nos points de vue et de lancer l'idée de ce livre. En plus des auteurs dont les relectures croisées ont permis l'unité de l'ouvrage, Bernard Carré (LIFL, Lille), Isabelle Crampé (Inria Rhône-Alpes, Grenoble), Michel Page (Université Pierre Mendès-France, Grenoble), Marie-Christine Rousset (LRI, Orsay) et Stefano Spaccapietra (EPFL, Lausanne) doivent être remerciés pour leur relecture de certains chapitres. L'aimable autorisation des éditions Hermès a permis la publication du chapitre 6 (une version antérieure était parue dans la revue « Technique et science informatiques »). Enfin, le service d'imprimerie de l'INRIA (ou UCIS) a pris en charge la couverture et la réalisation matérielle de ce livre.

Roland Ducournau
Jérôme Euzenat
Gérald Masini
Amedeo Napoli

# Liste des auteurs

**1 Objets, classes, héritage : définitions**

Jean-François Perrot
Laboratoire d'Informatique de Paris VI (LIP6),
Université Pierre et Marie Curie (Paris VI),
Case 169, 4 place Jussieu, 75252 Paris Cedex 05
*Mél : Jean-Francois.Perrot@lip6.fr*

**2 Eiffel : une approche globale pour programmer et réutiliser des composants logiciels**

Roger Rousseau
Laboratoire I3S, Université de Nice – Sophia Antipolis (UNSA)
Les Algorithmes, Bâtiment Euclide
2000, Route des Lucioles, 06410 Biot
*Mél : Roger.Rousseau@unice.fr*

**3 Introduction à C++ et comparaison avec Eiffel**

Gérald Masini
LORIA — UMR 7503, Boîte Postale 239, 54506 Vandœuvre-lès-Nancy Cedex
*Mél : Gerald.Masini@loria.fr*

Karol Proch
Université Henri Poincaré Nancy 1, Département Informatique,
Boîte Postale 239, 54506 Vandœuvre-lès-Nancy Cedex
*Mél : Karol.Proch@loria.fr*

**4 Méthodes et d'analyse et de conception par objets**

Michel Dao
France Télécom CNET/DAC/GTR
38–40 rue du général Leclerc, 92794 Issy-les-Moulineaux Cedex 9
*Mél : michel.dao@cnet.francetelecom.fr*

**5 Les systèmes de gestion de bases de données objets**

Thérèse Libourel
LIRMM, Université de Montpellier 2, 161 rue Ada, 34392 Montpellier Cedex 5
*Mél : libourel@lirmm.fr*

**6 Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances**

Jean-Pierre Briot
Laboratoire d'Informatique de Paris VI (LIP6)
Université Pierre et Marie Curie (Paris VI) – CNRS
Case 169, 4 place Jussieu, 75252 Paris Cedex 05
*Mél : Jean-Pierre.Briot@lip6.fr*

Rachid Guerraoui
Laboratoire des Systèmes d'Exploitation (LSE), Département d'Informatique,
École Polytechnique Fédérale de Lausanne, CH–1015 Lausanne, Suisse
*Mél : Rachid.Guerraoui@epfl.ch*

**7 Programmation parallèle et réactive à objets**

Denis Caromel
Université de Nice – Sophia Antipolis / INRIA Sophia Antipolis
2004 Route des Lucioles, B.P. 93, 06902 Sophia Antipolis Cedex
*Mél : caromel@unice.fr*

Yves Roudier,
Electrotechnical Laboratory (ETL), Computer Science Division
1–1–4 Umezono, Tsukuba, Ibaraki 305, Japon
*Mél : roudier@etl.go.jp*

**8 Les langages à prototypes**

Christophe Dony
LIRMM, Université Montpellier 2, 161 rue Ada, 34392 Montpellier Cedex 5
*Mél : dony@lirmm.fr*

Jacques Malenfant
Université de Bretagne Sud, CGU de Vannes, 1 rue de la Loi, 56000 Vannes
*Mél : Jacques.Malenfant@univ-ubs.fr*

Daniel Bardou
LIRMM, Université Montpellier 2, 161 rue Ada, 34392 Montpellier Cedex 5
*Mél : bardou@lirmm.fr*

**9 Objets et contraintes**

Jérôme Gensel
INRIA Rhône-Alpes, ZIRST, 655 avenue de l'Europe, 38330 Montbonnot St Martin
*Mél : Jerome.Gensel@inrialpes.fr*

**10 Représentation de connaissance par objets**

Jérôme Euzenat
INRIA Rhône-Alpes, ZIRST, 655 avenue de l'Europe, 38330 Montbonnot St Martin
*Mél : Jerome.Euzenat@inrialpes.fr*

**11 Les logiques de descriptions**

Amedeo Napoli
LORIA – UMR 7503, Boîte Postale 239, 54506 Vandœuvre-lès-Nancy Cedex
*Mél : napoli@loria.fr*

**12 La logique des objets : application à la classification incertaine**

Roland Ducournau
LIRMM, Université Montpellier 2, 161 rue Ada, 34392 Montpellier Cedex 5
*Mél : ducournau@lirmm.fr*

**13 Objets et musique**

François Pachet
Sony Csl, 6 rue Amyot, 75005 Paris
*Mél : pachet@csl.sony.fr*

François Pachet est actuellement en disponibilité du LIP6, Université Pierre et Marie Curie (Paris VI).

**14 RESYN : objets, classification et raisonnement distribué en chimie organique**

Philippe Vismara
ENSA-M, Bâtiment de Biométrie, 2 place Pierre Viala, 34060 Montpellier Cedex 1
(également membre du LIRMM)
*Mél : vismara@ensam.inra.fr*

Pierre Jambaud
LIRMM, Université Montpellier 2, 161 rue Ada, 34392 Montpellier Cedex 5
*Mél : jambaud@lirmm.fr*

Claude Laurenço
CCIPE, 141 rue de la Cardonille, 34094 Montpellier Cedex 5
(également membre du LIRMM)
*Mél : cl@lirmm.fr*

Joël Quinqueton
LIRMM – INRIA, Université Montpellier 2,
161 rue Ada, 34392 Montpellier Cedex 5
*Mél : jq@lirmm.fr*

**15 Représentations par objets et classifications biologiques**

Jacques Lebbe
Université Pierre et Marie Curie (Paris VI),
Boîte 31, 4 Place Jussieu, 75252 Paris Cedex 05
*Mél : lebbe@ccr.jussieu.fr*

# Introduction

# 1

# Objets, classes, héritage : définitions

Le but de ce premier chapitre est de proposer un système de définitions pour la majeure partie du vocabulaire spécifique utilisé dans ce volume. Non pour imposer une norme, mais pour fixer un système de référence par rapport auquel chacun des contributeurs pourra marquer sa différence : on espère ainsi réduire les malentendus qui proviennent de l'emploi des mêmes mots, dans le même contexte, en des sens différents.

La technique de base qu'est le modèle à classes et instances est remarquablement simple. Dès lors, vu la popularité persistante de l'approche objet, on ne peut esquiver l'interrogation : « pourquoi ça marche ? ». Le principe de la réponse se trouve en confrontant les besoins ressentis par les utilisateurs et les moyens techniques censés les satisfaire.

Ce chapitre procède donc en trois temps. Dans un premier temps, sont présentés les deux termes du débat, à savoir les besoins à satisfaire et les conditions techniques de la solution. Dans un deuxième temps, sont définis la solution en question et le vocabulaire assorti, avec les restrictions nécessaires pour simplifier et clarifier les choses. Enfin, dans un troisième temps, la solution avancée est discutée.

## 1.1 Ouverture

### 1.1.1 Intentions

**Idée générale**

Pour qui cherche à décrire l'univers de l'*orienté objet* (pour reprendre un barbarisme familier) le premier caractère observable est l'emploi d'un certain vocabulaire : les mots *classe*, *instance*, *héritage* occupent une position centrale, alors qu'ils sont plutôt rares dans le discours informatique ordinaire ; d'autres comme *envoi de message* et *méthode* sont utilisés en des sens très particuliers ; d'autres enfin comme *type* ou *attribut* ont des acceptions qui recoupent celles de l'usage informatique habituel, mais sont affectés de nuances subtiles ; le cas du mot *objet* lui-même est particulièrement délicat, car il est manifestement employé en un sens technique précis (mais lequel ?), qui reste compatible avec le sens d'*objet informatique* en usage courant. Malheureusement, si l'enquêteur cherche à en savoir davantage, un rapide coup d'œil à la littérature, depuis les origines jusqu'à aujourd'hui, le convaincra qu'en vingt ans les habitants de la planète des objets ne se sont toujours pas mis d'accord, ni sur le sens exact des termes qu'ils emploient, ni même sur les notions que ces termes sont censés désigner.

Le but de ce premier chapitre est de proposer un système de définitions pour la majeure partie du vocabulaire spécifique utilisé dans ce volume. Non pour imposer une norme, mais pour fixer un système de référence par rapport auquel chacun des contributeurs pourra marquer sa différence : on espère ainsi réduire les malentendus qui proviennent de l'emploi des mêmes mots, dans le même contexte, en des sens différents. Naturellement, pour expliquer la signification d'un terme, il est très utile d'indiquer aussi ce que cette signification ne recouvre pas. Nous proposerons donc une motivation et surtout une discussion critique de ce système de définitions. Ceci nous conduira à mettre l'approche objet en perspective, à discerner ses faiblesses et ses limitations. Au-delà des mots, l'on verra apparaître de nombreuses difficultés conceptuelles, certaines bien connues, d'autres soigneusement cachées. Ce texte prolonge et adapte un exposé antérieur plus détaillé sur certains points [Perrot, 1995], auquel on pourra se reporter le cas échéant.

Nous aurons plusieurs fois à invoquer Aristote et la tradition qui le suit. Pour cela nous utiliserons comme source le petit livre de la collection *Que sais-je ?* qui lui est consacré [Brun, 1988]. En effet, selon nous, l'approche objet reproduit fidèlement, dans le contexte informatique, une partie de la démarche logique et métaphysique du Stagirite, telle qu'elle nous a été transmise à travers les générations de penseurs qui ont forgé les outils intellectuels dont nous nous servons. Cette référence n'est pas innocente : elle manifeste que l'approche objet met en œuvre un système de pensée antérieur à la révolution opérée au XVII$^{e}$ siècle par Galilée et Descartes. Analysant le passage de la physique d'Aristote à la physique moderne (au sens de l'histoire des sciences), Alexandre Koyré note que *la physique d'Aristote [...] s'accorde — bien mieux que celle de Galilée — avec le sens commun et l'expérience quotidienne [...] Elle se refuse à substituer une abstraction géométrique aux faits qualitativement déterminés de l'expérience et du sens commun* (Galilée et la révolution scientifique du XVII$^{e}$ siècle, [Koyré, 1973, page 201]).

Mais l'approche objet ne vise pas la physique, direz-vous, et la critique de Koyré ne s'applique pas à elle. Ce point serait à examiner avec soin. Notons que dans le domaine de la classification des êtres vivants, la démarche d'Aristote révisée au siècle des Lumières a été bouleversée de nos jours par la cladistique, expression sans nuances du triomphe de l'évolutionnisme (voir la section 4 du *chapitre 15*). Il est clair que la démarche des naturalistes offre de nombreux points communs avec l'approche objet : les analystes et concepteurs par objets ont beaucoup à apprendre de l'expérience historique de leurs collègues biologistes systématiciens. La révolution cladistique devrait donc leur donner à réfléchir.

Le fond de la question, selon nous, est que l'approche objet permet de traduire en structures informatiques des opérations intellectuelles qui relèvent du *bon sens* (qui se dit en anglais, ne l'oublions pas, *common sense*). C'est là sa force. C'est aussi peut-être la source cachée de nombreuses difficultés qui sont aussi celles de l'aristotélisme si, pour citer de nouveau Koyré [Koyré, 1973] le sens commun est — et a toujours été — médiéval et aristotélicien.

**Précisions**

Notre analyse sera, sans aucun doute, partielle et partiale. Elle sera centrée sur la *programmation par objets* proprement dite et sur les *langages à objets*, qui sont historiquement à l'origine d'une partie des développements dont il est question dans ce volume. Elle prendra en compte, au moins en partie, la problématique de deux domaines voisins, celui des *bases de données à objets* (traitée dans le *chapitre 5*) et celui de la *représentation de connaissances*, issue de l'intelligence artificielle (traitée en particulier dans les *chapitres 10, 11 et 12*). Mais elle laissera de côté les préoccupations du génie logiciel qui ont donné naissance à l'*analyse* et à la *conception par objets* (traitées dans le *chapitre 4*), les questions de parallélisme et de distribution liées aux *objets concurrents* (traitées dans les *chapitres 6 et 7*), ainsi que les *systèmes d'exploitation à objets*.

Enfin, nous nous intéresserons aux langages à objets tels qu'ils sont, tels que l'industrie du logiciel les emploie : SMALLTALK, C++, EIFFEL, JAVA. Il nous semble en effet que ce sont eux, à travers les succès qu'ils ont remportés, qui sont la source de la popularité de l'ensemble de l'approche objet, et que leur architecture commune constitue la référence primaire en la matière. Au grand regret de l'auteur, nous ne dirons rien du *Common Lisp Object System* (CLOS) et surtout nous laisserons de côté OBJECTIVE-CAML, qui sort de notre cadre tant par ses bases implémentatoires que par son ampleur conceptuelle (double système de modules et de classes). Sur CLOS, on dispose d'un excellent texte introductif [Habert, 1995] ; sur OBJECTIVE-CAML, qui n'est pas encore complètement stabilisé, on se reportera à l'URL `http://pauillac.inria.fr/ocaml/`.

Nous prendrons comme point de départ la technique d'implémentation. C'est là procéder au rebours de tout un mouvement de pensée, assez répandu, qui cherche à dégager des concepts et des formalismes supposés indépendants de l'implémentation. Nous estimons que l'approche objet, telle qu'elle se pratique, ne relève pas de ce mouvement, pour des raisons qui apparaîtront plus loin, et qu'à vouloir l'examiner sans référence aux vicissitudes opérationnelles, on est conduit à prendre ses désirs pour des réalités. Cette tentation est particulièrement forte en raison d'une ambiguïté fondamentale — la classe comme extension ou comme intension — que nous analyserons longuement. En fin de compte, c'est sans doute cette ambiguïté qui est la source de la fécondité et de la technique, mais il faut en être bien conscient, pour ne pas se faire prendre aux pièges du vocabulaire et pour ne pas se laisser désorienter par les débats interminables sur le sens des mots, qui surgissent dès qu'on pose la question : « qu'est-ce qu'un objet ? ».

Comme on verra, la technique de base, le modèle à classes et instances, est remarquablement simple — on serait même tenté de la qualifier de « rustique ». Dès lors, vu la popularité persistante de l'approche objet, on ne peut esquiver l'interrogation : « pourquoi ça marche ? ». Cette question rejoint en fait le problème du sens précis donné aux termes du vocabulaire technique, car en fait on parle peu de ce qui ne marche pas. Le principe de la réponse, croyons-nous, se trouve en confrontant les besoins ressentis par les utilisateurs et les moyens techniques censés les satisfaire.

Nous procéderons en trois temps : les deux paragraphes suivants présenteront les deux termes du débat, à savoir les besoins à satisfaire et les conditions techniques de la solution. Dans un deuxième paragraphe, nous définirons d'un ton dogmatique la solution en question et le vocabulaire assorti, avec les restrictions nécessaires pour simplifier et clarifier (par exemple, supposer que tous nos objets résident en mémoire centrale). Dans la troisième, nous discuterons cette solution, comme annoncé plus haut.

### 1.1.2 Besoins ressentis par la communauté

L'émergence de l'approche objet se situe dans l'histoire de l'informatique à un moment où les progrès de la technique ont permis de réaliser des systèmes complexes, abordant des domaines d'application très variés, avec une exigence de sûreté renouvelée. Il est devenu indispensable de modéliser explicitement le problème traité, autrement que par un système d'équations, et de confier un rôle de plus en plus important au spécialiste du domaine. C'est dans cette perspective qu'il faut apprécier l'approche objet.

Elle répond clairement à trois préoccupations majeures que nous analysons ci-après, et qui rapprochent les structures informatiques des formes et des intuitions de la pensée ordinaire. Comme nous le verrons, une même technique permet d'apporter une solution approximative, mais efficace, aux trois problématiques à la fois.

**Unicité, unité, individualité**

Toute l'approche objet repose sur la conviction que les objets du monde réel offrent une base solide à notre intuition. Elle considère que les objets existent, qu'ils « sont bien là », n'en déplaise aux philosophes et à leur critique de la perception. Elle fait au besoin les restrictions nécessaires quant à son domaine de validité : s'il s'agit de manipuler un morceau de cire, on se gardera de l'approcher d'une flamme, pour échapper aux arguments de Descartes. En outre, elle ne les envisage pas en eux-mêmes, mais seulement dans une perspective d'utilisation, selon un certain projet : attitude pragmatique qui permet des simplifications salutaires. Moyennant ces limitations, il faut bien reconnaître que l'héritage culturel de l'*homo faber* nous donne une efficacité extrême dans la manipulation des objets matériels (le plus souvent vus comme des outils) : c'est cette efficacité qu'on souhaite transférer dans le domaine de la programmation.

Or, l'univers du programmeur est peuplé non pas d'objets réels, mais d'entités informatiques, qui ne sont que des amas d'octets en machine. Les objets du domaine d'application sont, comme on dit, représentés en machine. Dans les approches traditionnelles, dès qu'il s'agit d'objets complexes, cette représentation repose sur une analyse qui va faire éclater l'objet en de multiples facettes et finalement le dissoudre complètement pour se ramener à un système de variables à valeurs numériques : l'unité de l'objet, et sa signification, ne se réalisent que dans l'esprit du programmeur. Si l'on a affaire à une collection d'objets complexes, on aboutit avec les bases de données relationnelles à un jeu de tables contenant des valeurs élémentaires, d'où tout individu complexe est banni.

Il en résulte pour le développeur de logiciels une charge intellectuelle qui devient incompatible avec la conquête des nouveaux domaines d'applications rendus accessibles par les progrès de la technique, comme nous l'avons dit plus haut : la programmation devient trop difficile, les systèmes réalisés sont fragiles, etc. Au contraire, le principe même de l'approche objet est d'établir une correspondance directe, bijective, entre les objets du monde réel et leurs ersatz informatiques, permettant au programmeur de s'exprimer conformément à son intuition. C'est ce que résume la deuxième règle d'or du *Manifeste des Bases de données à objets* [Atkinson *et al.*, 1989] par l'injonction *Thou shalt support object identity*[1]. C'est en apparence un retour au simple bon sens... à condition de savoir le faire !

Ce désir raisonnable de retrouver une unité perdue rencontre un autre courant d'idées, qui est issu de l'intelligence artificielle. Dans ce domaine, l'utilisation de structures complexes réalisées par des jeux de pointeurs et accessibles via une adresse unique est monnaie courante dès l'origine (n'oublions pas que l'intelligence artificielle à ses débuts parlait LISP). Ces structures sont vues comme des réceptacles de connaissances, de compétences, propres à prendre à leur compte certains choix et certaines opérations, et à en décharger *ipso facto* le programmeur. Le problème est alors l'organisation de ces connaissances. Du point de vue qui nous intéresse, deux tendances se dégagent : l'une s'oriente vers la représentation des connaissances incomplètes, avec l'idée des *frames* de Marvin Minsky [Minsky, 1975], dont nous reparlerons ; l'autre rêve d'objets véritablement autonomes, auxquels on s'adresserait en leur envoyant des messages polis : les acteurs du langage PLASMA de Carl Hewitt [Hewitt, 1977], qui, sous l'impulsion de Patrick Greussay, fut soigneusement étudié à Paris et à Toulouse dans les années 80.

Naturellement, c'est la technique d'implémentation qui met une borne aux phantasmes des utilisateurs : dès qu'on quitte l'environnement LISP, les problèmes de réalisation deviennent aigus. Les objets ne sont pas des acteurs — mais la programmation par objets a conservé l'idée de l'envoi de message, et avec elle l'intuition que l'objet est un individu respectable et compétent, auquel on s'adresse dans sa langue.

### Abstraction : classes et types

L'attention portée à l'objet individuel « concret » donne une nouvelle coloration à la distinction fondamentale entre concept (abstrait) et instance (concrète) du concept — alias définition/exemple, espèce/individu, etc. Il s'agit d'un très ancien problème, celui du statut des concepts abstraits. Platon les appelait les idées, et il pensait qu'ils existaient réellement dans le monde comme entités séparées. Aristote, critiquant son maître, estime au contraire que les concepts généraux ne sont que des attributs, inséparables des individus auxquels ils s'appliquent [Brun, 1988, page 29]. Ce conflit se poursuit jusqu'à nos jours dans le débat entre nominalistes (qui suivent *grosso modo* Aristote) et réalistes (qui seraient du côté de Platon) avec, au Moyen-

1. Ton système doit gérer l'identité des objets.

Âge, la fameuse « Querelle des Universaux ». Comme nous le verrons, l'approche objet conduit à reposer la même question, dans un contexte renouvelé (voir § 1.3.3).

L'informatique suit une pente plutôt nominaliste. C'est la notion de type qui a occupé ce terrain, d'abord comme indications au compilateur sur l'organisation de la mémoire, ensuite comme annotations systématiques au texte du programme, donnant lieu au contrôle de types, enfin comme spécifications ou interfaces (types abstraits). Il faut souligner qu'un système de types est avant tout un langage qui doit permettre de décrire les entités manipulées par le programme d'une manière qui permette une vérification de cohérence *a priori*, sur le texte du programme (contrôle statique). Le malheur veut que l'exigence de contrôlabilité statique contraigne fortement ce langage, l'empêchant d'exprimer toutes les subtilités de la pensée du programmeur. Certains préfèrent donc ne pas s'encombrer de ces limitations, renoncer au contrôle statique et assumer dynamiquement leur liberté : la tradition SMALLTALK illustre brillamment les possibilités de l'option « non typée ». Toutefois, pour des raisons qu'il conviendrait d'analyser avec soin, la demande de sécurité paraît aujourd'hui majoritaire et, avec elle, la demande de langages fortement typés : EIFFEL est le représentant par excellence de cette orientation. La question des systèmes de types — au sens fort, pour contrôle statique — se pose donc pour les langages à objets. Or, elle se pose dans un contexte nouveau.

La promotion de la notion d'objet va en effet donner le branle à un double mouvement : d'une part, l'accent mis sur la présence d'individus en machine rend désirable leur catégorisation, selon un besoin irrépressible de l'esprit humain ; d'autre part, la technique d'implémentation va engendrer des catégories naturelles : les classes. Du coup, la pratique va s'engouffrer dans la programmation par classes et instances, avant de s'interroger sur le rapport entre les classes et les types.

En d'autres termes, il y a une pression exercée par les objets, ces « nouveaux individus », qui instaurent une nouvelle perception de la concrétude en machine et sont donc « demandeurs d'abstraction ». D'autre part, la « traction » engendrée par la technique implémentatoire des classes vient satisfaire ce besoin d'une manière qui diverge subtilement de la notion traditionnelle de type.

Et le problème des types ainsi renouvelé va se trouver enrichi par le besoin de classification hiérarchique qui représente, lui, une innovation en informatique. En effet, les classes entrent très naturellement dans une structure hiérarchique (via la relation d'héritage, voir le paragraphe suivant), ce que les types traditionnels ont bien du mal à faire. D'où une problématique nouvelle et féconde : d'une part, il faut savoir dans quelle mesure les classes peuvent entrer dans le système des types ; d'autre part, il faut absolument que le système des types accommode d'une manière ou d'une autre la relation d'héritage.

**Hiérarchisation : héritage, sous-typage**

Le besoin de classer se rencontre dans toutes les sociétés humaines : voir par exemple [Vogel, 1988, page 97] (au début du chapitre sur les taxinomies) Les concepts étant naturellement dotés d'une relation d'ordre, celle de leur plus ou moins grande généralité, nous ne pouvons pas nous empêcher de les comparer sous ce rapport. Un concept `B` est plus général qu'un concept `A` si toute entité qui, comme on dit, « tombe sous le concept `A` » tombe aussi sous `B` : ce qui se réalise, par exemple, si nous prenons pour `B` le concept de mammifère et pour `A` celui de félin. Ce qu'on exprime en disant *tout* `A` *est un* `B`, forme brève de *toute instance de* `A` *est instance de* `B` : ainsi, *tout félin est un mammifère*. Et en empruntant le langage ensembliste : `A` *est inclus dans* `B`.

En contre-coup du « réveil » du problème de l'abstraction évoqué ci-dessus, nous voulons faire de même avec les images des concepts en machine, d'abord les classes, ensuite les types.

En ce qui concerne les classes, le mécanisme d'héritage vient opportunément satisfaire ce besoin. Il a remporté suffisamment de succès pour être retenu par de bons auteurs comme la caractéristique majeure de la programmation par objets. Dans un grand projet logiciel, l'organisation hiérarchisée du système des classes apparaît vite comme une structure indispensable. Mais, en vérité, le mécanisme d'héritage s'éloigne fort de l'intuition commune — la spécialisation comme inclusion des concepts — qui lui donne sa force attractive, ce qui cause de multiples difficultés.

En ce qui concerne les types, cette idée de hiérarchie n'était pas apparue auparavant en informatique — en tous cas pas au premier plan : la notion de sous-type est absente d'un langage aussi perfectionné que ML. Elle ne s'introduit pas sans mal, sous la pression de l'héritage des classes dont il faut absolument aujourd'hui rendre compte en termes de types. En fait, comme on le verra, la constatation majeure est que l'héritage ne peut pas se traduire simplement par le sous-typage. L'exigence de typage statique va donc entraîner une complication du modèle, avec deux structures pour l'abstraction, d'un côté les classes et leur héritage, de l'autre les types et leur sous-typage.

#### Du général au particulier

Nous venons de distinguer très nettement deux mécanismes, celui d'abstraction qui oppose un objet (concret) à son concept (abstrait), et celui de hiérarchisation des concepts suivant leur degré de généralité. Dans nos langues « naturelles », ces deux mécanismes ne se distinguent pas : nous affirmons qu'*un chien est un mammifère* de la même manière que nous disons que *Médor est un chien*. La distinction stricte entre classes et instances nous impose de traduire ces deux énoncés par deux mécanismes différents, le premier par un lien de spécialisation entre les classes *chien* et *mammifère*, le second par un lien d'instanciation : l'individu *Médor* est instance de la classe *chien*. Devoir séparer dans la programmation des énoncés que la langue naturelle confond est une contrainte qui peut être jugée salutaire ou insupportable suivant les buts poursuivis ; l'approche objet l'accepte. On sait que d'autres démarches ont cherché à conserver l'ambiguïté de l'expression « X est un Y » : notamment celle des *frames*, avec le lien *is-a* (voir [Masini *et al.*, 1989, section 8.3.1], mais aussi le *chapitre 8*).

En fait, ces deux mécanismes peuvent être vus comme la manière dont l'approche objet aborde un des problèmes de fond des langages de programmation : le passage du général au particulier. Plus précisément, il s'agit de trouver le moyen d'exprimer des informations à caractère général et de les mettre en œuvre dans un cas particulier sans avoir à les reformuler. De ce point de vue, une classe est un énoncé général, dont les instances sont autant de particularisations ; de même, le lien d'héritage permet de passer d'une classe plus générale à une autre plus particulière.

On connaît un autre mécanisme qui répond au même besoin de particulariser des énoncés généraux, la *généricité paramétrique*, connue également sous le nom de *polymorphisme* en théorie des types : si je sais faire la théorie générale des piles dont les éléments sont de type `X`, je peux m'en servir automatiquement pour faire fonctionner une pile d'entiers. Ce mécanisme est en fait la solution au problème de la particularisation que propose l'approche par types abstraits. À son égard, l'approche objet n'a rien de spécifique à dire : l'idée de généricité s'applique banalement à la notion de classe, donnant naissance aux *classes génériques* d'EIFFEL (voir *chapitre 2*) et aux *patrons* (alias *templates*) de C++ (voir la section 6 de *chapitre 3*). D'autres langages préfèrent y renoncer (SMALLTALK, JAVA). Avec un peu d'habileté, on peut proclamer que le mécanisme d'héritage permet de se passer de généricité (voir [Meyer, 1986]). Mais il est certain que pour les langages qui s'offrent le luxe d'incorporer la généricité paramétrique, la présence de l'héritage et l'exigence de sous-typage viennent sérieusement compliquer l'implémentation. On verra plus loin un échantillon des difficultés rencontrées (§ 1.3.2).

### 1.1.3 Présupposés sur la technique d'implémentation

Les choix d'implémentation que nous allons expliciter, et qui sous-tendent la quasi-totalité des réalisations, sont fort difficiles à justifier si l'on ne tient pas compte des moyens informatiques supposés disponibles. Pour décrire d'une manière adéquate les mécanismes de base des langages à objets, nous nous situons dans un univers très primitif, *grosso modo* celui de la programmation en assembleur, ou celui du *bytecode* d'une machine abstraite — pas l'univers LISP, encore moins celui de ML. Nous souhaitons ainsi exposer les choses comme elles sont, non comme elles devraient être.

#### L'outillage de base

La première hypothèse est que les seules entités à notre disposition sont de deux sortes, les données et les procédures. Plus précisément, nous nous plaçons dans une architecture mono-processeur, avec une seule pile, et nous ne comptons pas sur le pseudo-parallélisme des processus, ni même des coroutines. Cela exclut notamment des êtres qui seraient extrêmement utiles dans de très nombreuses situations comme les contraintes, mais dont la réalisation informatique offre des difficultés supérieures d'au moins un ordre de grandeur à celle des entités de base que sont les données et les procédures.

#### Le tas

En revanche, nous supposons acquis le partage de la mémoire de travail en trois zones, la zone code, la pile et le tas. Le tas est administré par un gérant capable d'allouer des blocs de taille variable et, le cas échéant, de les récupérer grâce à un mécanisme de *ramasse-miettes* (*garbage-collecting*). Nous considérons exclusivement des données structurées hétérogènes, ou enregistrements (*records*), repérées par leur adresse dans le tas (alias pointeur), chaque champ composant étant connu par un déplacement (*offset*) à partir de cette adresse. Le rôle de la pile se limite au traitement de la récursion.

**Les procédures ne sont pas des données**

Enfin, point essentiel, nous admettons que données et procédures sont des êtres de natures différentes, qui ne sont pas représentés de la même manière et qui ne suivent pas les mêmes lois. C'est la situation commune, à laquelle n'échappent que les langages à fermetures (COMMON LISP, SCHEME, ML), qui sont bien au-dessus de notre modeste atelier.

Nous voyons les procédures comme des textes compilés sous forme exécutable, où tous les noms sont résolus en adresses et en déplacements. Ces textes sont repérés par leur adresse dans la zone code. Ceci signifie que nous renonçons à effectuer des calculs sur les procédures (ce qui demanderait de les considérer comme des données ordinaires), en particulier que nous n'envisageons pas des procédures qui seraient le résultat d'un calcul de notre programme. Nous ne nous autorisons donc à leur égard que trois opérations : la rédaction (écriture dans un fichier), la compilation (ou mise sous forme interne) et l'exécution.

Comme nous allons le voir, on peut proposer, dans ce cadre, une solution simple et efficace aux besoins évoqués au paragraphe 1.1.2 ci-dessus. En revanche, il sera fort malcommode de faire entrer la solution en question (et la pratique qui la met en œuvre) dans une spécification *a priori*.

## 1.2 Définitions

Comme d'ordinaire en informatique, les objets n'ont de sens qu'en raison de leur fonctionnement. Il nous faut donc introduire d'un même coup plusieurs termes qui se définissent l'un par rapport à l'autre.

### 1.2.1 Objets

Il y a trois aspects dans la définition d'un objet, qui concourent tous trois à forger son individualité.

**Adresse**

Un objet est une entité individuelle, repérée par une adresse unique. Pour simplifier, nous ferons l'hypothèse que cette adresse repère une zone mémoire située dans le tas, qui sera soumise au ramasse-miettes. Clairement, cette restriction est insupportable du point de vue des bases de données ! Nous estimons cependant qu'elle a l'avantage de livrer un principe d'implémentation qui peut se décliner en de nombreuses variantes pour raisons d'efficacité, alors que le problème des objets permanents n'admet pas de solution simple. Nous la proposons donc comme référence.

Le besoin d'unité/unicité étant ainsi satisfait d'entrée de jeu, la question est désormais : que trouve-t-on au juste à cette adresse et comment s'en sert-on ?

### État

Comme un système physique, un objet possède un état (interne) qui peut varier au cours du temps. C'est donc une entité *mutable* au sens de ML. Nous discuterons ce point plus loin (voir § 1.3.1).

Cet état se réalise de la manière la plus simple, sous la forme d'un enregistrement de PASCAL (`record`) ou d'une structure de C (`struct`), formé de plusieurs *champs* (en nombre fixé), connus par leurs noms. Matériellement, les champs sont repérés par leur déplacement à partir de l'adresse de l'objet prise comme adresse de base.

Chaque champ contient une valeur (en général l'adresse d'un autre objet), qui peut varier au cours du temps. L'ensemble des valeurs des champs constitue à chaque instant l'*état courant* de l'objet. Dans l'analogie avec un système physique, les noms des champs s'identifient aux variables d'état du système : on voit (sans surprise) que nous nous limitons à des systèmes à un nombre fini de degrés de liberté.

Nos champs correspondent aux *slots* de la tradition des *frames* (voir *chapitre 10*). Mais nous ne les analysons pas en *facettes* : nous considérons « bêtement » qu'ils contiennent des valeurs, sans nous demander si ces valeurs sont effectivement valides, etc. Tout au plus attacherons-nous à chaque champ l'indication du type de sa valeur.

Dès qu'on s'éloigne de la matérialité de la réalisation en machine, on préfère parler des *attributs* d'un objet plutôt que des champs d'un enregistrement. Dans notre usage, les deux termes sont synonymes. Les différents langages classiques emploient pour dire la même chose une terminologie riche et diversifiée : en SMALLTALK, on dit *variables d'instance* (et le cas échéant on précise qu'on parle de leur nom) ; en C++, ce sont les *données membres* (*data members*, voir *chapitre 3*) ; en EIFFEL, des *primitives* (*features*, voir *chapitre 2*).

Rappelons qu'il faut distinguer soigneusement le nom de l'attribut (que le compilateur convertit en un déplacement) de la valeur qu'il contient. À un instant donné, donc, l'ensemble (ou plutôt le vecteur) des valeurs des attributs constitue l'état de l'objet.

Mais comment se manifeste cet état ? Et comment s'en sert-on ?

### Comportement

L'objet réagit à des sollicitations extérieures. Il est donc doué d'un comportement. Nous reviendrons plus loin sur ce point essentiel. C'est en fait cette capacité réactive qui va donner à ses utilisateurs l'illusion d'une existence concrète ; elle distingue les objets des « structures de données » traditionnelles (voir discussion ci-après, § 1.3.1).

La modélisation du comportement est le point délicat de notre système de définitions. C'est que le comportement est dynamique, et il ne pourra être obtenu, en dernière analyse, qu'en exécutant du code. Quel code, et comment l'activer ? *Hic jacet lepus*[2] !

À titre de comparaison, rappelons que dans les années 1970, les acteurs de Carl Hewitt avaient leur comportement modélisé comme un script unique activé par filtrage de messages [Hewitt, 1977]. L'approche objet revient à une réalisation plus grossière, mais en pratique fort efficace.

Chaque objet n'a qu'un nombre fini de procédures à sa disposition, chacune portant un nom. Le comportement de l'objet consiste à exécuter l'une de ces procédures, en réponse à un « message » très simplifié, qui ne contient que le nom de la procédure et les arguments de l'appel. Bien entendu, l'effet de cette exécution dépend de l'état courant de l'objet et il peut modifier cet état.

Par rapport à la programmation procédurale classique, le changement vient de ce que le message mentionne le nom de la procédure à exécuter, mais que le choix de la procédure elle-même (du corps de procédure) est effectué par l'objet destinataire du message : deux objets différents peuvent associer à un même nom des procédures différentes (on parle à ce sujet de surcharge ou de polymorphisme). C'est pourquoi nous continuons à utiliser le mot « message », plutôt qu'appel indirect de procédure : on met ainsi en valeur l'autonomie de l'objet, pivot essentiel de toute notre approche.

Soulignons que dans cette définition les procédures activées par les messages, qui définissent le comportement de l'objet, ne sont pas traitées comme valeurs de champs de l'enregistrement, puisque, comme nous l'avons rappelé, nous ne considérons pas que des procédures puissent être traitées comme des valeurs. Un objet apparaît donc comme constitué de deux parties, d'une part l'enregistrement qui définit son état, d'autre part la table de correspondance noms – corps de procédures qui définit son comportement.

Du point de vue de la représentation des connaissances, l'inconvénient de cette solution est qu'il est difficile (pour une machine) de raisonner sur des procédures : par suite, il est difficile de raisonner sur le comportement d'un objet ; et comme, en outre, la structure interne de l'objet (le système des champs) va être traitée comme confidentielle (privée) la plupart du temps, il faut reconnaître que nos objets se prêtent mal au raisonnement informatisé. C'est pourquoi la représentation des connaissances préfère utiliser des *frames*, où le comportement des objets est obtenu par des « attachements procéduraux » : il s'agit bien, en dernière analyse, d'exécuter des procédures, mais dans ce cadre, elles sont appelées à travers des mécanismes de *réflexes* ou *démons*, qui sont censés déclarer une partie de leur sémantique et ainsi donner prise au raisonnement (voir *chapitre 10* et [Masini *et al.*, 1989, section 8.3.5]).

Nous avons insisté sur la nature procédurale du comportement de nos objets, mais dans le langage courant on préfère l'oublier. On parle alors de *méthodes*, dans le vocabulaire de SMALLTALK, le nom de la méthode étant appelé *sélecteur* (à ce sujet, on notera que les conventions très particulières de la syntaxe de SMALLTALK font que la forme du sélecteur dénote l'arité de la méthode, à l'opposé des possibilités de surcharge qu'offrent C++ et JAVA). EIFFEL pour sa part préfère parler de

2. Ou : *c'est là que les Athéniens s'atteignirent ...!*

*routines* (chapitre 2), C++ de *fonctions membres* (*member functions*, voir chapitre 3).

### 1.2.2 Classes

Il reste à expliquer comment les objets que nous avons définis ci-dessus peuvent être créés en machine. Nous y arriverons en introduisant une nouvelle entité, la classe, qui va aussitôt se trouver chargée de représenter les concepts abstraits, bien qu'elle n'ait pas été définie dans ce but.

Le problème central est celui de la genèse du comportement des objets. C'est ici que les hypothèses que nous avons faites plus haut sur la nature des procédures pèsent de tout leur poids. Puisque dans notre système une procédure ne peut pas être le résultat d'un calcul, elle doit provenir directement de la compilation de son texte. Ceci vaut en particulier pour les procédures qui définissent le comportement d'un objet. Cette circonstance donne la primauté à la représentation textuelle et conduit tout droit à la notion de classe. Si le comportement pouvait être obtenu par un véritable calcul, la classe perdrait peut-être de son importance. Mais il faut se rappeler que, même en programmation fonctionnelle, le calcul des procédures se réduit à fixer les valeurs des variables libres qui apparaissent dans leur texte. Les classes, même si l'on cherche à les cacher comme dans les langages à prototypes (voir chapitre 8), ne sont donc jamais bien loin.

#### Texte

Une *classe* est un texte structuré qui rassemble, d'une part, une liste de noms de champs, et, d'autre part, un *dictionnaire* ou *catalogue de procédures*, appelé encore *catalogue de méthodes*, (noms et textes). Les noms de champs apparaissent comme variables libres dans les textes des procédures. Ce texte apparaît sous deux formes, soit comme texte source dans un fichier, soit comme texte compilé en mémoire.

Une *instance* d'une classe est un objet constitué des champs dont la liste est donnée par la classe et dont le comportement est défini par l'interprétation des messages, comme expliqué ci-dessous.

Cet objet possède un champ supplémentaire qui contient l'adresse du code compilé de sa classe.

#### Interprétation des messages

Son principe est le suivant :

1. le message fournit directement le nom de la procédure à exécuter et le corps de procédure correspondante est recherché dans le catalogue de procédures de la classe ;
2. ce corps est ensuite exécuté avec la convention que chaque nom de champ qui y figure sera interprété comme désignant le champ correspondant de l'instance qui traite le message (par exemple, en interprétant les noms de champs comme des déplacements à partir de l'adresse de l'objet prise comme adresse de base).

On peut résumer le mode d'exécution du corps de procédure en disant que l'exécution a lieu « dans le contexte local de l'objet ». En effet, l'état courant de l'objet, à savoir le système des couples « (nom de champ,valeur du champ) », peut être vu comme un contexte d'exécution fixant les valeurs des variables libres du corps de procédure. Deux objets différents, instances de la même classe, définissent des contextes isomorphes mais différents, dans lesquels l'exécution d'un même corps de procédure peut donner des résultats différents : c'est ainsi que l'exécution du corps de procédure dépend de l'état de l'objet destinataire du message ; c'est ainsi également que cette exécution peut modifier l'état en question.

Si nous revenons aux considérations du paragraphe 1.2.1 sur l'autonomie de l'objet dans le choix de la procédure à exécuter en réponse à un message, nous voyons que deux objets qui sont instances de la même classe feront le même choix : pour que le même message adressé à deux objets différents provoque un comportement différent, il faudra que les catalogues des méthodes des deux classes associent au même nom (sélecteur) des procédures différentes.

**Instanciation**

La création d'une instance à partir de sa classe se décompose en deux phases :

1. l'allocation d'une zone-mémoire de taille $k + 1$, où $k$ est le nombre des champs, le premier mot recevant l'adresse de la classe, suivie de
2. l'affectation à chaque champ d'une valeur initiale.

La première phase est confiée au gestionnaire de mémoire ; elle échappe au contrôle du programmeur. La seconde, en revanche, fait l'objet de procédures d'initialisation qui doivent être explicitement rédigées (à moins qu'on ne dispose d'initialisations par défaut grâce à une valeur universelle `nil`). Le statut exact de ces procédures d'initialisation diffère selon les langages : elles sont souvent désignées par le nom trompeur de « constructeurs » (tradition de C++ malheureusement reprise par JAVA).

Avec cette mécanique extrêmement simple, nous avons le principe d'une réalisation complète de notre idée d'objet. Il est difficile de faire mieux, et même de faire autrement : on finit toujours par voir réapparaître la notion de classe sous des déguisements divers.

Deux remarques :

- on ne peut créer d'instance que si la classe est présente en mémoire, et l'instance ne peut fonctionner que tant que la classe reste présente,
- un objet ne peut être instance que d'une seule classe, et, en principe, il ne change pas de classe au cours de son existence (sauf manœuvre exorbitante comme la primitive `become:` de SMALLTALK ; le recours à la réflexivité — voir § 1.3.3 — permet d'aller plus loin [Rivard, 1997]).

**Les classes comme représentation des concepts**

À l'intérieur de la machine, classes et instances ont la même consistance matérielle, à savoir des octets en mémoire. Les unes ne sont pas plus abstraites ou concrètes que les autres. C'est l'interprétation que nous faisons de l'implémentation esquissée ci-dessus qui assigne aux classes un statut d'abstractions (concepts) et aux autres un statut d'objets concrets (instances). Cette interprétation s'appuie sur une analogie : de même que le concept abstrait est unique vis-à-vis de ses réalisations concrètes, qui sont multiples, de même la classe est une, et ses instances sont (potentiellement) multiples. En outre, comme nous l'avons dit, les instances (objets informatiques) ont pour vocation de représenter des objets concrets du monde réel, elles occupent donc le pôle « concret » de l'opposition abstrait/concret, vouant les classes au pôle « abstrait ».

Or, comme nous l'avons observé, les programmeurs ressentent un besoin d'abstraction aigu. Du coup, nos classes, bien qu'elles aient été introduites comme une technique de réalisation pour notre projet d'objets, vont devoir jouer le rôle de représentation des concepts. Bien évidemment, elles présenteront à cet égard des insuffisances nombreuses. Mais il est honnête de prendre conscience de la difficulté de cette nouvelle tâche ! Tout un courant de recherches en intelligence artificielle lui est consacré, commençant par les réseaux sémantiques (dès 1968, voir *chapitre 10*), passant par KL-ONE [Brachman et Schmolze, 1985] et aboutissant aujourd'hui aux logiques de descriptions (voir *chapitre 11*).

Il s'ensuit un glissement sémantique de la plus haute importance : étant donné un concept, on lui associe naturellement l'ensemble de ses instances, que l'on appelle l'*extension* du concept. Pris en lui-même, un concept (par exemple le concept de chat) est certes autre chose que son extension (qui, elle, est un ensemble : l'ensemble des chats). Mais très souvent le nom du concept sert à désigner son extension, et, en somme, le concept est vu comme représentant son extension (quand on dit « le chat est un animal domestique »).

Dans la mesure où elles représentent des concepts, nos classes se voient donc chargées d'une valeur sémantique ensembliste en plus de leur signification littérale, qui prescrit comment leurs instances sont faites et comment elles se comportent : chacune représente l'ensemble de ses instances, effectivement présentes à un moment donné dans le système ou seulement potentielles.

Soulignons que ce glissement sémantique se produit de manière inconsciente, comme dans de nombreux phénomènes langagiers, car notre pensée passe naturellement de la description à la chose décrite. Il faut donc faire un effort de réflexion pour s'en rendre compte. Il en résulte pour la notion de classe une ambiguïté essentielle, qui est précisément la source de sa fécondité. Cette ambiguïté est d'ailleurs corroborée par la polysémie du mot classe. Dans plusieurs branches du savoir plus anciennes que l'informatique, une classe désigne une espèce particulière d'ensemble : ainsi en mathématiques (classes d'équivalence), dans les sciences de la nature (la classe des mammifères, voir *chapitre 15*), en sociologie (les classes sociales), etc.

Nos classes viennent ainsi s'ajouter à la panoplie dont disposent les informaticiens pour représenter des ensembles. On sait de reste qu'en informatique la notion d'ensemble pose problème. Pour représenter un ensemble infini en termes finis acceptables par une machine, il faut recourir à un programme descriptif, qui définit de manière algorithmique tous les éléments de l'ensemble (sous sa forme la plus simple, ce programme va expliciter les sous-entendus cachés sous les points de suspension qui apparaissent si souvent dans les notations mathématiques). Ce programme est appelé une *intension*, mot emprunté au vocabulaire de la philosophie [3]. Et les classes de la programmation par objets sont bien des programmes, descriptions intensionnelles, les seules choses que nous sachions réaliser avec nos drôles de machines...

**Réflexion *a posteriori***

Pour construire un nouvel objet, il faut indiquer à l'ordinateur comment il se compose, et pour ce faire, en attendant les interfaces de programmation graphique, il faut rédiger à l'usage de la machine un texte décrivant le procédé constructif. Dans ce texte apparaîtront les noms des objets composants, avec le statut de variables libres. Et, du coup, notre texte ne décrira plus un objet singulier, construit à partir de composants eux-mêmes singuliers, mais toute la famille des objets qui peuvent être produits par le même procédé en donnant aux variables libres d'autres valeurs : il devient bel et bien une classe.

Cette incapacité d'un texte à décrire une entité singulière sans automatiquement décrire une famille d'entités isomorphes est bien connue de la tradition philosophique : Aristote notait déjà qu'il n'y a pas de définition de l'individu [Brun, 1988, page 36]. On peut la rapprocher de la règle de généralisation en logique du premier ordre : si l'on a `P(x)`, où `P` est un prédicat et `x` une variable libre, alors on a universellement $\forall$`xP(x)`.

Ceci explique les difficultés infinies que rencontrent les tentatives de réaliser en informatique une approche basée sur des prototypes, qui seraient des instances concrètes à partir desquelles d'autres instances concrètes se définiraient par « recopie différentielle » (voir *chapitre 8*) : on y voit toujours reparaître des objets constructeurs qui, en fait, sont très analogues aux classes. Ce phénomène ne perd de son importance qu'en présence d'une grande quantité d'objets déjà existants, dont on peut oublier l'histoire, et qui fournissent la base d'un « concret préalable ». Mais cette situation n'est pas encore courante.

### 1.2.3 Héritage

La dualité entre intension et extension apparaît au grand jour avec la notion d'héritage. Bien entendu, les langages classiques ont pour en parler chacun leur vocabulaire et leurs mots-clés : EIFFEL et SMALLTALK parlent d'héritage, C++ de *dérivation* et JAVA d'*extension* (avec le mot-clé `extends`).

**Définitions : ajouter des attributs et des méthodes**

1. On dit qu'une classe `A` *hérite* d'une classe `B` si tous les attributs définis par `B` se retrouvent dans `A`, c'est-à-dire si la liste des noms de champs de `B` est

3. Terme sorti de l'usage courant, synonyme de *compréhension*, dit le dictionnaire Lalande [Lalande, 1926].

incluse dans celle de `A` et si le catalogue des méthodes de `B` est contenu dans celui de `A`.

2. Lorsque `A` hérite de `B`, on dit aussi que `A` est une *sous-classe* de `B` et que `B` est une *super-classe* de `A`.

3. La relation d'héritage est une relation d'ordre, qui se représente par un graphe sans circuit appelé le *graphe d'héritage*.

En ce qui concerne l'écriture des textes, en pratique, on définira une sous-classe `A` à partir de la super-classe `B` supposée déjà connue, en donnant seulement les noms de champs et les procédures supplémentaires « propres à `A` », sans reproduire le texte de `B`, accessible via un pointeur *ad hoc*. Matériellement, une sous-classe se présente donc comme un « fragment de classe », incomplet du point de vue sémantique, et qu'il faut compléter par sa super-classe.

Les termes de sous-classe et de super-classe demandent une justification. En effet, du point de vue intensionnel, le texte de la sous-classe contient celui de la super-classe ! En quoi est-elle donc *sous*-classe ? Parce que son extension est un *sous-ensemble* de l'extension de sa super-classe. Mais cette inclusion ensembliste ne va pas de soi ! Elle repose sur l'interprétation que nous donnons au fonctionnement du système. Elle se traduit au niveau des types par l'apparition de la relation de sous-typage. Voyons en quoi consiste cette interprétation.

*A priori*, un objet appartient à (l'extension d') une classe si et seulement si il a été créé à partir de cette classe. Les extensions de deux classes différentes sont donc tout simplement disjointes, et toute hiérarchie est exclue. Il faut par conséquent adopter une démarche plus raffinée que cette vision simpliste. Il serait désirable de pouvoir définir l'extension d'une classe comme l'ensemble des objets qui possèdent tous les champs et le comportement définis (intensionnellement) par cette classe. Les instances d'une sous-classe, qui possèdent tout cela avec quelque chose de plus, feraient ainsi de plein droit partie de cette extension, et nous aurions l'inclusion souhaitée. On pourrait épiloguer sur le bien-fondé de cette définition par condition nécessaire et suffisante portant sur la structure, mais nous admettrons ici qu'elle est, dans le présent contexte, « raisonnable ».

Malheureusement, elle est trop difficile à implémenter pour nos faibles moyens. Chaque instance d'une classe possède évidemment tous les champs et le comportement en question. Mais pour la réciproque, rien, dans les mécanismes que nous avons décrits, ne permet d'analyser la structure d'un objet. La seule information que nous sachions exploiter est la « marque de fabrique » de la classe, sous la forme de son adresse (ou de son nom) qui fait partie de l'objet. En fait, on peut très bien écrire deux fois la « même » classe sous deux noms différents, et le système les traitera comme deux classes totalement différentes, aux extensions disjointes. Ce problème est abordé de front, avec des moyens plus puissants, par les logiques de descriptions dans la théorie de la subsomption (voir *chapitres 10, 11 et 12*).

Nous revenons à une définition aisément implémentable en disant qu'un objet appartient à l'extension d'une classe si et seulement s'il a été créé à partir de cette classe *ou d'une de ses sous-classes*. L'inclusion cherchée est ainsi assurée par définition. Quant à l'implémentation, il suffit de prévoir dans la structure des classes un champ supplémentaire contenant l'adresse de la super-classe pour pouvoir répondre à la question.

#### Interprétation des messages : la recherche ascendante *look-up*

Pour que toute instance d'une sous-classe puisse être considérée comme une instance de la super-classe, le mécanisme d'interprétation des messages doit être complété :

- en ce qui concerne les champs supplémentaires introduits dans la sous-classe, ils sont simplement ajoutés à la fin de la liste donnée par la super-classe ;
- en ce qui concerne les messages :
  - si le sélecteur — nom de la méthode — apparaît dans le catalogue de la sous-classe, le corps de procédure correspondant est exécuté comme expliqué dans le paragraphe précédent (naturellement, seules les méthodes définies dans la sous-classe sont habilitées à utiliser les champs supplémentaires) ;
  - sinon, la recherche se poursuit dans la super-classe : si le nom y figure, le corps de procédure associé est exécuté, sinon la recherche se poursuit dans la super-classe de la super-classe (si elle existe), etc., jusqu'à arriver à une classe qui n'a pas elle-même de super-classe, ce qui déclenche une erreur à l'exécution.

Ce processus de recherche en remontant dans le graphe d'héritage (en anglais *look-up*) est très facile à implémenter sous l'hypothèse que l'héritage est simple : l'unicité de la super-classe se traduit par la structure du graphe d'héritage en forêt, ou ensemble d'arbres. En fait, on considère la plupart du temps que le graphe d'héritage possède une racine unique, qui correspond à une super-classe générale exprimant les propriétés communes à tous les objets du système. Le graphe d'héritage est alors connexe, et lorsque l'héritage est simple, il s'agit d'un arbre.

Si l'héritage est multiple, il faut définir un parcours ascendant du graphe d'héritage, ce qui peut donner lieu à des algorithmes savants [Ducournau et Habib, 1989] [Ducournau *et al.*, 1995]. En outre, il faut recourir à un système un peu plus compliqué qu'une simple numérotation pour assurer la correspondance entre les noms des champs et les déplacements en mémoire.

#### Redéfinitions

L'interprétation extensionnelle de l'héritage est légitime si l'on se borne à ajouter dans la sous-classe des attributs et des méthodes qui portent des noms ne figurant pas dans la super-classe : le procédé d'interprétation ci-dessus fonctionne alors sans ambiguïté, et l'on peut affirmer qu'une instance de la sous-classe se comporte en tout point comme une instance de la super-classe (sauf en ce qui concerne les erreurs à l'exécution) et que par conséquent, il est licite de la considérer comme étant elle aussi une instance de ladite super-classe. Plus précisément, on peut dire que tout le comportement significatif (c'est-à-dire non erroné) des instances de la super-classe est reproduit sans changement par celles de la sous-classe, qui ainsi « se qualifient » pour appartenir à la super-classe.

Les difficultés commencent si l'on veut introduire des homonymes : on parle alors de redéfinition ou de masquage de l'attribut ou de la méthode déjà connu sous le même nom. Il s'agit en fait d'une idée fort naturelle, très fréquemment employée dans la pratique. Il n'est malheureusement pas facile d'en rendre compte sur un plan théorique.

La redéfinition d'attributs est possible dans certains langages typés. Le plus souvent, le compilateur impose une certaine cohérence entre les deux définitions, à savoir que le type de l'attribut redéfini (dans la sous-classe) soit compatible avec le type déclaré dans la super-classe. Pour un langage non typé (ou « typé dynamiquement ») comme SMALLTALK, la redéfinition d'attributs n'a pas de sens évident (il faut se méfier de l'ingéniosité des programmeurs SMALLTALK, ils sont fort capables d'en trouver un !). Aussi les différentes versions du langage ont-elles des attitudes qui vont de la tolérance sans commentaire à l'interdiction brutale.

Mais c'est la redéfinition de méthodes qui est universellement employée et c'est elle qui pose de graves problèmes. On peut immédiatement étendre le procédé de recherche ascendante (*look-up*) au cas où l'on donne dans la sous-classe une nouvelle définition à un nom de procédure déjà défini dans la super-classe : en général, on souhaite associer à ce nom un comportement légèrement différent de celui que spécifie la super-classe pour l'adapter au perfectionnement introduit par la sous-classe, par exemple pour tenir compte d'un attribut supplémentaire.

La difficulté saute aux yeux : en présence de redéfinition, le comportement significatif des instances de la sous-classe va être différent de celui des instances de la super-classe, et, en bonne logique, l'on ne pourra donc plus conclure comme précédemment que l'extension de la sous-classe est incluse dans celle de la super-classe. Pour maintenir cette conclusion, il faudrait étendre la notion de « différence significative de comportement » pour pouvoir dire que les comportements des deux classes ne diffèrent pas significativement ... Pas très commode ! On devine les difficultés qui nous attendent quand nous voudrons confier cette tâche à un contrôleur de types (voir plus loin, en 1.3.2 les questions de covariance et de contravariance).

### Sous-typage et lien dynamique

Du point de vue du compilateur, la recherche ascendante (*look-up*) est source d'inefficacité : le compilateur travaille non sur les objets eux-mêmes, comme l'interprète que nous avons esquissé ci-dessus, mais sur des expressions où les objets apparaissent comme valeurs de variables, le compilateur lui-même n'ayant connaissance que de la classe des variables, supposée déclarée par le programmeur. En l'absence d'héritage, la connaissance de la classe suffit au compilateur pour déterminer « statiquement » la procédure à appliquer en réponse à un message.

Mais si la classe déclarée (disons `B`) possède des sous-classes (disons `A`), alors une variable déclarée de classe `B` peut très bien contenir en fait, à un moment ou à un autre du calcul, une instance de la sous-classe `A` : puisque toute instance de `A` est aussi instance de `B` ! Attention ! C'est ici que l'interprétation extensionnelle de l'héritage se transforme en exigence de sous-typage : dire que toute instance de `A` est aussi instance de `B` entraîne que partout où une instance de `B` peut légitimement apparaître, on peut lui substituer une instance de `A` sans changer le comportement du programme. Dans le langage de la théorie des types, cette possibilité de substitution signifie que, en tant que type, la sous-classe `A` est sous-type de `B` : c'est la définition même du sous-typage. On verra plus loin les redoutables conséquences de cette exigence, laissons de côté pour l'instant le problème du contrôle de types.

Dès lors, pour peu que la méthode visée par le message à compiler ait été redéfinie dans la sous-classe `A`, le compilateur doit activer soit la procédure déclarée dans la super-classe `B`, soit celle qui provient de la redéfinition dans la sous-classe. Et ce n'est que dynamiquement, à l'exécution, que cette incertitude sera levée. Ce problème est connu sous le nom de *liaison dynamique*, notamment dans la tradition de C++ (voir *chapitre 3*). Il a fait l'objet de tout un travail d'implémentation, voir par exemple [Ducournau, 1997] ou l'article [Zendra *et al.*, 1997] sur le compilateur SMALL EIFFEL.

### Classes abstraites

On s'est aperçu très tôt que les classes jouaient au moins deux rôles, à la fois créatrices d'objets utilisables (instances) et détentrices d'informations exploitables par héritage (sous-classes). Et de plus, que certaines classes ne pouvaient jouer utilement que le second rôle : conçues pour être complétées par des sous-classes, elles ne possédaient pas assez d'informations pour engendrer des instances « viables ». Dans la tradition SMALLTALK, on les appelle des classes *abstraites*, en JAVA aussi, alors qu'en EIFFEL elles sont dites *retardées* (*deferred*). Les classes destinées à la procréation directe sont au contraire dites *concrètes*.

C'est l'exigence de contrôle statique qui a introduit un traitement particulier pour ces classes. En SMALLTALK, une classe abstraite est traitée par le compilateur comme une classe ordinaire : si le programmeur lui fait engendrer une instance, il prend le risque d'une erreur ultérieure s'il essaye de faire fonctionner cette instance — tant pis pour lui. Le mécanisme d'héritage suffit, que l'on interprète la complétion de la classe abstraite par un *ajout* de nouvelles méthodes ou par la *redéfinition* de méthodes qui étaient définies de manière non opératoire (définitions « bidon », se bornant à fixer le sélecteur, comme la définition standard `self SubclassResponsibility`).

Il est évident qu'un contrôle statique est possible, empêchant le programmeur de créer des instances qui ne jouiraient pas de toutes leurs facultés : les classes possédant des méthodes insuffisamment définies sont marquées « abstraites », avec interdiction de les instancier, et dans les sous-classes prétendues concrètes, le compilateur est en mesure de vérifier que les définitions sont bien complètes. Il suffit pour cela d'étendre la syntaxe afin de donner un statut aux définitions « bidon » (mot-clé `deferred` en EIFFEL, expression `= 0` en C++ (voir *chapitre 3*).

Cette variation très simple par rapport à l'héritage standard se révèle être un merveilleux outil de conception, voir par exemple *chapitre 3*. On notera que la distinction classe abstraite / classe concrète recoupe exactement la distinction aristotélicienne entre le genre et l'espèce [Brun, 1988, page 36]. Elle répond à l'observation courante selon laquelle on ne rencontre jamais un mammifère, mais toujours soit un chat, soit un chien, soit un homme, etc. L'espèce apparaît la dernière dans la hiérarchie des genres (en systématique, on dirait "des taxons", voir *chapitre 15*), auquel on ne peut plus ajouter que des accidents qui définissent des individus.

**Limitations**

La relation d'héritage entre classes, telle que nous l'avons décrite, est censée satisfaire le besoin de hiérarchisation des concepts. Il est manifeste qu'elle n'y réussit que d'une manière très partielle, puisque les seuls cas de particularisation qu'elle traite sont ceux qui proviennent d'une complication (ou d'un perfectionnement !), avec ajout d'attributs et/ou de méthodes. Le problème général se pose ainsi : étant donné deux ensembles d'objets `A` et `B`, avec `A` contenu dans `B`, peut-on trouver des définitions intensionnelles pour ces deux ensembles, sous forme de classes telles que nous les avons décrites ici, sur lesquelles le compilateur (ou le contrôleur de types) pourra effectivement conclure que toute instance de `A` est aussi instance de `B` ? En général, ce problème se pose alors qu'on a déjà une classe décrivant `B`, et il s'agit de trouver une description de `A` sous forme de sous-classe.

La réponse est en général non. Voici deux contre-exemples.

1. Prenons pour `B` la classe des rectangles et pour `A` celle des carrés. La manière intensionnelle ordinaire de décrire un rectangle est de le définir par deux points, alors qu'un carré se définit par un point et une longueur : il faudrait un démonstrateur de théorèmes puissant pour conclure automatiquement sur ces bases que tout carré est un rectangle ! La difficulté vient de ce qu'un carré n'est pas un rectangle « avec quelque chose en plus », mais bien un rectangle « tel que deux côtés consécutifs soient égaux », cette dernière restriction échappant complètement à nos possibilités descriptives.
2. À partir d'une hypothétique classe `Homme`, on imagine comment écrire une sous-classe `Barbu` en ajoutant un attribut et les méthodes afférentes. Mais comment décrire par des sous-classes les notions d'unijambiste ou de manchot ? Nous savons programmer la présence, mais non l'absence. Voyez dans un autre registre les problèmes de la négation en PROLOG ; il s'agit essentiellement de la même difficulté.

Pourtant, on considère depuis Aristote que toute définition est constituée du genre, qui est commun à plusieurs espèces, et des différences par lesquelles se distinguent les espèces [Brun, 1988, page 36]. Dans notre vocabulaire, ceci revient à dire qu'une classe se définit par la donnée de sa super-classe (son genre) et par ses différences spécifiques : or, Aristote aurait sans doute accepté comme différence valable la propriété que deux côtés consécutifs soient égaux, ou qu'un membre soit absent, alors que justement ces exigences sont inexprimables dans notre langage de programmation. Le problème est donc moins la structure générale de la définition par genre et différences que le domaine des différences exprimables dans le langage, qui pour nous se limite à l'ajout de méthodes et d'attributs.

On voit donc que le besoin de hiérarchisation n'est que très imparfaitement satisfait par le mécanisme simple de l'héritage. Mais ce besoin est tellement fort que cette solution partielle a été saisie avidement, quitte à épiloguer sur les difficultés qu'elle engendre.

## 1.3 Discussion

### 1.3.1 Objets et valeurs

Il s'agit d'une distinction fondamentale, qui mérite d'être mise au premier plan de nos préoccupations. Qu'elle ait été jusqu'ici largement occultée par les tenants de la programmation par objets est en soi un intéressant sujet de réflexion.

#### Que les objets sont essentiellement réactifs, donc mutables

Qu'un objet soit doué de capacités réactives de par son comportement est une propriété fondamentale, indispensable pour lui donner l'individualité, la personnalité, qui vont créer chez le programmeur l'illusion d'existence concrète qui est le ressort de toute l'approche objet. Bien que notre objet informatique se réduise à une séquence de bits en mémoire, il faut qu'il déclenche les mêmes processus psychologiques qui fonctionnent dans notre intellect lorsque nous manipulons des objets réels. Nous éprouvons l'existence des objets réels parce qu'ils nous résistent, parce qu'ils réagissent, parce qu'ils répondent à nos enquêtes sensorielles en nous renvoyant des stimuli (visuels, tactiles, etc). Dans l'univers de la machine, l'acte de « toucher » ou de « sentir » un objet se transpose en un envoi de message, et la sensation correspondante se traduit par la réponse de l'objet. C'est donc en dotant nos entités informatiques d'une capacité de réaction que nous les faisons exister : *il fonctionne donc il existe*.

Notons ici que cette réactivité entraîne la plupart du temps que l'état de l'objet puisse changer au cours du temps. On pourrait penser que certains objets rigides ne changent jamais d'état, mais, en fait, la modélisation informatique ajoute souvent à l'état intrinsèque des variables comme la position de l'objet : un point est un bon exemple d'objet immuable, sauf si justement son état inclut ses coordonnées, et si on veut le faire bouger sur l'écran ... Nos objets sont donc de manière essentielle des entités mutables, ce qui entraîne des conséquences déplaisantes pour la théorie des types, comme on le sait (voir notamment la thèse de X. Leroy [Leroy, 1992]).

Mais il y a plus grave : vu qu'il existe de toute évidence, dans notre univers mental, des entités non réactives et non mutables (par exemple les nombres, plus généralement les abstractions mathématiques), l'observation précédente a comme corollaire que, dans un système de représentation équilibré, il doit y avoir aussi des structures qui ne sont pas des objets, ce que, dans une certaine tradition, on appelle des valeurs — notamment en bases de données (voir *chapitre 5*). Cette opposition entre les concepts d'objet et de valeur a été parfaitement analysée dans par B. Mac Lennan en 1982 [MacLennan, 1982], donc au moment même où l'approche objet entamait sa carrière.

**Objets et valeurs : halte au « tout objet » !**

C'est d'abord une affaire de modélisation : l'idée que l'objet est un individu connu par son adresse, et dont l'état peut changer, ne s'applique pas également bien à toute espèce d'entités. Par exemple, s'il est clair qu'une personne — ou une organisation — est un objet à part entière, identifié de manière unique, qu'en est-il de la date de naissance de la personne — ou de l'adresse de l'organisation ? En faire un objet à part entière expose au grave danger de voir une rectification anodine modifier les données relatives à d'autres personnes — ou organisations — ayant la même date de naissance (ou la même adresse), sauf à s'astreindre à une discipline particulière de recopie dans l'attribution des dates et des adresses. Or une telle menace est intolérable dans l'éthique des bases de données. Comme on le voit, la question de fond est : qu'entend-on au juste par « avoir la même date de naissance » (ou « avoir la même adresse ») ? Est-ce bien la même chose que « avoir la même mère » ou « travailler dans la même compagnie » ? On trouvera dans *chapitre 8* (section 6) une analyse des problèmes liés à l'identité des objets qui prolonge la présente.

Il vaudra donc mieux, dans certains cas, considérer que dates et adresses ne sont pas des objets, mais des valeurs, et construire une mécanique adéquate pour les traiter (implémentant une sémantique de valeur, dit-on en bases de données, voir *chapitre 5* et [Capponi, 1995]). Ainsi, des difficultés seront évitées, qui ne sont que désagréables pour les uns, mais carrément intolérables pour d'autres.

Évidemment, cette distinction complique le langage, et elle va à l'encontre du principe du « tout objet » revendiqué par des langages comme SMALLTALK et EIFFEL. Historiquement, ce principe simplificateur a pu être, il y a quelques années, générateur de progrès. Mais aujourd'hui nous pouvons nous permettre d'observer que, dans les faits, il ne s'applique jamais d'une manière complète : il existe toujours des classes bizarres (les nombres, les booléens, les caractères), dont l'implémentation échappe au modèle standard et qui n'entrent dans le cadre syntaxique uniforme que par une pétition de principe abusive. La distinction entre objets et valeurs se rencontre partout. Mais, dans certaines traditions, elle est mise au premier plan (notamment, dans celle des bases de données), et dans d'autres, on choisit de la cacher soigneusement (SMALLTALK), ou de la déguiser sous des considérations d'efficacité (voir la notion d'*objet expansé* en EIFFEL, *chapitre 2*, section 4). *Caveat programmator*[4] !

### 1.3.2 Classes et types

Comme nous l'avons indiqué plus haut (§ 1.1.2), leur interprétation extensionnelle donne aux classes une vocation évidente à servir de types. Et, dès lors, la relation d'héritage induit une relation de sous-typage (définie par la règle de substitution, voir paragraphe 1.2.3). EIFFEL a été le premier langage à tenter sur ces bases l'aventure d'un typage fort (voir *chapitre 2*). Le résultat est, disons, plus compliqué que prévu. Si l'on veut conserver des distinctions simples, on arrive avec [Cook *et al.*, 1990] à la conclusion que l'héritage n'est pas du sous-typage (*Inheritance Is Not Subtyping*).

4. Que le programmeur prenne garde !

On est ainsi conduit à séparer les notions de classe et de type (appelé *interface* en JAVA) et à les relier par un lien d'implémentation : un type se réduisant à une signature de méthodes[5], une classe qui implémente ce type (il peut y en avoir plusieurs) doit fournir un système d'attributs et les corps de procédure idoines réalisant la signature. Nous ne discuterons pas cette démarche, qui marque un retour à l'approche par types abstraits. Nous répéterons seulement quelques observations élémentaires attestant que, si l'on s'en tient au point de vue naïf que nous avons adopté ici, les choses ne se passent pas comme on le voudrait [Cook *et al.*, 1990].

**Contrôle statique**

Rappelons l'idée du contrôle de types statique, à la compilation : les objets apparaissent dans le texte d'un programme sous la forme d'expressions, dont on voudrait pouvoir prédire le type indépendamment de l'exécution du code. Il faut pour cela édifier une théorie des types qui doit rendre possible la détermination du type d'une expression à la compilation. L'entreprise est remplie d'embûches si l'on veut un typage sûr, qui garantisse qu'à l'exécution aucune erreur due à une incompatibilité de type ne se produira. Par opposition, on parle d'un *typage dynamique* (par exemple pour SMALLTALK) lorsque la non conformité des classes des objets avec les messages qui leur sont envoyés n'est détectée qu'à l'exécution.

Quoi qu'il en soit, il est clair que le typage statique impose au programmeur de s'expliquer beaucoup plus à fond que ne le demande le typage dynamique. Cette exigence peut être ressentie comme vexatoire ou comme salutaire, suivant le contexte. En particulier, elle rend pratiquement indispensable le recours à l'héritage multiple et aux *mixins*. Rappelons qu'on appelle *mixin*, après [Stefik et Bobrow, 1986] une classe qui ne spécifie qu'un genre de comportement très limité (par exemple, le fait de porter un nom), et que l'on peut à volonté insérer dans un graphe d'héritage multiple pour « ajouter » ce comportement à toute une hiérarchie de classes définies par ailleurs. On s'attend à ce que le vocabulaire correspondant soit unique dans le système, pour éviter les conflits qui sont la plaie de l'héritage multiple.

En effet, considérons un fragment de code exprimant une opération qui doit pouvoir s'appliquer à des instances appartenant à des classes différentes — par exemple, une visualisation. Dans ce fragment de code, une même variable doit pouvoir contenir lesdites instances et recevoir les messages adéquats — dans notre exemple, le message `voir`. Pour pouvoir déclarer cette variable, il faudra lui attribuer un type qui garantisse au compilateur que les envois de messages en question sont bien licites. À moins de disposer d'opérations sur les types comme la réunion ensembliste, il faudra donc créer une classe abstraite où seront déclarés lesdits messages, avec les attributs afférents, et dont devront hériter les classes concrètes des instances candidates à l'utilisation.

En d'autres termes, il faudra expliciter par une classe *mixin* le point de vue particulier sous lequel notre fragment de code regarde les objets — dans notre exemple, la visualisation. C'est là une technique classique, que le typage statique rend obligatoire.

5. Les interfaces de JAVA autorisent en plus la définition de constantes partagées par toutes les classes qui implémentent l'interface ; ce supplément ne change pas le fond de la question.

**Contravariance des arguments**

Cette question touche le type que l'on peut assigner à une méthode lorsqu'on la redéfinit dans une sous-classe. Elle fait apparaître un conflit entre la cohérence de la théorie, garante de l'économie de pensée du programmeur, et les intentions que l'on souhaite exprimer à travers les déclarations de types.

Rappelons que l'interprétation extensionnelle de l'héritage, qui dit que chaque instance d'une sous-classe est aussi instance de sa super-classe, se traduit du point de vue des types par la règle de substitution : étant donné une classe `B`, une sous-classe `A` de `B`, dans toute construction du langage où apparaît une instance `b` de `B`, soit `C[b]` (avec `C` comme *contexte*) et qui est déclarée correcte par le contrôleur de types, on peut substituer à l'objet `b` n'importe quelle instance `a` de la sous-classe `A`, sans que le contrôleur de types trouve à redire à la nouvelle construction `C[a]`.

Soit alors `p(x)` une procédure à un argument de type `X` définie dans `B` et redéfinie dans `A` avec `X'` comme type de son argument. Considérons la construction (fragment de texte de programme) `C[b] = b.p(u)`, où `u` est un argument effectif pour la procédure `p`. Cette expression est supposée correcte, l'argument `u` appartient donc au type `X` déclaré dans `B`. D'après la règle de substitution, l'expression `C[a] = a.p(u)` doit aussi être correcte. Or dans cette dernière, `p` désigne la procédure redéfinie (en vertu du lien dynamique), et, par conséquent, le contrôleur de types exige que `u` appartienne au type `X'`. Ceci étant vrai pour tout $\mathtt{u} \in \mathtt{X}$, cette exigence se traduit par l'inclusion de `X` dans `X'`.

On déduit de ce raisonnement fort simple que, lorsqu'on redéfinit une procédure dans une sous-classe, les nouveaux types de ses arguments doivent contenir les types stipulés dans la définition initiale. Cette règle, bien connue en théorie des types, est appelée « règle de *contravariance* des arguments ». Elle énonce en effet que, pour être « plus spéciale », une procédure doit avoir une domaine de définition « plus général ».

En revanche, le même raisonnement appliqué à une fonction au lieu d'une procédure montre que le domaine du *résultat* de la méthode redéfinie dans la sous-classe doit être inclus dans le domaine déclaré initialement, ce qui se traduit par la règle de « *covariance* des résultats ».

Or, l'intuition commune voudrait plutôt que la spécialisation d'une procédure redéfinie dans une sous-classe s'exprime en restreignant son domaine de définition, c'est-à-dire en déclarant `X'` inclus dans `X`, selon la règle dite de *covariance*. La plupart des concepteurs de langages ont décidé d'adopter cette règle, ce qui les oblige à introduire un appareillage spécifique pour éviter les incohérences dans le contrôle de types. Le meilleur exemple est fourni par la distinction entre *class-level* et *system-level* en EIFFEL, avec la théorie des *catcalls* (voir *chapitre 2*, section 8.2).

Cette contradiction entre la règle théorique simple qui gouverne le sous-typage et l'usage qu'on veut en faire explique la prudence de JAVA en ce qui concerne le profil des méthodes redéfinies : ce langage interdit tout simplement de changer de profil par rapport à la déclaration initiale, ni quant aux arguments, ni quant au résultat.

### Types récursifs

La contravariance des arguments est la cause de maux innombrables. En voici un exemple bien connu.

L'expression « types récursifs » fait immédiatement penser aux arbres. Mais, dans le contexte de l'approche objet, le type associé à une classe devient récursif dès que le nom de la classe intervient dans son code, ne serait-ce que comme type d'argument ou de résultat dans une méthode. L'exemple classique de l'égalité mérite d'être médité.

Supposons que nous souhaitions doter une de nos classes `B` d'une méthode destinée à tester l'égalité de deux de ses instances. Elle aura tout naturellement pour signature : `egale(x:B):bool`, et le type associé à `B` sera récursif. Si, à présent, nous écrivons une sous-classe `A` de `B`, nous allons souhaiter redéfinir la méthode `egale`, pour tenir compte d'un supplément de structure, en `egale(x:A):bool` — ce qui est contraire à la contravariance !

Outre les remèdes généraux qui ont été proposés, il faut mentionner ici l'approche par *surcharge* : doter la classe `A` de deux méthodes `egale`, l'une de signature `egale(x:B):bool` (héritée de `B`), l'autre de signature `egale(x:A):bool` (redéfinie dans `A`), le choix entre les deux méthodes étant fait sur le type de l'argument à l'appel. On est alors dans le cadre des *multi-méthodes* de CLOS, où l'objet destinataire d'un message n'a plus le privilège du choix de la procédure à employer, et n'est plus que l'un des arguments de l'appel, le choix de la procédure effective se faisant sur la base des classes de tous les arguments. Cette généralisation qui semble au premier abord purement technique modifie en fait profondément le style de programmation : on revient au style fonctionnel, cette fois dans un cadre élargi, où les entités manipulées ne sont plus des données inertes, mais des objets complexes, le cas échéant réactifs, et classés de manière hiérarchique (voir [Habert, 1995]).

### Mutabilité et généricité

Voici pour finir un exemple très simple montrant comment les exigences de typage cohérent en présence de généricité paramétrique (voir § 1.1.2) peuvent conduire à des situations inattendues. L'intuition ordinaire pousse à croire que, si `A[X]` est une classe générique, et si `U` et `V` sont deux classes concrètes, U héritant de `V`, alors le type `A[U]` est sous-type de `A[V]`.

Supposons que la classe `A[X]` soit celle de la pile générique et qu'elle possède une procédure `push(x)` à un argument. Soit `q` une variable déclarée de type `A[V]`, `v` une variable déclarée de type `V`, et considérons l'instruction `q.push(v)` : elle doit certainement être bien typée. Soit `p`, instance de `A[U]`, une « pile de `U` » : puisque `A[U]` est sous-type de `A[V]`, l'objet `p` peut être donné comme valeur à la variable `q`. Supposons à présent que la variable `v` ait pour valeur une instance « propre » de la super-classe `V` (c'est-à-dire, qui n'est pas instance de la sous-classe `U`). L'exécution de l'instruction `q.push(v)` aura pour effet de rompre l'homogénéité de la pile `p` en y insérant un élément qui n'est pas de type `U`. C'est précisément le genre d'erreurs que le contrôle de types est censé éviter !

La morale de l'histoire est que `A[U]` ne peut pas être sous-type de `A[V]`. La cause en est que l'argument de `push` (déclaré dans la classe générique `A[X]`) est contraint (par l'effet que produit la procédure, qui modifie l'état de son destinataire) à être de type `X`, ce qui est incompatible avec la contravariance des arguments.

### 1.3.3 Classes et métaclasses

Un des aspects les plus stimulants de l'approche objet est la tournure que prend la recherche en matière de réflexivité. Comme on sait, cette manière d'aborder les langages de programmation est née en programmation fonctionnelle. Avec les objets, elle s'introduit d'une manière très naturelle, comme on va l'esquisser ici.

**Les classes comme objets**

Le vieux problème du statut des Universaux (en ce qui nous concerne, celui des concepts abstraits) se pose en des termes renouvelés dans le cadre de l'approche objets. Il s'agit de savoir si les concepts existent réellement dans le monde (position dite réaliste) ou s'ils n'apparaissent que dans notre discours sur le monde (position nominaliste), de savoir s'ils sont des choses ou seulement des mots. Dans notre image informatique, cela revient à savoir si les classes sont des objets (manipulables comme tels) ou seulement des textes (plus ou moins compilés). Sur ce débat qui remonte à la critique de Platon par Aristote, on trouvera une information exhaustive dans le livre d'Alain de Libera « La querelle des Universaux » [Libera, 1996]. La question est difficile : Aristote lui-même, tout en déniant aux concepts abstraits la qualité de sujet, leur accordait celle de substance — substances secondes, certes, mais néanmoins substances [Brun, 1988, page 32].

Dans le cadre de l'approche objets, le contexte se précise. Partant de la position nominaliste standard en informatique, on observe deux mouvements conjoints vers le réalisme. D'une part, dans la pratique de la programmation, on a souvent besoin de traiter les classes comme des objets — disons qu'il est commode d'adopter la même syntaxe pour les classes et pour les instances (cela simplifie l'écriture), et que dès lors il est souhaitable d'unifier aussi la sémantique. D'autre part, le fait que les classes soient des entités informatiques ne laisse aucun doute (surtout pour des programmeurs ayant subi l'influence de LISP, langage éminemment réaliste) sur la possibilité matérielle de les manipuler, pour peu qu'on en prenne la décision. Il y a donc un besoin, la possibilité technique est là, le réalisme devrait triompher ... Pourtant, le débat n'est pas clos.

Il y a d'abord des arguments « idéologiques » sur la nécessité ou l'inutilité de manipuler les classes à l'exécution. Si les classes sont vues avant tout comme des types, donc comme des instruments destinés à un contrôle statique effectué par le compilateur, on peut légitimement considérer qu'elles ont « disparu » au moment de l'exécution, et que l'idée même de les modifier ou d'en créer de nouvelles par programme est aussi incongrue que celle du mouvement en l'absence de moteur pour un disciple d'Aristote. Si, au contraire, on voit en elles des outils faits pour construire des objets, il en va tout autrement : les outils sont des objets particuliers, que l'on répare, perfectionne, voire que l'on construit dans l'atelier même. Dès lors, rien ne s'oppose à ce que la structure de ces outils soit décrite sous forme de classes. Comme les instances de ces classes sont elles-mêmes des classes, on les distingue en les appelant *métaclasses*, par un emploi du préfixe *méta* courant en informatique. Sur le modèle du rapport entre physique et métaphysique, mais plus précisément en parallèle avec la distinction que les logiciens font entre langage et métalangage, le « niveau méta » désigne un niveau de représentation où sont définis les cadres de la représentation du niveau de référence[6]. Les classes sont ainsi au « niveau méta » par rapport aux instances, et les métaclasses par rapport aux classes.

Ces différences que nous qualifions d'idéologiques viennent en fait recouper le partage entre interprétation et compilation, ou plus exactement (car plus aucun langage opérationnel n'est strictement interprété) entre compilation incrémentale et interactive, façon SMALLTALK, et compilation séparée style EIFFEL ou C++. Dans le premier cas, où la tradition LISP est vivace, la matérialité de la représentation des classes à l'exécution ne pose point de problème, ni par conséquent leur élévation au rang d'objets de plein exercice. Dans le second, l'intervention du compilateur réduit les classes à des tables de *look-up* pour la liaison dynamique ; il est donc beaucoup moins facile de reconnaître leur existence comme objets véritables. Il est intéressant de noter que JAVA, qui dans sa première version se rangeait du côté nominaliste, devient nettement réaliste avec le JDK 1.1 et le *package* `Java.reflect`.

Il se pose aussi des problèmes d'ordre strictement logique. Notamment, celui de la régression à l'infini : si toute classe est un objet, une métaclasse est elle aussi un objet, instance d'une méta-métaclasse, etc. Il est intéressant de noter que cet argument a été employé par Aristote dans sa critique de Platon (argument dit « du troisième homme », [Brun, 1988, page 30] [Libera, 1996, page 75]). Dans la formulation informatique, ce problème devient celui du *bootstrap* (amorçage). Tout système qui admet les métaclasses doit le résoudre d'une manière ou d'une autre (voir [Masini *et al.*, 1989, sections 2.5 et 5.4], et en dernier lieu la thèse de Fred Rivard [Rivard, 1997]).

### Réification et réflexivité

Dans un langage à objets, il n'y a pas que les classes dont on puisse se demander ce qu'elles sont. Les messages, notamment, ne sont en général pas des objets, et pourtant il est souvent utile de pouvoir les « attraper au vol » pour les analyser et, le cas échéant, leur appliquer un traitement adapté. À l'intérieur des classes, le statut des méthodes n'est pas clair : peut-on demander à une méthode, par exemple, combien elle accepte d'arguments ?

Mieux qu'avec le problème des métaclasses, on voit apparaître avec ce genre de questions la notion fondamentale de réification, clef de voûte de toute l'approche objet. Un message par exemple est une entité qui n'a pas normalement le statut d'objet. Pure forme syntaxique dans le texte, indiscernable de l'opération d'envoi qui le met en œuvre, il est traduit en une séquence d'instructions exécutables et n'a pas d'individualité repérable à l'exécution : le message n'apparaît que dans un envoi de message, donc dans un acte. Toutefois, nous concevons son existence objective :

6. Rappelons que *meta* est une préposition grecque (μετὰ) qui signifie « avec » ou « après » selon qu'elle gouverne le génitif ou l'accusatif. Le sens qu'elle prend dans « métaphysique » vient de ce que, dans l'ordre traditionnel des œuvres d'Aristote, les livres qui traitaient de la « philosophie première » venaient après les livres de physique — μετὰ τὰ φυσικά βιβλιά.

à nos yeux, le message a une structure, un objet destinataire, des objets arguments, un sélecteur. Dans certains cas, nous souhaitons pouvoir changer le point de vue du système d'exécution et obtenir qu'il considère effectivement le message comme un objet, conformément à la vue que nous en avons, afin de l'examiner et de décider sur son sort (par exemple, pour le ranger dans la boîte à lettres d'un acteur avec la priorité convenable).

L'opération qui fait passer du message « en acte » à l'objet analysable qui le représente s'appelle la *réification* (du latin *res*, chose : on pourrait dire *chosification*). Elle est plus ou moins disponible suivant les systèmes. En SMALLTALK, le compilateur réifie le message en cas de refus par le destinataire, et il utilise l'objet-message ainsi synthétisé dans son traitement d'erreur : il existe toute une technique d'implémentation fondée sur les possibilités ainsi ouvertes (par redéfinition de la méthode `doesNotUnderstand:`).

L'opération inverse, qui replonge un objet dans le flot de calcul, s'appelle *réflexion*. Elle apparaît dans la démarche générale connue sous le nom de réflexivité, notamment en programmation fonctionnelle, mais dans l'approche objet elle reste invisible. En effet, les objets du « niveau méta » sont activés comme les objets du niveau de référence, sans distinguer différents niveaux d'interprétation.

Dans le monde des objets *stricto sensu*, la mécanique est simple, et il n'y a pas grand'chose à réifier. La tentative extrême dans ce domaine reste le *Meta-Object Protocol* (MOP) de CLOS [Kiczales *et al.*, 1991]. En revanche, dans le monde des acteurs et des objets distribués, l'envoi de message revêt des formes variées (envoi avec ou sans attente de réponse, *broadcast*, envoi différé, etc), donnant à l'approche réflexive une vaste carrière (voir en dernier lieu la thèse de Claude Michel [Michel, 1997]).

## 1.4 Conclusion

Nous venons d'introduire le terme de *réification* dans un contexte strictement technique, dans l'espoir de faire « toucher du doigt » la nature de cette transformation. Bien évidemment, cette notion est d'un emploi très général : toute l'approche objet repose sur la possibilité de représenter les entités du domaine d'application envisagé par des objets. Mais certaines entités ne prennent pas « naturellement » le statut d'objet, de chose ! Un geste, un mouvement, sont-ils des objets ? Un avion est clairement un objet (compliqué), mais un vol (au sens des agences de voyage) ? une réservation ? Question de point de vue ? En vérité c'est une question de modélisation (voir *chapitre 4*). Une modélisation réussie réifie à bon escient, ce qui n'est pas toujours facile.

Le progrès des techniques rend accessibles des applications de plus en plus complexes, abordant des domaines de plus en plus éloignés des territoires bien balisés du calcul scientifique et de la gestion. Ces ambitions nouvelles conduisent souvent à des réifications hasardeuses. Plutôt que de chercher un exemple « réel » qui demanderait de longues justifications, je voudrais illustrer mon propos par une sorte d'apologue tiré du livre [Libera, 1996, pages 51–52]. Il s'agit d'un passage du Ménon de Platon, où Socrate exerce sa maïeutique sur un esclave et arrive à lui faire dire que les abeilles ont une propriété en commun à l'égard de laquelle elles sont indistinguables — nous dirions, qu'elles sont instances d'une même classe. Et Socrate ajoute : *Eh bien c'est pareil pour les vertus ! Même s'il y en a beaucoup et de toutes sortes, elles possèdent du moins une seule forme caractéristique identique chez toutes sans exception, qui fait d'elles des vertus. Une telle forme caractéristique est ce qu'il faut bien avoir en vue pour répondre à qui demande de montrer en quoi consiste la vertu* (Traduction de M. Canto, collection Garnier-Flammarion).

En d'autres termes, pour Socrate les vertus sont elles aussi des objets, instances de la classe « Vertu » ! Il me semble que notre expérience collective nous suggère qu'écrire une classe « Abeille » en vue d'un projet informatique bien défini est chose *a priori* faisable, mais qu'il en va autrement pour les vertus, plus précisément que réifier la vertu est un procédé si épouvantablement réducteur qu'on ne peut l'envisager hors d'un projet informatique lui-même tellement réducteur qu'il vaut mieux ne plus parler de vertu ...

Plus profondément, il nous faut réfléchir sur les limitations intrinsèques de la démarche aristotélicienne induisant les concepts généraux à partir de l'expérience sensible que nous avons des individus, et dont la modélisation par objets est l'expression informatique directe. Les révolutions qui ont marqué l'histoire de la pensée scientifique et qui dans certains domaines ont disqualifié l'aristotélisme (notamment en physique) nous informent que le sens commun n'a pas toujours raison, et qu'il faut parfois passer par des cheminements cachés.

# Première partie

# Génie logiciel et objets

# 2

# Eiffel : une approche globale pour programmer et réutiliser des composants logiciels

Ce chapitre présente quelques traits saillants des systèmes EIFFEL, ce qui comprend le langage, sa méthode de programmation implicite, ses bibliothèques de composants et des ateliers de GLAO pour développer et réutiliser des composants logiciels de qualité. Après une discussion rapide de la situation d'EIFFEL face à ses concurrents, nous donnerons un aperçu de mécanismes EIFFEL parmi les plus caractéristiques, en insistant plus particulièrement sur l'héritage, la généricité paramétrique, le typage et les assertions. Nous soulignerons aussi les aptitudes des environnements EIFFEL pour réconcilier es bénéfices des approches statiques et dynamiques.

## 2.1 Introduction

EIFFEL existe depuis une dizaine d'années. De nombreuses discussions ont eu lieu à son sujet sur les forums d'Internet ; il a été utilisé pour écrire des milliers de composants et des applications de centaines de milliers de lignes de code. Cela permet aujourd'hui d'apprécier de manière réaliste son aptitude à atteindre ses objectifs et de mesurer la qualité de ses implémentations. Nous aurions aimé développer des perspectives d'évolution de ce langage dans les domaines de la persistance [Lahire, 1992], de la gestion de l'évolution [Brissi et Rousseau, 1995], des assertions et des requêtes [Collet, 1997], voire de la métaprogrammation : mais cela supposerait que le lecteur connaisse déjà bien EIFFEL. Il faut donc nous résoudre à présenter ici des aspects plus classiques, car bien qu'il ait marqué incontestablement les domaines de la programmation par objets et du génie logiciel, il reste assez mal connu et beaucoup d'idées fausses circulent encore à son sujet.

Cependant, vu le nombre de mécanismes en jeu, il n'est pas possible de les présenter tous en un seul chapitre, ni même d'en expliquer quelques uns en profondeur. Notre présentation se limitera donc à quelques apports marquants au génie logiciel, et en privilégiant naturellement ceux dont la problématique a été introduite dans le *chapitre 1*. Le lecteur pourra consulter avec profit le dernier ouvrage du concepteur d'EIFFEL [Meyer, 1997] qui présente de manière approfondie son point de vue sur la plupart des aspects discutés ici. Par ailleurs, une comparaison avec C++ (*chapitre 3*) et une extension pour le parallélisme (*chapitre 7*) sont donnés dans cet ouvrage-ci.

La section 2.2 précise les objectifs du langage et les compare à ceux de ses concurrents. La section 2.3 donne une vue d'ensemble sur la structure des classes, le langage de configuration LACE et les aspects syntaxiques. La section 2.4 explique l'instanciation dynamique et les deux mécanismes d'accès aux composants d'objets, par référence ou par expansion. Nous présentons les deux principales relations entre les classes EIFFEL, la clientèle (§ 2.5) et l'héritage (§ 2.6), puis la généricité paramétrique (§ 2.7). Les contrôles de type et les assertions jouent un rôle primordial pour contrôler la fiabilité des programmes : ils seront exposés dans la section 2.8. Enfin, nous avons préféré présenter le rapprochement des aptitudes statiques et dynamiques des environnements EIFFEL (§ 2.9), plutôt que les mécanismes d'exceptions et d'interfaçage, plus classiques.

## 2.2 Eiffel : un langage pour le génie logiciel

Le domaine des langages de programmation est souvent un lieu de débats passionnés qui oublient toute objectivité scientifique. Si les qualités d'un langage ont une influence, certes importante, sur celles des programmes et des coûts de mise en œuvre, elles ne sont pas suffisantes pour mesurer tout l'intérêt qu'il présente. Des outils peuvent compenser des insuffisances apparentes, si on leur donne l'information nécessaire ; les bibliothèques et la méthode de programmation associée jouent un rôle majeur pour la productivité et les qualités finales des logiciels. Un langage doit donc offrir une *solution globale à un cahier des charges de programmation*. Et il est plus utile de mesurer la pertinence de celui-ci et la cohérence des solutions proposées, que de polémiquer sur des détails ou des traits de style dont un utilisateur ne peut mesurer toutes les implications avec les autres constructions, qu'après plusieurs années de pratique. Ainsi, si C++, ADA, EIFFEL et JAVA, tous dédiés aux composants logiciels, aboutissent à des solutions de qualité différente, cela vient avant tout des différences d'objectifs et de l'état de l'art à l'époque de leur conception.

L'objectif général d'EIFFEL est de permettre la construction de grandes bibliothèques de milliers de composants et leur réutilisation pour programmer rapidement toutes sortes d'applications, avec un haut niveau de qualité : correction, robustesse, efficacité, réutilisabilité, extensibilité, transportabilité, composabilité, documentation à jour, facilité d'utilisation... C'est un objectif ambitieux qui nécessite une approche réaliste et pragmatique pour trouver les meilleurs compromis permettant d'atteindre au mieux toutes ces qualités, tout en ayant la ferme volonté d'offrir une solution simple, élégante et cohérente. Si l'idée de production massive de composants logiciels est ancienne [McIlroy, 1968], sa mise en œuvre n'est toujours pas effective aujourd'hui. Bien que des progrès significatifs aient été réalisés depuis une vingtaine d'années, certains doutent même de sa faisabilité, échaudés par des expériences malheureuses avec des langages ou des environnements inadaptés.

Le problème est difficile. Il nécessite non seulement un langage et un environnement de développement adaptés pour manipuler le logiciel, ce matériau subtil chargé d'une sémantique évolutive, mais aussi des techniques et des plates-formes pour résoudre les problèmes contingents de propriété industrielle, d'interopérabilité dans un cadre hétérogène et distribué, de persistance, de parallélisme et d'évolution. D'autre part, l'usage de composants n'est pas la seule solution au problème de la réutilisation.

Les autres alternatives sont l'écriture de progiciels génériques ou de canevas d'application (*frameworks*) [Revault, 1996] pour des domaines spécialisés : la persistance, le dialogue homme/machine, la gestion distribuée d'objets multimédias... En remarquant que le pourcentage des instructions binaires d'une application qui sont consacrées à ces sous-systèmes dépasse souvent les 80%, on constate déjà un excellent taux de réutilisation. Mais ce taux ne fait que déplacer provisoirement la frontière du volume à programmer. Dès que les programmeurs sont libérés de la programmation fastidieuse de ces sous-systèmes, ils peuvent se concentrer sur les 20% restant, qui grossissent alors rapidement pour reposer les mêmes problèmes qu'auparavant de productivité, de fiabilité... Aussi, la réutilisation par composants, comparée aux *frameworks* et aux techniques de composition hétérogène de logiciels [Nierstrasz et Tsichritzis, 1995], offre une solution plus générale, mais moins productive. Ces différentes techniques ne sont donc pas rivales, seulement complémentaires.

Pour les composants, les solutions données par les langages sont variées. Certains langages de l'ancienne génération (FORTRAN, C, MODULA-2, ADA...) font reposer la réutilisabilité sur la détermination *a priori* des variantes des composants qui pourraient être réutilisées. Les techniques de base reposent sur la modularité et sur le paramétrage, avec des formes parfois évoluées comme la généricité paramétrique (cf. *chapitre 1*). *Ces techniques sont encore d'actualité*, car elles restent utilisables chaque fois que la prévision est sûre, soit que le domaine visé est bien formalisé, soit que l'évolution est lente : calcul scientifique, logiciels pilotant du matériel lourd, etc.

Au contraire, les langages à objets offrent plus de souplesse en permettant des réutilisations *a posteriori*, soit imprévues grâce au polymorphisme, soit par adaptation manuelle grâce à l'héritage et aux techniques de métaprogrammation. Mais cette souplesse est très souvent contrebalancée par un maintien perpétuel de l'*ouverture* changements (des paramétrages ou des choix d'implémentation non arrêtés), ce qui se traduit par l'impossibilité d'effectuer les *fermetures*, c'est-à-dire de figer, au moins pour un certain temps, les valeurs effectives nécessaires aux contrôles statiques et aux optimisations.

D'autres problèmes se posent, dès que la taille des bibliothèques augmente. Il faut des outils de GLAO[1] pour trouver les composants et réaliser rapidement les adaptations, en évitant le *copié-collé*, « péché mortel » dans un monde en perpétuelle

1. Génie Logiciel Assisté par Ordinateur *(CASE, "Computer Aided Software Engineering")*.

évolution[2]. Au contraire, il faut pour chaque composant réconcilier deux points de vue, celui des clients et des héritiers[3] : c'est le principe *d'ouverture-fermeture* [Meyer, 1997]. Le point de vue des clients doit être fermé, c'est-à-dire stable, abstrait, nécessaire et suffisant, alors que celui de héritiers doit être plus ouvert aux adaptations, donc moins limité, mais plus « complice ». Dans les deux cas *il faut un cadre sémantique fort*, qui soit capable de contrôler que la réutilisation d'un composant respecte bien les contraintes du point de vue choisi, mais avec la souplesse requise par l'évolution.

Pourtant, la plupart des langages à objets n'accordent qu'une importance médiocre aux contrôles de fiabilité, qui reposent encore sur les tests et le débogage, parfois aidés par le typage statique, comme en C++, JAVA, EIFFEL, OBJECTIVE-CAML [Rémy et Vouillon, 1997], mais, sauf pour EIFFEL, jamais sur les assertions. Les contrôles de fiabilité sont pourtant indispensables à toute forme de programmation, *ex nihilo* ou par réutilisation, adaptation ou extension. Ainsi, à défaut de moyens de contrôles efficaces et souples, l'héritage sera sous-utilisé, en limitant sa largeur comme dans le cas de l'héritage simple (SMALLTALK, JAVA, ADA95...), l'usage de l'héritage multiple (CLOS, C++ [Booch et Vilot, 1996]) ou les possibilités d'adaptations (contravariance des arguments (cf. *chapitre 1*) comme dans SATHER [Omohundro et Stoutamire, 1995], absence de renommage...). De même, la plupart des langages à objets ne disposent pas de la généricité paramétrique (SMALLTALK, JAVA), se contentent d'une forme réduite (C++) ou offrent un mécanisme mal intégré à l'héritage (ADA95 [Rousseau, 1994]).

Au contraire, EIFFEL est un langage de classes bien équipé en moyens de contrôles de fiabilité, par typage fort et assertions pour la correction, et par un mécanisme d'exceptions pour la robustesse. Cela permet au programmeur de contrôler la sémantique des adaptations de composants, soit *a priori* par généricité éventuellement contrainte, soit *a posteriori* en plaçant le composant dans un graphe d'héritage multiple et profond, en osant même les redéfinitions covariantes (cf. *chapitre 1*) peu prisées dans les théories des types [Abadi et Cardelli, 1996 ; Castagna, 1997]. De plus, les contraintes spécifiques de la réutilisation par un client ou un héritier reposent sur des *contrats* bien établis, contrôlés par les assertions. Enfin, l'objectif d'efficacité n'est pas oublié et des optimiseurs éliminent les ouvertures inutilisées dans les programmes d'applications, ce qui permet de générer un code portable en C ANSI d'une efficacité voisine, voire meilleure [Collet *et al.*, 1997] que celle qu'on obtiendrait en programmant directement l'application en C.

Les aptitudes statiques d'EIFFEL (typage fort, optimisations) ne l'empêchent pas pour autant de disposer d'environnements interactifs comme EIFFELBENCH [Meyer, 1995a], qui reposent sur une compilation incrémentale proche de l'interprétation, avec un confort d'utilisation comparable à celui de SMALLTALK, pourtant interprété et non typé. Comme ce langage, EIFFEL dispose d'un ramasse miettes et

2. Rappelons le danger, s'il en est besoin : si des programmeurs (plus généralement des auteurs quelconques) construisent des textes $B_1$, $B_2$ ... $B_n$ à partir d'un copié-collé d'un texte $A$, avec ou sans modifications, toute évolution du texte original $A$ ne se répercutera pas automatiquement dans les emprunts $B_1$ ... $B_n$. Cela peut être souhaitable ou non, en tout cas délicat à contrôler s'il n'y a pas de lien explicite entre les emprunts et la version originale.

3. Par abus de langage, les mots *client* ou *héritier* d'un composant `C` désignent le programmeur d'un composant qui réutilise `C` par un lien de clientèle ou d'héritage.

implémente à sa façon un modèle « tout objet ». Cela lui permet de décrire avec des classes, non seulement les concepts d'une application, mais aussi des métaconcepts utilisés par des outils de génie logiciel, par réification[4]. L'existence de la classe `ANY`, super-classe de toutes les autres, facilite l'accès à des services d'introspection ou à des extensions au langage, par exemple pour la gestion de l'évolution, de la persistance ou des manipulations de types, comme dans les systèmes IREC [Brissi et Rousseau, 1995], FLOO [Lahire, 1992 ; Chignoli *et al.*, 1995 ; Chignoli *et al.*, 1996] ou OQUAL [Collet et Rousseau, 1996a ; Collet et Rousseau, 1996b ; Collet, 1997]. Cependant, ces aptitudes à la réification d'EIFFEL donnent beaucoup moins de possibilités que les langages ou les systèmes ouverts réflexifs (cf. *chapitre 6*) comme SMALLTALK, CLOS ou CODA [McAffer, 1995]. Cela s'explique par la difficulté à laisser n'importe quel programmeur — même un métaprogrammeur expérimenté — libre de redéfinir les mécanismes les plus vitaux du langage, patiemment élaborés par des compromis subtils au cours de six années de conception par Bertrand Meyer.

Les trois versions d'EIFFEL ont été conçues de 1985 à 1991. La version 2 [Boussard *et al.*, 1990] fut distribuée à partir de 1988 et la version 3 actuelle [Meyer, 1992a] a apporté des améliorations sensibles, notamment pour l'héritage ; de nouvelles extensions mineures sont proposées dans [Meyer, 1997]. Le langage est du domaine public, mais son évolution est contrôlée par le consortium NICE[5]. Le cahier des charges d'EIFFEL et sa méthode ont été donnés en partie dans les livres de son concepteur [Meyer, 1992a ; Meyer, 1994b ; Meyer, 1995b ; Meyer, 1997]. D'autres présentations ou de systèmes liés à EIFFEL peuvent être trouvées dans [Switzer, 1993 ; Thomas et Weedon, 1995 ; Wiener, 1995 ; Rist et Terwilliger, 1995 ; Waldén et Nerson, 1995 ; Gore, 1996 ; Jézéquel, 1996 ; Wiener, 1996 ; Gautier *et al.*, 1996].

## 2.3 Organisation générale d'un programme Eiffel

### 2.3.1 Structure d'une classe

Une classe (Fig. 2.1)[6] est organisée en sections, toutes facultatives, sauf celle d'identification, dans l'ordre suivant : indexation, identification, héritage, création, déclarations de *primitives* (attributs (variable d'instance), constantes ou routines) et invariant de classe. La section d'indexation (**indexing**) contient des informations ignorées des compilateurs, mais utilisées par des outils d'exploration (*browsing*)), d'indexation, de gestion de projet... Celle d'identification (**class**) donne le nom de la classe, ses paramètres éventuels de généricité et son statut particulier : *abstraite*

4. Du latin *res*, « la chose » et *facere*, « faire ». En approche objet, réifier, c'est décrire explicitement par un objet (donc présent s'il le faut en mémoire) un concept d'habitude implicite, dispersé ou inacessible durant l'exécution : un type, une classe, une routine, l'extension d'un type, une instruction... Au sens large, toute représentation par objets est une réification [Ferber, 1990].

5. *Nonprofit International Consortium for* EIFFEL .

6. Les premiers exemples d'EIFFEL que nous donnons illustrent de manière groupée plusieurs mécanismes expliqués dans ce chapitre. Il ne faudra pas chercher à les comprendre en entier dès leur apparition, car des explications seront données plus loin dans le texte.

```
indexing
  description: "Dates de l'ère chrétienne,%
           % julienne (<= 04/10/1582) ou grégorienne (>= 15/10/1582)"
  key_words: date, calendar, christian, julian, gregorian...
  revision: "$ Revision: 2.2 $" -- Par exemple, gestion des versions par RCS
  ...
class DATE
inherit
  COMPARABLE -- la clause d'héritage complète sera vue plus loin (Fig. 2.7, p. 56)
creation
  make, make0
feature -- Attributs
  day        : INTEGER
  month      : INTEGER
  year       : INTEGER
  ORIGIN_str : STRING is "01/01/0001" -- origine de l'ère chrétienne.
feature -- Initialiseurs
  make0 is -- Création initialisée à l'origin de l'ère chrétienne...
  make (d, m, y : INTEGER) is -- Création d'une date au jour d, mois m, année y
feature -- Changement des attributs
  update_from_string (s: STRING) is -- mise à jour avec la représentation s.
feature -- Accesseurs secondaires
  abscissa : INTEGER is -- Nombre de jours depuis origin ...
feature -- Opérations algébriques
  infix "-" (other: like Current) : INTEGER is -- Nb de jours dans Current - other
feature {NONE} -- Accès à des fonctions externes C
  time (t: INTEGER) : INTEGER is -- Accès à time(3) de la bibliothèque C Standard
invariant
  date_is_valid     : is_valid(Current)
  date_is_christian : Current >= date("01/01/0001")
  date_is_credible  : Current <= date("31/12/2200")
end -- classe DATE
```

FIG. 2.1: *Structure d'une classe* EIFFEL *: extrait de la classe* DATE.

(**deferred**, § 2.6.3) ou *expansée* (**expanded**, § 2.4). Par défaut, une classe normale n'est ni abstraite, ni expansée. La section consacrée à l'héritage (**inherit**) cite les super-classes avec des clauses éventuelles d'adaptations. La liste des *initialiseurs* (**creation**) donne le nom des procédures à utiliser pour initialiser les instanciations. Les sections de déclarations des primitives (**feature**) sont regroupées par catégories fonctionnelles (à la manière des protocoles de SMALLTALK) et précisent leur exportation (§ 2.5). L'*invariant de la classe* (**invariant**) est l'une des trois catégories d'assertions externes (§ 2.8.3).

Comme nous l'avons vu dans le *chapitre 1*, le concept de classe peut s'appréhender de deux points de vue, intensionnel ou extensionnel. Pour le génie logiciel, ces points de vues correspondent respectivement aux aspects *modulaires* et *sémantiques*.

*Du point de vue modulaire*, les classes jouent le rôle d'*unités de structuration* (seule unité pour décrire un composant), de programmation (sous la responsabilité d'un seul programmeur) et de stockage (une et une seule classe par fichier). Les primitives d'une classe se composent de constantes (simples ou structurées), d'attributs et de routines (méthodes), procédures ou fonctions. Une routine peut déclarer des variables locales de n'importe quel type, mais ne peut contenir d'autres routines ou d'autres classes. Le langage ne dispose pas de la notion de bloc, pour inciter le programmeur à localiser tous les traitements dans des routines réutilisables. La taille des corps de routines est donc le plus souvent réduite (en moyenne, cinq ou six lignes), mais le surcoût à l'exécution de cette atomisation est le plus souvent éliminé par les optimiseurs (§ 2.9).

*Du point de vue sémantique*, les classes implémentent des *types abstraits*. Chaque classe décrit un type (par exemple la classe `DATE`), ou plusieurs si elle est générique (par exemple `LIST[DATE], LIST[WINDOW]`...). Inversement, tous les types, même ceux qui sont de base (`INTEGER, BOLLEAN, STRING, ARRAY`...) sont décrits en EIFFEL dans des classes de bibliothèque, donc adaptables par héritage. Les implémentations traitent cependant les types de base de manière particulière pour l'optimisation, tant qu'ils ne sont pas redéfinis. Les procédures jouent le rôle de *modifieurs* ou d'*initialiseurs* de l'état des instances et les fonctions celui d'*accesseurs*, pour consulter leur état, sans effet de bord apparent.

### 2.3.2 Classe par opposition à système

La modélisation des procédés de développement logiciel *(software processes)* qui sous-tendent les différentes activités d'un projet logiciel (conception, spécification, implémentation, réutilisation...) nécessite d'autres concepts que les constructions syntaxiques d'un programme, en particulier dans un cadre réparti ou persistant. L'approche EIFFEL fait le minimum, mais distingue quelques entités comme les notions d'*univers*, de *classe*, de *système*, de *répertoire de classe*, etc., grâce à l'emploi de deux langages bien distincts, EIFFEL et LACE, ayant chacun sa propre syntaxe, quoique de style identique. EIFFEL est utilisé pour décrire les classes, indépendamment de tout usage, alors que LACE sert à préciser l'usage des classes pour construire un programme exécutable, appelé *système* dans la terminologie EIFFEL. Cette séparation est fondamentale pour ne pas polluer le texte des classes d'informations liées aux conditions d'exécutions, donc instables et non réutilisables, puisque classes et systèmes ont des cycles de vie très différents. Cela évite l'erreur des langages qui placent, dans le texte même des classes, des indications destinées au compilateur pour l'aider pour les optimisations : `pragma` (ADA), `register` (C), `virtual` (C++), `native` (JAVA)...

Un texte en LACE (appelé *Ace*, Assemblage de Composants EIFFEL) décrit donc un « métasystème » qui contrôle la construction et l'évolution d'un système. Un *Ace* indique la classe *racine* — dont une routine d'initialisation joue le rôle de routine principale — l'univers du système, l'armement des assertions, les options d'optimisation et de traduction dans des paquetages portables en C ANSI ou en *bytecode* JAVA, équipés d'un *Makefile*. L'*univers* d'un système précise l'organisation hiérarchique des *répertoires de classes (clusters)* qui participent à sa construction, avec d'éventuels renommages en cas de conflits. Les fichiers *Ace* jouent aussi le rôle de fichiers de configuration et dispensent de l'usage de *Makefiles* qui, de toute façon, seraient inaptes à décrire les dépendances circulaires qui peuvent exister entre les classes EIFFEL.

```
feature -- Accesseurs secondaires
  origin : DATE is
    -- Origine de l'ère chrétienne définie dans Origin_str.
    once Result := date("01/01/0001")
    ensure
      christian(Result.year) ; Result.out.is_equal(Origin_str)
    end

  year_day : INTEGER is
    -- Numéro du jour dans l'année, en commençant à 1 pour le 1er janvier.
    local m : INTEGER
    do
      if (year = 1582) and (month = 10) then
        if day <= 4 then
          Result := day
        else check day >= 15 end
          Result := day - 10
        end
      else
        Result := day
      end
      from m := 1 until m = month loop
        Result := Result + days_in_month(m, year) ; m := m+1
      end
    ensure
      greater_than_min:        Result >= 1
      less_than_year_max:      not is_leap_year(year) implies Result <= 365
      less_than_leap_year_max: is_leap_year(year) implies Result <= 366
    end -- year_day
feature -- Opérations algébriques
  infix "-" (other: like Current) : INTEGER is
    -- Nombre de jours dans Current - other
    do Result := abscissa - other.abscissa
    ensure Result = abscissa - other.abscissa
    end -- infix "-"

  infix "-y", difference_in_years (other: like Current) : REAL is
    -- Nombre d'années grégoriennes dans Current - other
    do Result := (Current - other) / 365.2425
    ensure Result = (Current - other) / 365.2425
    end -- infix "-y", difference_in_years
```

FIG. 2.2: *Exemples de routines de la classe* DATE.

### 2.3.3 Aspects syntaxiques

La syntaxe EIFFEL ressemble à celle du langage ADA, avec une soixantaine de mots réservés. Elle autorise les références en avant et les ‘;’ sont facultatifs en fin de ligne. Les commentaires débutent par des “--” jusqu'à la fin de ligne ; certains commentaires ont une position citée dans la syntaxe et jouent un rôle précis qui est exploité par des outils de documentation automatique : catégorie fonctionnelle d'une section **feature**, résumé de spécification d'une primitive en langage naturel. Des marqueurs de commentaires peuvent préciser leur intention, ou les supprimer de la vue des clients s'ils commencent par “--|”.

La structure d'une routine se compose (cf. `year_day`, Fig. 2.2 ou `creer`, Fig. 2.3, p. 46) d'un nom, d'une signature, d'une spécification résumée en commentaires, d'une précondition (**require**), d'une section de déclarations de variables locales (**local**), d'un corps de routine, d'une postcondition (**ensure**) et enfin d'une clause de rescousse (**rescue**) pour traiter les exceptions. La signature indique les paramètres de la routine et, s'il s'agit d'une fonction, le type de l'objet retourné. Le corps de routine peut être concret avec des instructions d'implémentation (**do** ou **once**), retardées (**deferred**) ou défini dans un autre langage (**external**), typiquement en C. Les routines **once** ne sont exécutées qu'une seule fois, grâce à un « drapeau secret » produit par le compilateur. Les routines retardées sont définies plus tard, dans des sous-classes. De toutes les sections d'une routine, seuls le nom, la signature et le corps sont obligatoires. Le langage ne dispose que d'un jeu minimal d'instructions pour décrire les comportements : instanciation (`!!`), appel de routine, affectation directe (`:=`) ou une tentative d'affectation (`?=`), choix logique (**if** ... **then** ... **elseif** ...**then** ... **else** ... **end**), choix discret (**inspect** ... **when** ... **then** ... **when** ... **then** ... **else** ... **end**), itération (**from** ... **invariant** ... **variant** ... **until** ... **loop** ... **end**). Le transfert de contrôle se fait toujours à la fin de chaque construction et EIFFEL ne dispose d'aucune construction comme `return, break, continue, exit`...

Pour atteindre les composantes d'un objet, EIFFEL utilise une seule *notation qualifiée* (accès navigationnel) avec l'opérateur de traversée `'.'`. Comme en programmation fonctionnelle ou par l'utilisation de tubes dans le système UNIX, une expression peut se composer avec d'autres expressions, sans limitation, car *toute expression* EIFFEL *renvoie un objet*. Celui-ci peut servir pour une affectation, comme argument d'une routine ou d'un opérateur ; mais on peut lui appliquer aussi n'importe quelle primitive exportée, de consultation ou de modification. Ce processus de composition peut se poursuivre, tant que les primitives traversées sont des consultations (variable, constante ou appel de fonction). Ainsi, avec les notations :

```
(x - 1).out.count
dupont.enfants(3).marier(jean)
```

la première expression calcule le nombre de caractères nécessaires à la représentation sous forme d'une chaîne de caractères, du nombre qui représente la valeur $x - 1$ ; la seconde notation est une instruction qui marie l'objet `jean` au troisième enfant de l'objet `dupont` (normalement une fille, ce qui sera contrôlé par les assertions).

En EIFFEL, les variables[7] qui peuvent désigner un objet sont les attributs (variable d'instance), les paramètres et les variables locales de routines. Implicitement, toute qualification d'une expression commence par le contexte courant. Celui-ci est composé des variables de la routine et de celles de sa classe, mais sans le mécanisme habituel de masquage des autres langages à structure de bloc. Ainsi, `x` et `dupont` sont des variables directement visibles dans le contexte des expressions précédentes. Les mots réservés `Current` et `Result` (cf. `year_day`, Fig. 2.2, p. 42) désignent des variables prédéfinies qui font référence respectivement à l'instance elle-même et à l'objet retourné à la sortie d'une fonction.

La définition d'une constante d'un type de base (`INTEGER, CHARACTER, REAL, DOUBLE, BOOLEAN, STRING, ARRAY`) peut utiliser une *constante littérale*, comme pour la chaîne `ORIGIN_str` (cf. Fig. 2.1, p. 40). Les tableaux littéraux sont indiqués

7. Appelées *entités* en EIFFEL ; pour faciliter la lecture, nous garderons la terminologie usuelle.

par une liste d'expressions entre guillemets (« `expr1, expr2... exprn` »). Ils servent aussi à la transmission de listes de paramètres effectifs pour les routines qui admettent un nombre variable d'arguments. Une routine peut ainsi avoir une ou plusieurs listes d'arguments dont le type peut être quelconque comme `ARRAY[ANY]` ou beaucoup plus restreint comme `ARRAY[OPTION]` ou `ARRAY[INTEGER]`. Ainsi l'exemple suivant :

```
-- définition d'une fonction avec deux listes à nombre variable d'arguments :
list_dir (options : ARRAY [ OPTION ] ; files : ARRAY [ STRING ]) is...
-- utilisation de cette fonction :
list_dir (« long, recursive, block », « "fichier1", "fichier2" » )
```

montre l'utilisation d'une commande de listage de fichiers avec une liste d'options et une liste de noms de fichiers. Ce mécanisme que nous avons suggéré à Bertrand Meyer, combine la souplesse des langages dynamiques pour le nombre des arguments avec un contrôle statique des types des arguments de chaque liste.

Pour définir une constante d'une classe quelconque, ou lorsque la définition doit être retardée jusqu'à la phase d'initialisation du programme, on utilise le mécanisme d'évaluation unique des routines **once** (cf. `Origin`, Fig. 2.2, p. 42). EIFFEL compense en partie l'absence de *variables de classe* par le mécanisme des fonctions **once** pour construire des objets instanciés une seule fois, donc uniques et partagés entre toutes les instances de la classe. Cela ne permet pas de changer d'objet partagé, mais on peut le modifier. Dans le cas d'un système persistant, la sémantique de « une seule fois » manque cependant de précision.

EIFFEL autorise la redéfinition de la plupart des opérateurs prédéfinis, préfixes (`-`, **not**...) ou infixes (`<=`, `*`, **and**...), mais pas celle des opérateurs d'égalité (`=`, $\neq$) et d'affectation (`:=`,`?=`) dont la sémantique générale est stable. L'absence de surcharge des opérateurs[8] est compensée par la possibilité d'en définir un nombre illimité, préfixes ou infixes, avec un symbole formé de n'importe quelle suite de caractères qui commence par `+`, `-`, `*`, `/`, `<`, `>`, `=`, `\`, `^`, `|`, `@`, `#` ou `&` [Meyer, 1997]. Ainsi, dans la classe `DATE` (cf. Fig. 2.2, p. 42), l'opérateur '`-`' exprime une différence en jours et l'opérateur '`-y`' une différence en années.

## 2.4 Instanciation, référence et expansion d'objets

### 2.4.1 Retour sur le « halte au tout objet »

Comme annoncé dans le *chapitre 1*, les objets peuvent servir, soit à modéliser des comportements dont l'effet est perçu par des changements d'états, soit des connaissances plus stables et moins individualisées, les valeurs. Pour percevoir le changement d'état d'un objet, il faut distinguer son *identité* de sa *valeur*. Ainsi, on peut dire d'un objet qui représente une horloge qu'il affichait « midi moins cinq » avant d'afficher « midi », mais on ne peut parler d'évolution de la valeur « midi ». Et

8. Ne pas confondre la surcharge des opérateurs avec leur redéfinition. En EIFFEL, toute classe peut définir ou redéfinir des opérateurs comme `+`, mais une seule fois. Ainsi, le type d'un objet et le nom d'une méthode ou d'un opérateur suffisent à déterminer le code à utiliser. S'il y avait surcharge comme en C++ ou ADA95, ou multisélection comme en CLOS, il faudrait en plus considérer la liste des paramètres pour effectuer cette détermination.

dès qu'on dispose d'un moyen pour distinguer « c'est le même objet » de « c'est la même valeur », on débouche aussitôt sur la possibilité de partage et de copie des objets : « deux employés ont le même chef ». Ainsi, si l'on examine les attributs d'un objet, certains seront propres à l'objet (l'âge, le nom d'une personne), sans véritable identité, proches des valeurs ; d'autres attributs au contraire seront des références à des objets externes (des enfants, un chef de service), repérables et partageables par d'autres objets. Enfin pour compliquer le tout, il se peut très bien qu'un attribut privé évolue (par exemple pour corriger un nom), ce qui permet de le considérer, à la fois comme une valeur et comme un objet évolutif, mais avec des points de vue différents.

Cette problématique est bien connue du monde des bases de données (*chapitre 5*), mais elle a été singulièrement négligée par tous les concepteurs de langages généralistes qui se sont d'abord focalisés sur des problèmes de représentation en mémoire, avant de s'intéresser à la sémantique des liens entre objets. Cela explique l'absence de constructions et de mécanismes pour exprimer et contrôler les diverses sémantiques des liens de composition [Oussalah, 1997] ou d'association [Cattell *et al.*, 1997] et l'absence de tuples de valeurs... comme le proposent en partie les langages de description des bases de données (*chapitre 5*) ou d'analyse et de conception (*chapitre 4*). EIFFEL n'échappe pas à cette règle : il ne fournit que deux mécanismes de représentation des attributs, les *références* et les *objets expansées*, dont la sémantique colle à la représentation en machine.

### 2.4.2 Références et objets expansés

Par défaut, une variable contient une *référence* à une instance de classe. Une référence diffère d'un pointeur, car elle peut être implémentée de diverses façons sans que cela intéresse le plus souvent le programmeur : par une adresse d'instance en mémoire volatile (comme suggéré dans le *chapitre 1*), une entrée dans une table d'indexation de base de données, une `URL` sur le réseau... Il n'y a pas de pointeurs en EIFFEL [9], sauf pour les interfaçages avec C ANSI. D'un point de vue sémantique, les références permettent d'implémenter des partages d'instances.

Une variable peut contenir aussi directement la valeur d'une instance, si la variable est déclarée avec le mot **expanded**. Sur l'exemple de la figure 2.3, la variable expansée `date_naissance` est stockée directement dans la représentation des instances de `PERSONNE` pour la rendre spécifique et solidaire des déplacements de ces instances. EIFFEL permet aussi de rendre automatiquement expansée toutes les instances d'une « classe expansée » dont la signature indique **expanded**. Ainsi, les types de base comme `INTEGER`, `BOOLEAN`, `DOUBLE`... sont décrits dans des classes expansées, car leurs valeurs n'ont presque jamais de signification propre. Les objets expansés sont de plus *anonymes*, c'est-à-dire sans référence pour les partager entre plusieurs instances. Mais ce sont de vrais objets, mutables (*chapitre 1*), instances de leur classe. Ils ont donc nécessairement une identité, qui peut être obtenue en associant celle de l'objet qui les contient au nom de la variable expansée, mais sans utiliser de référence.

9. C'est-à-dire pas de possibilité d'adresser une primitive.

```
class PERSONNE inherit
  ETAT_CIVIL -- accès à des primitives globales comme masculin, feminin, today...
creation creer
feature -- attributs
  nom              : expanded NAME
  prenom           : expanded NAME
  sexe             : INTEGER -- masculin ou feminin
  date_naissance   : expanded DATE
feature -- créations
  creer (n, p : STRING ; s: INTEGER ; d : STRING) is
    -- Créer une personne de nom n, prénom p, sexe s et date de naissance d.
    require
      nom_valide        : n /= void and then n.count >=2 and n.count <= 25
      prenom_valide     : p /= void and then p.count >=2 and p.count <= 20
      sexe_valide           : s = masculin or s = feminin
      date_naissance_valide : d /= void and then today.is_date_valid(d)
      pas_trop_jeune        : (today -y date(d)) >=0 -- nouveau né
      pas_trop_vieux        : (today -y date(d)) <= 122 -- Jeanne Calment
    do
      nom := n ; prenom := p ; sexe := s
      date_naissance.update_from_string(d)
    ensure
      nom = n ; prenom = p ; sexe = s
      date_naissance.is_equal(date(d))
      celibataire : mariages = Void
    end -- creer
feature -- accesseurs secondaires
  etat_matrimonial : INTEGER is
    do
      if    mariages = void             then Result := celibataire
      elseif mariages.max.date_fin = void then Result := marie
                                         else Result := divorce
      end
    ensure
      Result = celibataire or Result = marie or Result = divorce
      definition: (Result = celibataire) = (mariages = void)
    end -- etat_matrimonial
feature {NONE}
  mariages       : ORDERED_SET [ MARIAGE ]
    -- Ensemble des mariages, par ordre chronologique, Void si célibataire
feature {MARIAGE} -- Seule la classe MARIAGE  a accès aux primitives de cette section
  noter_nouveau_mariage (m: MARIAGE) is ...
...
```

FIG. 2.3: *Extrait de la classe* PERSONNE *(vue concrète).*

### 2.4.3 Instanciation

L'utilisation d'une variable exige la présence d'un objet obtenu par une instanciation ou une affectation préalable, sinon cela déclenche une exception. En EIFFEL, l'*instanciation* est uniquement dynamique, soit explicite pour les variables de référence, soit implicite pour les variables expansées, en même temps que l'instance qui les contient. La notation `!T!v` crée une instance du type `T` et l'affecte à la variable de référence `v` ; par défaut `!!v` crée une instance du type indiqué dans la déclaration de `v`.

L'instanciation initialise les champs de l'objet à la valeur neutre de leur type, mais peut être combinée à une initialisation explicite, en choisissant l'une des routines de la section **creation**. Ainsi, pour créer une date (cf. Fig. 2.1, p. 40), on peut écrire `!!d.make0` ou `!!d.make(14, 7, 1996)`. La notation `!!d` est syntaxiquement correcte, mais violerait ici l'invariant de l'instance créée, en donnant par défaut la valeur `0` (neutre pour le type `INTEGER`) aux champs `day`, `month`, `year`, soit la date `"0/0/0000"` qui est invalide.

L'instanciation automatique des variables expansées nécessite que le compilateur puisse choisir seul, et sans ambiguïté, le mécanisme d'initialisation à effectuer. C'est le cas si la classe est initialisée par défaut ou s'il n'y a qu'une routine d'initialisation sans argument, comme dans l'exemple de la classe `DATE` (cf. Fig. 2.1, p. 40), grâce à son unique initialiseur `date0`.

### 2.4.4 Propriétés essentielles et contigentes d'un objet

Tout objet EIFFEL a deux sortes de propriétés, un *type*, et un *statut* qui sont repérés par des champs distincts dans son en-tête. Le *type* concerne des propriétés *essentielles*, d'ordre sémantique [10]. Le *statut* caractérise des propriétés *contingentes*, instables, comme le fait d'être expansé, persistant, mobile sur le réseau, marqué par un ramasse miettes, verrouillé par une transaction...

Ainsi, l'instruction d'affectation (`:=`) est contrôlée statiquement par des critères de compatibilité sémantiques et contingente, en considérant la faisabilité technique comme la taille des objets, la disponibilité... Dans le cas de variables référencées, il s'agit de l'*attachement d'un objet* à la variable ; le polymorphisme ne pose ici aucune difficulté de taille des variables réceptrices, puisque toutes les références ont la même taille. Pour les variables expansées, il s'agit d'une *recopie de valeur*, champ par champ, qui n'est techniquement possible que si la taille des variables est compatible : le polymorphisme est interdit dans ce cas [11]. Dans le cas de l'affectation d'un objet expansé à une variable référencée, il y a *clonage* (création d'une instance avec recopie de la valeur) et attachement de l'instance créé. Dans le cas inverse, il y a recopie de la valeur de l'objet référencé dans la variable expansée.

### 2.4.5 Discussion sur le mécanisme d'expansion

Le mécanisme d'expansion est donc de bas niveau et a été conçu au départ pour introduire des champs de taille quelconque dans les objets, par exemple pour les nombres réels en double précision. Il ne faut pas l'utiliser sur des critères d'efficacité qui sont très hypothétiques. En effet, sauf pour les types de base, l'usage des variables expansées ne fait gagner souvent, ni temps d'accès, ni place en mémoire. À notre avis, la vraie justification de l'expansion devrait être beaucoup plus sémantique, sur des critères de partage ou non de valeurs. Cela laisserait l'implémentation libre

10. Dans les implémentations actuelles, la référence au type d'une instance est un simple pointeur à la réification de sa classe, comme indiqué dans le modèle simplifié présenté au *chapitre 1*. Dans le cas d'EIFFEL, c'est une erreur, car cela ne permet pas de manipuler correctement les types génériques.

11. C'est une règle trop pessimiste. On pourrait déterminer statiquement si la taille des instances des sous-types d'une variable expansée permet l'affectation.

de choisir, entre expanser ou référencer, pour implémenter de manière efficace le partage ou la solidarité et supprimerait l'actuelle restriction sur l'affectation polymorphe entre variables expansées.

## 2.5 Relation de clientèle

La relation de *clientèle* exprime une relation *externe* entre une classe *fournisseur* de primitives, les services, et les classes qui les utilisent, ses *clients*. Bien qu'elle ne soit pas explicitée par un mot-clef, cette relation est facilement reconnue comme telle par les programmeurs ou par les outils. Elle apparaît dès qu'une classe cite une autre classe en dehors de la clause d'héritage, pour déclarer le type d'une variable ou dériver un type effectif d'une classe générique. D'un point de vue plus abstrait, la relation de clientèle est souvent utilisée pour exprimer un lien d'agrégation entre instances, explicité dans les méthodes de conception (cf. *chapitre 4*). Le graphe de la relation de clientèle est quelconque, *avec circuits*. Une classe peut être cliente de n'importe quelle classe, y compris d'elle-même, de ses super-classes ou de ses sous-classes. L'aspect externe de la relation de clientèle implique une *sélection* et une *abstraction* des services offerts, un *contrat de responsabilité* entre les programmeurs et un contrôle des *droits d'accès*.

La *sélection* des services se fait par un mécanisme d'exportation qui regroupe les primitives en trois catégories qui sont précisées par des accolades après le mot **feature** : visibles à tous les clients (par défaut ou en précisant {**all**}), cachées à tous les clients (en indiquant {`NONE`}) et enfin visibles, par une exportation sélective qui cite les clients « amis » sélectionnés. Ainsi, sur la figure 2.3, les attributs `nom`, `prenom`... sont publics, la liste `mariages` est privée et la primitive `noter_nouveau_mariage` n'est visible qu'au client `MARIAGE`. Une instance a accès sans restriction à ses propres primitives, qu'elles soient définies dans sa classe ou héritées de super-classes. Mais pour l'accès aux primitives d'une autre instance, même si celle-ci est de la même classe, les règles de clientèle s'appliquent : une classe peut ainsi être amenée à s'exporter à elle-même certaines primitives. Par défaut, une classe $H$ qui hérite d'une classe $P$ exporte les primitives de $P$ de la même manière que $P$. Cette exportation peut être adaptée de manière plus restreinte ou plus large dans la clause d'héritage de $H$ (§ 2.6.5), selon le *point de vue* de ses propres clients.

L'*abstraction* des services affine *la vue* produite par la sélection en la simplifiant. Il ne s'agit pas seulement de voir ou non une primitive, il faut n'en voir que la partie *nécessaire et suffisante* à son utilisation externe. Cela est bénéfique pour la maîtrise de l'évolution (les parties cachées peuvent évoluer dans les fournisseurs sans qu'il soit nécessaire de réactualiser les clients), pour faciliter la réutilisation (les clients n'ont pas besoin d'étudier les détails d'implémentation de leurs fournisseurs) et pour la fiabilité (les changements d'état effectués par des clients ne peuvent se faire que par des routines exportées). En EIFFEL, les vues abstraites des classes produites pour les clients ne sont pas construites à la main dans un texte d'interface explicite, redondant et séparé du code, comme en ADA, C++, JAVA. Ces vues sont construites automatiquement par des outils d'abstraction (par exemple la commande *short*) à partir du code autodocumenté. Pour chaque primitive exportée, la vue abstraite ne mentionne que le nom, la signature, le commentaire de spécification et la partie des assertions **require** et **ensure** qui est strictement nécessaire aux clients. Nous verrons à la section 2.8.3, le principe de sélection des assertions et leur application sur l'exemple de la classe `PERSONNE`.

En ce qui concerne les *droits d'accès*, les routines qui appartiennent à la vue d'un client sont utilisables par celui-ci, sans restriction. Au contraire, les attributs exportés ou les paramètres de routine ne peuvent qu'être lus, non écrits directement par des affectations de leurs clients. Cela ne les empêche pas de modifier l'état de ces variables, mais en passant par des modifieurs exportés de leur classe, comme le montre l'exemple suivant d'utilisation de la classe `PERSONNE` (cf. Fig. 2.3, p. 46) :

```
p: PERSONNE ...
p.nom := "Durand" -- illegal
p.nom.put(6,'t') -- modifie le 6e caractère du nom : légal
```

Dans le cas où un contrôle plus strict est exigé, cela peut être obtenu en rendant privé les attributs sensibles, et en exportant des routines qui réalisent le contrôle recherché :

```
class PERSONNE ...
feature {NONE} -- attributs privés
  i_nom: NAME ...
feature -- fonctions d'accès aux attributs privés
  nom : NAME is do Result := i_nom end ...
```

Cette solution générale permet d'exprimer le contrôle exact qui est souhaité, sans entraîner de perte d'efficacité, grâce aux optimiseurs [12]. Elle nécessite cependant une écriture laborieuse qui peut être éliminée par une génération automatique d'un atelier de GLAO , d'après une sélection interactive des accès nécessaires.

## 2.6 Relation d'héritage

En EIFFEL, l'*héritage* peut être simple, multiple ou répété. L'héritage exprime une relation d'usage *interne*, de « complicité » entre les super-classes et les sous-classes. La clause d'héritage cite la liste de toutes les *parentes* [13], avec d'éventuelles clauses d'adaptations, parfois complexes en EIFFEL, et toujours sous le contrôle du programmeur. La cohérence des choix de celui-ci est en grande partie contrôlée par le compilateur et les assertions. L'héritage est répété de manière explicite si la clause d'héritage cite plusieurs fois la même parente ou implicite si, par transitivité, plusieurs parentes ont la même super-classe (héritage dit en « losange »).

Contrairement au graphe de clientèle, celui d'héritage est sans circuit. Chaque classe hérite implicitement de `ANY` et a `NONE` comme sous-classe. La classe `ANY` dispose des caractéristiques utiles à toutes les classes, ce qui permet d'étendre facilement les possibilités du langage EIFFEL en y introduisant de nouveaux services. La classe `NONE` a une sémantique absurde [14], mais ses instances sont spéciales, et tout

12. Ce type d'optimisation n'est évidemment pas spécifique à EIFFEL et peut être réalisé dans toute implémentation de langage, y compris dans des langages dynamiques comme SMALLTALK.

13. Dans la terminologie EIFFEL, on appelle *parentes* les super-classes directes qui sont indiquées dans une clause d'héritage.

14. Puisqu'elle hérite de toutes les classes, par exemple `RECTANGLE` et `TRIANGLE` ; les contradictions seraient révélées par l'invariant de `NONE` comme nous le verrons plus loin (§ 2.8.3).

appel de l'une de ses primitives déclenche une exception. La valeur `Void`, qui sert à initialiser par défaut toute variable de référence, désigne en fait une référence à une instance de `NONE`. Ainsi, toute utilisation d'une telle variable non attachée à un objet normal, par affectation ou instanciation explicite, déclenche une exception. De même l'exportation d'une primitive à la classe `NONE` (§ 2.5) n'est pas utilisable par celle-ci, ce qui indique une primitive privée.

L'aspect interne et « complice » de l'héritage est reflété par les règles d'accès aux primitives héritées. Une instance a accès à toutes les primitives héritées dans sa classe, avec les mêmes droits que ses super-classes, car l'exportation ne joue que pour les clients. Ces primitives peuvent cependant être renommées, redéfinies, voire fusionnées (§ 2.6.5) avec d'autres primitives héritées ou propres à la classe. L'abstraction des primitives héritées est moins forte que dans la vue des clients et montre tout ce qui n'est pas interne aux corps des routines. Ainsi, la totalité des assertions externes sont visibles aux héritiers et participent aux *contrats d'héritage* (§ 2.8.3).

Les deux points de vue, modulaire et sémantique d'une classe (§ 2.3.1), se retrouvent dans l'utilisation de l'héritage pour des objectifs variés. L'héritage pour la *factorisation* (§ 2.6.1) est aussi appelé héritage *d'implémentation* ou *d'inclusion* ; l'héritage pour la *compatibilité de substitution* (§ 2.6.2) est également appelé héritage *d'interface*, de *sous-typage* ou de *spécialisation*. On trouve aussi des combinaisons de ces objectifs avec une compatibilité partielle dans des « mariages de raison » (§ 2.6.4).

### 2.6.1 Point de vue modulaire de l'héritage : inclusion

La *factorisation* utilise l'héritage comme un moyen de récupération automatique des primitives des super-classes. Cela est proche du mécanisme d'*inclusion* dans nombre de langages (C, `makefile`, LaTeX...). C'est une technique utile pour éviter la duplication, non seulement pour gagner de la place, mais surtout pour éviter la réactualisation des copies en cas d'évolution des versions originales (problème du *copié-collé*). L'héritage d'inclusion ignore les contraintes sémantiques, ce qui l'empêche d'être exploité pour le polymorphisme. On peut donc se débarrasser des primitives inutiles en supprimant leur exportation ou en redéfinir sans se soucier de contraintes de compatibilité. Cela correspond à la technique utilisée dans les définitions différentielles de fichiers de description (`termcap`, `printcap`, `termio`...) du système UNIX, mais aussi à un usage abusif de certains programmeurs [15]. Ainsi, la classe `ETAT_CIVIL` qui regroupe des constantes comme `masculin...` et donne accès à la date du jour peut être utilisée par un héritage d'inclusion dans diverses classes comme `PERSONNE` (cf. Fig. 2.3, p. 46).

Le programmeur n'a pas à se soucier de l'encombrement des primitives héritées non utilisées, car celles-ci peuvent être éliminées par les optimiseurs, en phase de finalisation (§ 2.9). Il doit en revanche résoudre d'éventuels conflits de noms par des clauses de renommage, mais peut être aidé par un environnement interactif.

15. Comme de faire hériter deux fois la classe `ELLIPSE` de la classe `CERCLE`, pour récupérer deux diamètres pour la largeur et la hauteur, et deux centres pour les foyers...

### 2.6.2 Point de vue sémantique de l'héritage : sous-typage

La *compatibilité de substitution* garantit la cohérence des *affectations polymorphes*. Une sous-classe doit ici être *conforme à toutes ses super-classes* : *tout ce que celles-ci savent faire, la sous-classe doit savoir le faire aussi !* Mais peut-être en s'y prenant autrement, voire avec des aptitudes supplémentaires. C'est l'idée de *spécialisation* ou d'établissement d'un lien « *is a* » entre les classes (dans le sens où un rectangle « est un » polygone). Une sous-classe définit donc un *sous-type*[16] (ou plusieurs, si elle est générique, § 2.3.1, p. 39) de tous les types définis par ses super-classes, éventuellement paramétrés en cas de généricité. Ainsi, en héritant de la classe `PERSONNE`, la classe `MANAGER` définit le type *manager* qui est un sous-type de *personne*, défini par la classe `PERSONNE` ; si `LINKED_LIST[E]` hérite de `LIST[E]`, le type *linked_list[integer]* est un sous-type de *list[integer]*, lui-même un sous-type *list[numeric]... list[any]*, puisque la classe `INTEGER` hérite de `NUMERIC`... `ANY`.

Une variable acquiert son type de manière statique, dans sa déclaration ; une instance obtient le sien de manière dynamique, lors de sa création. C'est le fait qu'une même variable puisse attacher successivement des objets de types différents qui justifie le terme *d'affectation polymorphe*. Mais contrairement aux langages non typés, cette souplesse est acquise ici avec beaucoup plus de sécurité, car tous les objets susceptibles d'être attachés à une variable sont conformes à son type (§ 2.8.2). Ainsi, une variable déclarée avec le type `POLYGONE` pourra attacher des instances de `TRAPEZE`, `TRIANGLE`, `RECTANGLE`..., car ces classes héritent toutes de `POLYGONE`. De même, une instance de la classe `CARRE` pourra être attachée à une variable de type `RECTANGLE`, `LOSANGE`, ou `PARALLELOGRAMME`, si l'on suppose que la classe `CARRE` hérite des classes `RECTANGLE` et `LOSANGE` (héritage multiple) et que celles-ci héritent de la classe `PARALLELOGRAMME`.

Il faut aussi noter le *point de vue* qu'exprime le type d'une variable et le contrat implicite qu'il induit entre le programmeur et le compilateur. Le programmeur s'engage à n'utiliser les variables qu'avec le point de vue déclaré ; le compilateur, après avoir vérifié que le programmeur respectait ce point de vue, s'engage alors à fournir les primitives demandées durant l'exécution. En cas d'héritage multiple et d'affectations polymorphes, cela permet d'utiliser le même objet selon les points de vue de ses super-classes : un carré pourra être manipulé comme un rectangle, un losange, un trapèze...

Supposons maintenant que la variable `p : POLYGONE` référence une instance de la classe `CARRE` et que `perimetre : REAL` soit une primitive de `POLYGONE`. Si les contrôles de conformité du typage statique fonctionnent bien, il n'y a aucun risque que l'exécution de la notation `p.perimetre`, qui est légale puisque `p` est *vu comme un polygone*, ne trouve pas une implémentation conforme de `perimetre` pour l'objet qui est attaché à `p` et qui représente un carré. Cette primitive sera même sûrement redéfinie pour le carré, pour des raisons d'efficacité en multipliant par quatre la longueur d'un côté. Ainsi, le polymorphisme empêche le compilateur de déterminer statiquement l'adresse des routines utilisées, mais pas de contrôler que leur usage est conforme aux intentions déclarées dans les types des variables. C'est ce qu'on

16. Comme nous l'avons vu dans le *chapitre 1*, l'idée de sous-typage repose sur le point de vue extensionnel des classes.

appelle la *liaison dynamique* (ou *tardive*) et cela correspond aussi à la métaphore de *l'envoi de message* (cf. chapitre 1). Ainsi, le typage statique garantit l'existence des primitives souhaitées, alors que la liaison dynamique se charge de les trouver durant l'exécution.

Contrairement à une idée répandue, la liaison dynamique ralentit très peu l'exécution des programmes [Ducournau, 1997 ; Collet *et al.*, 1997]. Dans un langage typé statiquement comme EIFFEL, il est possible de construire des tables d'indexation optimisées qui réduisent le temps d'accès à un temps constant réduit, indépendant de la largeur et de la profondeur de l'héritage. D'autres accélérations peuvent aussi s'ajouter, comme l'utilisation de caches matériels, l'élimination des liaisons dynamiques inutiles, l'expansion de code... Enfin, et c'est sans doute le plus important, les liaisons dynamiques qui demeurent expriment en fait des discriminations qui appartiennent à la logique même du problème.

Programmées dans un langage sans héritage, elles le seraient par des énoncés à choix multiple. Ainsi, plutôt qu'écrire `p.perimetre`, un programmeur en langage C écrirait probablement une sélection explicite des différents cas avec une instruction `switch`, ce qui aurait une efficacité comparable à l'emploi des tables d'indexation. Mais le risque d'erreurs serait manifeste, car cette discrimination devrait être adaptée pour les dizaines de primitives de figures (centre, diagonale, aire, affichage, translation, rotation...). De plus, il faudrait connaître par avance tous les types de figures. Enfin, il faudrait réviser toutes les primitives des figures, en cas d'ajout ou de retrait de l'une d'entre elles. Au contraire, la programmation par objets permet d'écrire des notations stables comme `p.perimetre`, sans avoir à connaître par avance toutes les figures, car la discrimination est automatiquement réalisée à l'exécution.

### 2.6.3 Classes abstraites

Comme les autres langages à objets, EIFFEL permet de définir des *classes abstraites* dont certaines primitives seront implémentées dans des sous-classes par un héritage de sous-typage. Une classe abstraite n'est pas nécessairement totalement abstraite, car elle peut déclarer des primitives abstraites (**deferred**) et concrètes. De plus, lorsqu'une classe implémente une primitive abstraite, elle n'est pas tenue d'implémenter toutes les autres. Ce mécanisme souple permet de définir les primitives, à l'endroit exact qui convient dans le graphe d'héritage : ni trop tôt, ni trop tard. Ainsi, la classe `COMPARABLE` (Fig. 2.4) qui définit un ordre total n'a pas besoin de laisser abstraits tous ses opérateurs, puisque l'un d'eux peut servir à définir les autres. On factorise ainsi mieux les implémentations qui peuvent l'être et l'on donne moins de travail aux implémenteurs des sous-classes. Si l'on souhaite améliorer l'efficacité algorithmique de ces primitives dans les sous-classes, il suffit de les redéfinir lors des héritages ultérieurs.

Pour empêcher statiquement les clients d'activer des primitives non implémentées, EIFFEL choisit comme les autres langages la solution de facilité, en interdisant la création d'instances de classes abstraites [17]. Une variable de classe abstraite

17. C'est un peu pessimiste. On pourrait, mais en compliquant le contrôle statique, — pour un bénéfice il est vrai assez rare — interdire l'activation des primitives qui ne sont pas implémentées ou qui utilisent des primitives non implémentées.

```
deferred class COMPARABLE
inherit
  PART_COMPARABLE
    redefine
      infix "<", infix "<=", infix ">", infix ">=", is_equal
    end
feature -- Comparison
  infix "<" (other: like Current): BOOLEAN is
      -- Is current object less than other ?
    deferred
    ensure then
      asymmetric: Result implies not (other < Current)
    end

  infix ">" (other: like Current): BOOLEAN is
      -- Is current object greater than other?
    do
      Result := other < Current
    ensure then
      definition: Result = (other < Current)
    end

  is_equal (other: like Current): BOOLEAN is
      -- Is other attached to an object of the same type
      -- as current object and identical to it?
    do
      Result := (not (Current < other) and not (other < Current))
    ensure then
      trichotomy: Result = (not (Current < other) and not (other < Current))
    end

  ... -- idem pour infix "<=", infix ">=", is_equal

invariant
  irreflexive_comparison: not (Current < Current)
end -- class COMPARABLE
```

FIG. 2.4: *Exemple de classe abstraite :* `COMPARABLE`.

ne peut donc attacher des instances concrètes qu'en utilisant des affectations polymorphes. Pour faciliter le repérage des classes abstraites, on doit utiliser le mot **deferred** dans leur signature. Les classes abstraites servent donc à organiser des hiérarchies d'héritage en regroupant des propriétés générales selon des critères de factorisation (par exemple `LIST, COLLECTION, SET`...) ou pour exprimer des contraintes de généricité (par exemple `COMPARABLE, NUMERIC, INDEXABLE`...), comme nous le verrons à la section 2.7.

L'exemple de la classe `COMPARABLE` montre aussi le rôle majeur des assertions pour aider à la compréhension d'une abstraction et contrôler que l'implémentation ultérieure satisfait l'intention de l'auteur d'une classe abstraite (contrat d'héritage). C'est par exemple le cas de la seule primitive abstraite `"<"`, qui est utilisée pour implémenter et définir les autres primitives, et dont la propriété d'antisymétrie est propagée aux autres primitives, par définition. La classe `DATE` (cf. Fig. 2.1, p. 40) hérite ainsi de `COMPARABLE` pour exprimer qu'elle satisfait cette propriété d'ordre total.

```
a  : ARRAY [ INTEGER ]
as : ARRAYED_STACK [ INTEGER ]
s  : STACK [ INTEGER ]
...
!!as         -- création d'un objet d'adresse #7ba de type arrayed_stack[integer]
as.push(5)   -- empile la valeur 5 comme premier élément de l'objet #7ba
as.put(3,1)  -- erreur détectée statiquement : put n'est pas exportée
a := as      -- affectation polymorphe : a référence l'objet #7ba
a.put(3,1)   -- tentative de modification du 1er élément du tableau,
             -- ici le sommet de pile => erreur détectée statiquement
s := as      -- affectation polymorphe
s.push(7)    -- utilisation polymorphe normale => pas d'erreur
```

FIG. 2.5: *Utilisation erronée, par polymorphisme, d'une primitive non exportée.*

### 2.6.4 Héritage à compatibilité partielle

EIFFEL permet aussi d'utiliser l'héritage avec un objectif de compatibilité partielle. Dans ce cas, les primitives héritées qui perturbent la sémantique de la sous-classe sont « mises en sommeil », en ne les montrant pas aux clients, par modification de leur exportation.

C'est le cas par exemple d'une classe `ARRAYED_STACK` qui implémente une pile avec un tableau, en héritant à la fois d'une classe `STACK` (point de vue sémantique) et `ARRAY` pour s'implémenter (point de vue modulaire). Dans ce cas que Bertrand Meyer appelle un « mariage de raison », certaines des primitives de `ARRAY`, comme `put(element, index)` qui permettrait de modifier n'importe quel élément de la pile, sont contradictoires avec la sémantique d'une pile. Le programmeur peut choisir la sémantique dominante (ici celle de pile) en n'exportant pas cette primitive aux clients. Toute utilisation comme client d'une instance de `ARRAYED_STACK` avec le point de vue, soit de de `ARRAYED_STACK`, soit de `STACK` (suite à une affectation polymorphe) fonctionne ainsi correctement.

Mais que se passerait-il (Fig. 2.5), si le programmeur voulait utiliser l'instance `as` de `ARRAYED_STACK` avec le point de vue de la classe `ARRAY` (variable `a`) ? La primitive `put`, bien que non exportée dans la classe `ARRAYED_STACK`, existe cependant et est exportée dans `ARRAY`. Elle fonctionne par ailleurs parfaitement avec le point de vue d'un tableau, si l'on se limite à cet usage. Ce qui pose problème, c'est que le même objet ne puisse évidemment pas fonctionner selon différents points de vue contradictoires, *dans la même application*. En effet, l'usage de `put` avec le point de vue `a` d'un tableau permet *a priori* de modifier un élément quelconque de l'objet `ARRAYED_STACK`. Cela n'est erroné que si on l'utilise ensuite comme une pile. Diverses techniques sont possibles pour détecter ce genre d'erreurs, soit par des assertions, soit par des contrôles de types statiques, comme nous le verrons à la section 2.8.2. D'autres langages comme JAVA optent pour la solution d'interdire l'héritage multiple d'implémentation. C'est plus facile à contrôler, mais cela réduit la souplesse des réutilisations.

```
inherit
  A
    rename
      ancien_nom1 as nom1, ancien_nom2 as nom2...
    export
      { all } prim_publique1, prim_publique2...
      { NONE } prim_privée1, prim_privée2...
      { F1, F2 } prim_exportée_sélective1, prim_exportée_sélective2...
    undefine
      prim_deferred1, prim_deferred2...
    redefine
      prim_redef1, prim_redef2...
    select
      prim_sélectionnée1, prim_sélectionnée2...
    end
  B
    idem
  C
    idem
  ...
```

FIG. 2.6: *Structure générale d'une section d'héritage en* EIFFEL.

### 2.6.5 Mécanismes d'adaptation par héritage

L'héritage multiple est un mécanisme puissant qui permet de factoriser les primitives et de les réutiliser selon une multitude de points de vue, ce qui facilite la réutilisation des composants et les possibilités de paramétrages génériques. Pourtant, des adversaires de l'héritage multiple pensent qu'il serait « problématique ».

Pour le programmeur, il n'y a pas de difficulté intellectuelle particulière à comprendre qu'une chose puisse être constituée de plusieurs aspects, qu'on peut souhaiter considérer globalement ou séparément. Ainsi, un carré est à la fois un rectangle et un losange ; une fenêtre d'un système graphique est une figure rectangulaire, un objet graphique et un arbre de fenêtres... Des études comme [Ouaggag et Godin, 1997] montrent aussi que l'héritage multiple apporte moins de désordre (entropie) conceptuel que l'héritage simple.

Cependant, le programmeur est sollicité pour exprimer explicitement dans les clauses d'héritage ce qu'il souhaite exactement. Celles-ci ont la structure générale indiquée sur la figure 2.6. Dans les héritages sans adaptation, seul le nom de la classe parente est indiqué. Dans le cas contraire, différentes clauses, toutes facultatives, permettent d'exprimer les choix du programmeur, classe parente par classe parente.

La clause **rename** sert à adapter dans la sous-classe les noms des primitives parentes, soit pour résoudre un conflit de nom [Ducournau *et al.*, 1995], soit pour adapter une terminologie au point de vue local. Ainsi, dans le cas de la classe `DATE` (Fig. 2.7), les opérateurs `"<"` et `">"`, hérités de la classe `COMPARABLE` pour exprimer une relation d'ordre, sont renommés ici pour mettre en évidence leur sémantique particulière d'antériorité : `"<before"` `">after"` sont plus parlants pour des dates que `"<"` et `">"`.

```
class DATE
inherit
  ANY
    redefine
      out
    end
  COMPARABLE
    rename
      infix "<" as "<before", infix ">" as ">after"...
    undefine
      out, is_equal
    end
```

FIG. 2.7: *Clause d'adaptation par héritage de la classe* DATE.

La clause **export** a une syntaxe voisine de celle utilisée dans les sections **feature**. Elle permet d'ajuster l'exportation des primitives héritées au point de vue de la sous-classe, comme dans l'exemple de la primitive put de la classe ARRAYED_STACK.

La clause **redefine** cite les primitives concrètes déjà définies dans les classes parentes qui sont redéfinies dans la sous-classe. C'est le cas de la primitive perimetre de la classe CARRE ou des opérateurs de comparaison de la classe COMPARABLE (cf. Fig. 2.4, p. 53).

Aucune construction syntaxique n'est utilisée pour indiquer qu'une primitive abstraite héritée est *définie* dans une sous-classe : il suffit que celle-ci dispose d'une implémentation de la primitive. Et pour l'obtenir, comme cela est nécessaire aussi aux redéfinitions, on dispose de deux moyens : soit on définit une nouvelle primitive dans la sous-classe, soit on utilise une primitive concrète héritée de l'une des super-classes.

Cela permet aussi d'implémenter « d'un seul coup » plusieurs primitives issues de super-classes différentes, en réalisant une véritable *fusion sémantique* de primitives. Dans certains cas, cette fusion est obtenue en utilisant une clause **undefine** qui permet de rendre temporairement abstraite les primitives concrètes héritées que l'on souhaite fusionner. La fusion s'opère chaque fois que dans un ensemble de primitives héritées ou définies localement, de même nom, de même signature et de même sémantique, une et une seule primitive est concrète. Cette règle est contrôlée d'abord statiquement, puis de manière plus fine par les assertions pour la compatibilité sémantique.

### 2.6.6 Héritage répété

L'héritage répété est un cas particulier d'héritage multiple où l'on hérite plusieurs fois, explicitement ou implicitement, de la même classe. Par exemple, la classe CARRE hérite deux fois de la classe PARALLELOGRAMME, par ses parentes LOSANGE et RECTANGLE. Toutes les sous-classes héritent ainsi au moins deux fois, et souvent un grand nombre de fois, de la classe ANY. Les règles d'EIFFEL pour l'héritage répété sont particulièrement simples et naturelles. Si l'on souhaite dupliquer une primitive héritée, avec la même implémentation ou une implémentation différente, il est nécessaire de lui donner deux noms distincts. De même, il est interdit (contrôle statique) de disposer du même nom pour deux implémentations distinctes. Le renommage, contraint ou voulu, suffit à déterminer les partages ou les duplications. Ainsi la classe `CARRE` n'hérite qu'une seule fois des deux diagonales de la classe `PARALLELOGRAMME`, puisqu'elles gardent le même nom de `diagonale1` et `diagonale2` et la même implémentation dans ses parentes. Les propriétés de ces diagonales, qui sont exprimées par des invariants dans les classes `PARALLELOGRAMME` (même milieu), `RECTANGLE` (égales) et `LOSANGE` (médiatrices l'une de l'autre) se cumulent dans le carré, mais sans contradiction.

En cas d'héritage répété, direct ou indirect, la clause **select** permet de sélectionner les variantes de primitives qui seront utilisées en cas d'utilisation polymorphique, avec le point de vue des super-classes communes.

### 2.6.7 Exemples d'adaptations et de fusions par héritage

Illustrons l'emploi de quelques possibilités d'adaptation avec l'exemple de la classe `DATE` (cf. Fig. 2.7). La primitive `out` qui est définie dans la classe `ANY` rend une chaîne qui représente la valeur d'un objet d'un type quelconque, en listant les noms et les valeurs de ses différents attributs. Dans le cas d'une date, il est logique d'adapter cette représentation selon un format plus concis comme `"17/08/1997"`. Pour redéfinir cette primitive, on utilise un héritage de `ANY` (d'habitude implicite) avec une clause de redéfinition. Mais la primitive `out` existe aussi dans la classe `COMPARABLE`, puisque celle-ci hérite de `ANY`. Il y a donc conflit, car nous avons maintenant deux implémentations différentes de `out`, celle d'origine à travers la classe `COMPARABLE` ($out_o$) et celle que nous voulons redéfinir dans la classe `DATE` ($out_n$).

Pour résoudre ce conflit on pourrait renommer la variante $out_o$ en `COMPARABLE_out`. Mais cela encombrerait la classe `DATE` d'une primitive inutile, même si les optimiseurs nous en débarrasseraient en phase de finalisation. De même, nous avons écarté la solution qui redéfinit `out`$_o$ par $out_n$, car lors d'une affectation polymorphe, $out_n$ ne serait utilisée qu'avec le point de vue des sous-classes de `COMPARABLE` et non avec celui plus général de ses super-classe, comme la classe `ANY`. Nous avons donc préféré réaliser la fusion de $out_o$ et $out_n$, en rendant $out_o$ abstraite par la clause **undefine**.

D'autre part, la classe `ANY` dispose d'un prédicat `is_equal` qui teste l'égalité de valeur de deux objets quelconques, en comparant leurs différents champs. Cet prédicat d'égalité doit évidemment être cohérent avec les opérateurs de comparaison de la classe `COMPARABLE`, ce qui a été effectué en redéfinissant `is_equal` dans cette classe (cf. Fig. 2.4, p. 53). Dans le cas de la classe `DATE`, l'héritage explicite de `ANY` nécessité pour traiter la fusion des variantes de `out` a réintroduit la variante d'origine de `is_equal` dans la classe `DATE`. Celle-ci dispose donc maintenant de deux implémentations différentes de ce prédicat, bien qu'elles aient même nom, même signature et des sémantiques parfaitement équivalentes : comparaison de la totalité des champs d'une date. Il faut donc opérer à nouveau une fusion ; cette fois, nous avons gardé la version d'origine de `ANY`, non pour des raisons sémantiques, mais d'efficacité.

### 2.6.8 Discussion sur l'utilisation de l'héritage en Eiffel

Comme on le voit sur cet exemple simple, une adaptation d'héritage sémantiquement cohérente requiert toute l'attention du programmeur. Celui-ci est heureusement aidé par les contrôles statiques du compilateur et par les assertions qui peuvent révéler d'autres erreurs moins évidentes. Certaines clauses d'adaptation peuvent également être complexes et occuper plus d'une page de listing. Cette complexité, qui peut surprendre à première vue, nous paraît cependant incontournable. Elle permet en effet de faire jouer à la clause d'héritage le rôle d'un véritable langage de *composition de logiciel par fusion*, qui complète les autres techniques d'assemblage de composants comme les « langages de glu » [Nierstrasz et Tsichritzis, 1995]. Enfin, pour des arbitrages fastidieux comme des renommages en série, un bon environnement interactif peut faire le travail de manière semi-automatique.

Le fait qu'on utilise en EIFFEL l'héritage pour toutes sortes d'utilisations potentiellement contradictoires se démarque aussi nettement de l'approche suivie dans les autres langages. CLOS et SMALLTALK laissent le programmeur totalement libre, sans contrôle de cohérence ; JAVA, ADA95 et C++ séparent syntaxiquement les interfaces et les implémentations des classes et proposent des mécanismes différents pour l'héritage d'implémentation et d'interface. Cela facilite les contrôles, mais réduit les possibilités de réutilisation.

Le point de vue de Bertrand Meyer [Meyer, 1997] est qu'il existe une grande variété de cas d'emploi défendables de l'héritage, et qu'il faut laisser le programmeur libre de choisir, à condition de lui donner les moyens de contrôler qu'il fait bien ce qu'il veut. C'est parce qu'EIFFEL dispose de mécanismes de contrôle, par typage statique et par assertions, qu'on peut utiliser la même construction pour ces objectifs contradictoires. Mais comme aucun langage ne saurait empêcher d'écrire des constructions insensées, un programmeur EIFFEL peut bien sûr utiliser l'héritage de manière abusive : par exemple pour l'agrégation, en faisant hériter la classe `AVION` une fois de la classe `FUSELAGE`, deux fois de la classe `AILE`... Normalement, le bon sens devrait suffire pour éviter ce genre de confusion évidente...

## 2.7 Généricité paramétrique

Comme nous l'avons dit dans la section 2.2, les techniques fondées sur le paramétrage et la généricité ont été développées dans les années 70 pour programmer les évolutions possibles d'un composant. Comparées aux techniques à base d'héritage, cette programmation *a priori* présente l'avantage d'une automatisation de l'adaptation, mais a l'inconvénient de nécessiter de prévoir par avance les aspects qui peuvent varier, avec le risque d'en faire trop ou pas assez. Cependant, chaque fois que la prévision est sûre, notamment dans les paramétrages évidents ou pour des structures bien étudiées ou stables, les techniques génériques gardent tout leur intérêt. Bertrand Meyer a montré la dualité de ces deux techniques [Meyer, 1986], qui ne sont pas rivales mais complémentaires. Cependant, EIFFEL est paradoxalement encore le seul langage à objets à offrir une solution quasi complète à la généricité, qui s'intègre bien avec l'héritage. Lorsqu'une classe est paramétrée par

```
class ARRAY [ ELEMENT ]
  e1, e2: ELEMENT
...
  if e1 = e2 then ...
  e1 := e2...
  print(e1.out)...
end -- class ARRAY
```

FIG. 2.8: *Exemple de classe générique sans contrainte :* ARRAY.

un ou plusieurs types, elle indique ses paramètres formels (le plus souvent un seul) entre crochets.

Dans le cas de la *généricité simple*, aucune hypothèse particulière n'est émise sur ces paramètres. C'est le cas de nombreuses classes qui décrivent des structures de données comme les tableaux, listes, ensembles, arbres... Ainsi, sur l'exemple de la classe `ARRAY` (Fig. 2.8), `ELEMENT` indique un paramètre formel qui peut être remplacé par un type quelconque lors d'une utilisation : `ARRAY [INTEGER]`, `ARRAY [PERSONNE]`, `ARRAY [LIST[PERSONNE]]`... À l'intérieur de la classe `ARRAY`, seules les primitives qui sont disponibles dans la classe `ANY`, super-classe de toutes les classes, sont autorisées. On pourra donc comparer l'égalité de deux éléments `e1` et `e2`, attacher l'objet référencé par `e2` dans la variable `e1`, imprimer l'objet `e1`, sans même savoir le type exact de ces objets.

Si d'autres propriétés que celles de la classe `ANY` sont requises sur un type formel $T$, on exprime cette contrainte en précisant dans la clause de généricité un super-type $P$ de $T$ qui dispose au minimum de ces propriétés. Ainsi, sur l'exemple d'un ensemble ordonné (Fig. 2.9), on peut maintenant, en plus des cas d'utilisation cités sur la figure 2.8, comparer deux éléments avec l'opérateur `"<"` de la classe `COMPARABLE` déjà étudiée. Pour choisir la contrainte de généricité *nécessaire et suffisante*, il faut parcourir le graphe d'héritage de haut en bas ; dès qu'on a assez de fonctionnalités, il faut s'arrêter pour ne pas exiger des propriétés inutiles. Si l'on ne trouve pas la classe cherchée, le plus souvent abstraite, il faut en écrire une et la placer dans le graphe d'héritage. Ce travail n'est cependant pas inutile, puisqu'il pourra être réutilisé dans d'autres paramétrages génériques et les classes abstraites ainsi produites ont souvent des fonctionnalités fondamentales réutilisables.

L'utilisation d'une classe générique, avec ou sans contrainte, est très simple : il suffit de substituer aux paramètres formels des expressions de types effectifs compatibles. Ces expressions peuvent s'utiliser partout où l'on utilise des types, dans les déclarations de primitives, de variables locales, de paramètres de signatures ou dans les clauses d'héritage.

```
class ORDERED_SET [ ELEMENT -> COMPARABLE ]
  e1, e2: ELEMENT
...
  if e1 < e2 then ...
end -- class ORDERED_SET
```

FIG. 2.9: *Exemple de classe générique avec contrainte :* ORDERED_SET.

Dans la classe PERSONNE (cf. Fig. 2.3, p. 46), l'utilisation d'un ensemble ordonné de mariages est possible pour décrire les différents mariages d'une personne, en ordre croissant de dates (attribut `mariages`), car l'hypothèse d'un ordre total est assurée par l'héritage de la classe COMPARABLE (cf. Fig. 2.4, p. 53) dans la classe MARIAGE (non donnée dans ce texte). En revanche, un ensemble ordonné de personnes serait illégal (détection statique), car la classe PERSONNE n'hérite pas de COMPARABLE. Un programmeur qui voudrait absolument réaliser des ensembles ordonnés de personnes, ne pourrait pas se contenter de faire hériter la classe PERSONNE de la classe COMPARABLE. Il lui faudrait aussi implémenter l'opérateur "<" de comparaison de personnes (différé dans la classe COMPARABLE) et respecter la contrainte d'intégrité exprimée dans l'invariant de la classe ORDERED_SET. Cette contrainte repose inévitablement sur les propriétés de l'opérateur "<" qui confèrent à la classe COMPARABLE et à ses sous-classes les propriétés algébriques d'un ordre total. Celles-ci devraient normalement être explicitées dans cette classe par des assertions vérifiables [18]. De plus, ce programmeur devrait aussi redéfinir le prédicat `is_equal`, comme nous l'avons fait précédemment (§ 2.6.5).

Sans la généricité simple ou contrainte, il faudrait redéfinir d'innombrables classes comme INTEGER_ARRAY, REAL_ARRAY, MARRIAGE_ORDERED_SET... ou renoncer aux bénéfices du typage statique. L'apport de la généricité à la lisibilité est également manifeste. Comme les compilateurs disposent de toute l'information voulue, ils peuvent effectuer des contrôles statiques pour la compatibilité des paramètres effectifs et pour l'utilisation des paramètres formels par la classe générique. Pour substituer les paramètres effectifs aux paramètres formels, les implémentations actuelles d'EIFFEL utilisent la liaison dynamique. Cela a l'avantage de la simplicité, mais manque de précision : les types ne sont pas réifiés avec leurs paramètres et il n'est pas possible d'instancier une variable dont le type est formel. Ainsi, si l'on veut construire l'élément neutre de l'anneau des matrices (matrice remplie d'éléments neutres), on ne peut le faire en EIFFEL [Boussard *et al.*, 1990], comme cela serait possible en ADA83 avec le mécanisme d'instanciation générique [Ada, 1983 ; Le Verrand, 1982].

## 2.8 Aides à la fiabilité

### 2.8.1 Nécessité de techniques complémentaires

La fiabilité exprime la conformité du fonctionnement d'un programme à ce qui est souhaité par un utilisateur, ou par un programmeur en cas de réutilisation de composants. On distingue traditionnellement deux composantes de la fiabilité, la correction et la robustesse. La *correction* (ou *validité*) suppose que l'environnement du programme soit lui-même fiable ; dans ce cas, un programme est correct s'il satisfait sa spécification. La définition de la *robustesse* est moins précise et peut atteindre des niveaux variables : un programme est robuste, s'il fait « tout son possible »

18. Le langage actuel d'assertions d'EIFFEL est insuffisant pour exprimer de tels axiomes ; en revanche, le langage OQUAL, qui étend les assertions d'EIFFEL avec des quantifications [Collet, 1997], peut les exprimer et les tester.

lorsqu'une défaillance est constatée dans son environnement... Ces brefs rappels de génie logiciel montrent combien l'idée de contrat est naturelle pour aborder la fiabilité. D'une part, on voit qu'un programmeur ne peut être tenu pour responsable d'une erreur, que s'il dispose d'une définition précise de ce qui lui est demandé. D'autre part, la correction ne peut être atteinte que si des hypothèses de fiabilité de l'environnement du programme sont satisfaites ; lorsqu'elles ne le sont pas, il faut détecter les défaillances ou les attaques et agir comme il convient pour satisfaire la robustesse.

Depuis trente ans, de nombreux travaux ont porté sur la fiabilité. Les *approches théoriques* visent à la correction démontrée par des preuves, à partir d'une *spécification formelle* et totale du fonctionnement du logiciel. Cela est indispensable pour les logiciels critiques et utile pour les composants stables, souvent réutilisés, comme ceux situés en haut d'une hiérarchie d'héritage. Bien que la preuve puisse parfois être automatisée, les spécifications formelles sont coûteuses à élaborer, même avec l'assistance d'environnements interactifs. Elles sont de plus dérisoires pour les logiciels en perpétuelle évolution, et pour certains programmes visuels.

Le *typage statique* et de manière générale tous les contrôles statiques révèlent avant toute exécution de nombreuses erreurs, avec un effort de spécification très limité : dans le cas de la programmation par objets, la définition statique d'un type se réduit le plus souvent à placer une classe dans un graphe d'héritage. En s'appuyant sur ce graphe, un compilateur peut alors déterminer les compatibilités d'assemblage de primitives, de variables ou de valeurs littérales et vérifier l'existence *a priori* des primitives appelées à partir du type statique des « objets receveurs des messages ». Mais l'insuffisance de la spécification d'un type ne permet évidemment pas de révéler toutes les erreurs. Rien n'empêche, du point de vue du typage, de faire hériter la classe `MACHINE-À-LAVER` de la classe `CAMION`, même s'il faut être bien fatigué pour en arriver là. Et avoir la certitude qu'une routine existe, même avec la contrainte d'une signature conforme, ne garantit évidemment pas que l'effet obtenu sera celui souhaité : la mise en marche d'une machine à laver pourrait très bien défoncer le mur de la buanderie...

D'autre part, la correction théorique acquise par une preuve ou un contrôle de type *ne dispense jamais des tests*, car toute preuve repose sur des hypothèses que la réalité peut contredire ; inversement, les tests, même très complets, ne garantissent jamais l'absence d'erreurs, car ils ne peuvent atteindre l'exhaustivité pour d'évidentes raisons de combinatoire. Enfin, même si l'on parvient péniblement à obtenir un programme correct, en supposant que son environnement respecte certaines hypothèses, on n'a fait que la moitié du chemin. La vraie fiabilité doit en effet toujours être pessimiste et admettre la possibilité de violation des hypothèses : erreurs des logiciels utilisés, erreurs humaines plus ou moins volontaires, défaillance des machines, des supports de stockage et des lignes de transmission...

Aucune technique n'est donc suffisante pour approcher la fiabilité avec un haut niveau de confiance ; *le bon sens pratique recommande de les utiliser toutes*, de manière complémentaire. C'est ce que propose le langage EIFFEL avec son système de typage (pourtant covariant), d'assertions et d'exceptions. Par manque de place, nous ne présenterons pas les exceptions d'EIFFEL, qui s'intègrent très bien avec les autres constructions du langage, en particulier les assertions.

### 2.8.2 Typage statique et « catcall »

Le typage statique consiste à déterminer avant toute exécution si une primitive appliquée à une variable existe pour le type de cette variable, même en cas de polymorphisme. Il faut aussi s'assurer, dans le cas où cette primitive est une routine, que ses arguments sont acceptables pour cette routine. Avec l'exemple canonique des figures géométriques :

```
p: POLYGONE
r: RECTANGLE
...
p:= r
x:= p.perimetre -- point de vue des polygones : sommation de tous les côtés
x:= r.perimetre -- point de vue des rectangles : algorithme simplifié
```

le compilateur peut vérifier qu'il existe bien une primitive `perimetre` pour l'objet attaché à la variable `p`, même si celui-ci est un trapèze ou un rectangle (comme c'est le cas ici), puisque toutes ces figures héritent directement ou non de la classe `FIGURE_FERMÉE` qui dispose de `perimetre`. Comme le fonctionnement de l'héritage garantit la propagation des primitives des super-classes dans leurs sous-classes, le contrôle semble simple à réaliser.

Dans le cas d'EIFFEL cependant, celui-ci est délicat à faire dans deux cas qui sont appelés par Bertrand Meyer *catcall*[19] :

- si, une classe modifie le statut d'exportation des primitives héritées et si l'on utilise l'une de ses instances de manière polymorphe, cela peut conduire à une incohérence, comme nous l'avons vu avec la primitive `put` dans l'exemple de la pile implémentée par héritage de la classe `ARRAY` (cf. Fig. 2.5, p. 54).
- à cause du choix des redéfinitions covariantes (*chapitre 1*), on risque d'avoir des variables polymorphes incohérentes.

Illustrons ce dernier cas avec la classe `ORDERED_SET[E → COMPARABLE]`. Celle-ci suppose que ses éléments sont d'un type `E`, équipé des opérations de comparaison. Dans la classe `ORDERED_SET`, une routine d'ajout d'un élément aura par exemple la signature `put (e: E)`. Cette signature peut être redéfinie de manière covariante en `put (e: INTEGER)` pour un ensemble ordonné d'entiers, puisque les entiers, les chaînes, les dates... héritent de la classe `COMPARABLE`. Ainsi un ensemble ordonné d'entiers est normalement assuré de ne contenir que des entiers ou des sous-types des entiers, ce qui permet de comparer ses éléments de manière cohérente. Mais que se passerait-il si l'on utilisait une affectation polymorphe du genre :

```
osc: ORDERED_SET [ COMPARABLE ]
osi: ORDERED_SET [ INTEGER ]
osi.put(5) -- l
osc:= osi -- l
osc.put("toto") -- catcall erroné : on ajoute une chaîne à osi
```

on pourrait à première vue insérer une chaîne (ou une date) dans un ensemble ordonné d'entiers, ce qui poserait évidemment un problème pour l'addition ou la comparaison de ses éléments...

19. CAT est l'acronyme de *Changing Avaibility or Type*.

Pendant longtemps, les implémentations d'EIFFEL étaient incapables de détecter statiquement les deux types d'erreurs de *catcall*. Plusieurs tentatives ont été effectuées avant d'arriver à la solution qui suit, proposée par Meyer en 1996. Le lecteur trouvera au chapitre 17 de [Meyer, 1997] une discussion complète sur les aspects du typage en EIFFEL et sur les différents travaux liés. De manière simplifiée, la règle du *catcall* s'articule en trois points :

- elle définit un *catcall* comme un appel à une primitive dont le statut d'exportation (hérité de ses parentes) a changé, ou si l'un de ses arguments a été redéfini de manière covariante.
- elle définit un appel polymorphe comme l'appel d'une primitive du type d'une variable qui, potentiellement (cela est déterminé statiquement), attache un objet de l'un de ses sous-types par une affectation polymorphe.
- elle interdit les *catcalls* polymorphes.

Sur l'exemple précédent des ensembles ordonnés d'entiers, tout appel à la routine `put` (redéfinition covariante) est un *catcall*. La variable `osc` est polymorphe (`osc := osi`), mais pas `osi`. Ainsi, l'appel `osi.put(5)` est légal (*catcall* non polymorphe), alors que `osc.put("toto")` est illégal (*catcall* polymorphe). Sur l'exemple des piles `ARRAYED_STACK` (cf. Fig. 2.5, p. 54), tout appel à la routine `put` est un *catcall* (changement d'exportation). Les variables `s` (`s := as`) et `a` (`a := as`) sont polymorphes, donc les appels `a.put(3,1)`, `s.push(7)` le sont aussi. Le premier est illégal (*catcall* polymorphe), mais pas le second (appel polymorphe, mais non *catcall*).

Ce mécanisme de contrôle est simple et a le mérite de pouvoir fonctionner de manière incrémentale, ce qui est utile pour des environnements interactifs comme EIFFELBENCH. Il est cependant un peu pessimiste et asymétrique : il interdit d'utiliser des objets de type `ARRAYED_STACK` dans une application qui ne les utiliserait que comme `ARRAY`, alors que cela n'aurait aucun inconvénient (quoique de peu d'intérêt pour cet exemple). De plus, le contrôle des *catcalls* devrait se faire sur tous les programmes d'une même application, notamment en cas de persistance. Tant qu'à en arriver là, une solution qui exhiberait le *point de vue* souhaité pour utiliser des composants de bibliothèque, par une construction syntaxique explicite (par exemple par des *modules* comme dans OBJECTIVE-CAML), nous semblerait préférable.

### 2.8.3 Assertions

**Présentation du mécanisme**

Les assertions du langage EIFFEL sont des expressions booléennes qui portent sur les variables d'un programme, avec la possibilité d'appeler n'importe quelle fonction ou attribut, mais sans quantification. Elles expriment qu'une propriété (qui ne porte que sur une petite partie de l'état d'un programme) doit être vraie à un moment précis de son exécution. L'évaluation des assertions serait trop coûteuse en phase opérationnelle ; elles ne sont donc *armées normalement qu'en phase de mise au point*, par des indications données dans le fichier *Ace*. Celui-ci sélectionne des ensembles de classes de même niveau de vérification, ce qui permet de faire des évaluations plus ou moins complètes, selon ce niveau.

On distingue actuellement cinq sortes de clauses d'assertions [20], selon l'endroit où elles sont situées dans la classe. L'invariant de classe (**invariant**), les préconditions (**require**) ou les postconditions (**ensure**) de routines correspondent aux *assertions externes*, visibles en partie par les clients ou totalement par les héritiers. Les *assertions internes* aux routines sont les clauses **check** (cf. routine `year_day`, Fig. 2.2, p. 42) qui vérifient qu'une étape importante d'un algorithme est bien atteinte, et les invariants d'itération qui testent si une propriété reste vraie, pour tous les pas d'une itération.

Chaque clause d'assertions se présente comme une suite de propositions évaluées dans leur ordre d'apparition dans le texte. Elles peuvent être étiquetées pour faciliter leur repérage dans le texte de la classe et pour comprendre l'intention de la proposition (par exemple la précondition de la routine `creer`, Fig. 2.3, p. 46). Par défaut, elles sont numérotées dans chaque clause.

Pour exprimer les changements d'états dans les postconditions, EIFFEL offre deux opérateurs prédéfinis. **old** placé devant une expression demande son évaluation à l'entrée de la routine. Ainsi, l'assertion `count =` **old** `count +1` indique un effet d'incrémentation. L'expression **strip**`(a,b...)` donne accès à la liste des champs d'un objet (sous la forme d'un `ARRAY[ANY]`), sauf ceux mentionnés. Ainsi, **strip**`()` donne tous les champs d'un objet et **strip**`(a) =` **old** **strip**`(a)` indique que la routine qui contient cette assertion ne change au plus que la valeur de la variable `a`.

Lorsqu'une assertion armée échoue, cela déclenche une exception qui peut être traitée dans une clause de rescousse (**rescue**). Si une classe hérite de la classe `EXCEPTION` de la bibliothèque, on dispose de toute l'information voulue pour connaître le lieu exact du problème, sa signification et le programmeur responsable. Si les exceptions levées par les assertions ne sont pas récupérées, elles sont propagées en parcourant les routines actives de la pile d'exécution, jusqu'à trouver une routine équipée d'une section de rescousse. Cela se produit au plus tard dans la routine de lancement de l'exécutif EIFFEL dont le traitement par défaut consiste à afficher un message précis et complet sur la voie standard d'erreur, avec l'identité des objets et des routines impliquées. On peut aussi activer directement un débogueur interactif et explorer l'état des objets fautifs. Bien que ce mécanisme permette une forme de programmation défensive, il faut absolument éviter toute confusion dans l'emploi des assertions et des exceptions. Les assertions sont faites *pour déceler des erreurs de programmation* et doivent disparaître lorsque le logiciel est dans sa phase opérationnelle. Au contraire, les traitements d'exceptions demeurent en phase opérationnelle et permettent *d'améliorer la robustesse*, en particulier pour détecter les erreurs des utilisateurs.

### Utilisation des assertions

Les assertions ont trois utilisations principales :

- l'obtention automatique d'une documentation *testable*, en accord avec la réalité du code et *avec différents points de vue* ;

20. Le langage OQUAL [Collet, 1997] propose une classification plus fine avec dix sortes d'assertions.

- l'amélioration de la fiabilité des programmes,
- l'écriture explicite des droits et des devoirs de chaque programmeur : les *contrats de clientèle et d'héritage*.

Pour la documentation automatique, le choix délibéré de Bertrand Meyer est malheureusement de ne fournir qu'un langage d'assertions simpliste (sans quantification) pour garder des temps d'évaluation acceptables. Ce manque d'expressivité est certainement à l'origine du scepticisme de ceux qui doutent de l'utilité de ce mécanisme. Mais si l'on développe en EIFFEL, on est surpris par l'aptitude des assertions à déceler un grand nombre d'erreurs. En effet, la réutilisation intense des classes, qui sont en général assez bien équipées en assertions[21], explique qu'une erreur de programmation a toutes les chances de trahir une propriété qui porte sur l'état des classes réutilisées, en particulier celles de structures de données. Ainsi, les assertions permettent d'équiper les composants de moyens de défense et de contrôle qui favorisent leur autonomie pour les différentes réutilisations. Elles jouent aussi un rôle important dans les programmes de tests, puisqu'elles réalisent des diagnostics précis des échecs éventuels. L'utilisation des outils de débogage est ainsi rare, réservée aux cas pathologiques inexpliqués par les assertions.

Les assertions peuvent aussi être utilisées par différents outils pour comprendre la sémantique d'une classe de manière beaucoup plus fine que ne le permettent les types, mais sans pouvoir rivaliser avec des spécifications formelles. Ainsi, dans le projet IREC [Brissi et Rousseau, 1995], il est souvent possible de comprendre le sens d'une modification de classe et de réactualiser automatiquement les classes qui en dépendent. La possibilité de placer des assertions dès les premières étapes de construction d'un programme est exploitée par certaines méthodes de conception « sans couture », avec des possibilités de traces ou de rétroconception, comme dans la méthode BON [Waldén et Nerson, 1995] qui est utilisée dans le système d'aide à la conception EIFFELCASE. Mais l'utilisation sans doute la mieux connue des assertions est l'énoncé des responsabilités des programmeurs par des contrats explicites [Meyer, 1991].

### Contrats de clientèle

Un *contrat* établit de manière explicite les droits et les devoirs de chaque partie. Dans le cas d'un développement de logiciel, ce contrat peut être juridique et explicité dans un cahier des charges, ou moral entre les programmeurs. Les assertions permettent d'établir des contrats pour les deux types de relations entre classes, de clientèle et d'héritage.

Pour la relation de clientèle, le contrat est fixé par les pré et les postconditions de chaque routine exportée. Le client doit respecter la précondition et peut compter sur la postcondition. Inversement, le fournisseur n'est tenu de respecter la postcondition que si la précondition est vraie. Ainsi précisés les droits et les devoirs de chacun, on peut déterminer qui est responsable d'une erreur, et éviter des tests redondants de

21. Des statistiques faites sur la bibliothèque ISE (622 classes) et sur le projet K2 (800 classes) [Lahire et Jugant, 1995] donnent respectivement 41% et 23% de routines équipées d'une précondition, 24% et 20% de routines avec une postcondition.

l'appelant et de l'appelé, qui aboutissent aussi souvent à l'absence de test, chacun pensant que c'est à l'autre de le faire. Pour que le contrat ne soit pas *léonin*, il faut que le client ait les moyens de le satisfaire. Il est donc illégal en EIFFEL de faire apparaître un attribut ou une fonction non exportés dans une précondition. Pour la postcondition en revanche, il est permis d'exprimer des propriétés contingentes, qui ne servent qu'à tester un comportement connu seulement de l'implémentation. Pour ne pas obscurcir la vue abstraite des clients, les constructeurs d'abstraction comme l'outil *short* retirent automatiquement toutes les assertions qui mentionnent une primitive cachée aux clients.

En reprenant l'exemple de la classe `PERSONNE` (cf. Fig. 2.3, p. 46), l'outil *short* fournit la vue abstraite dont un extrait est donné sur la figure 2.10. Seules les primitives exportées à tous les clients comme `nom`, `creer`, `etat-matrimonial`... sont montrées, avec leur signature, leur commentaire, leur précondition intégrale et la partie utile de leur postcondition. Ainsi, la deuxième assertion (`definition`) de la fonction `etat_matrimonial` est omise, puisqu'elle mentionne l'attribut `mariage` qui est privé. Il en va de même pour la dernière assertion (`celibataire:`) de la routine `creer`, pour la même raison. Dans ce cas cependant, il s'agit d'une erreur de formulation du programmeur qui aurait dû utiliser l'assertion `etat_matrimonial = celibataire` qui serait alors apparue dans la vue abstraite, puisque cette fonction et les valeurs possibles sont exportées aux clients. On voit donc qu'en EIFFEL, il faut non seulement tester le comportement des routines, mais aussi la *cohérence des vues* produites automatiquement : c'est typiquement le rôle d'un chef de projet. On voit aussi sur cet exemple que les clients de la classe `PERSONNE` doivent s'assurer que les arguments fournis pour créer une personne doivent satisfaire toutes sortes de conditions de validité. Même si le programme client obtient ces données de l'utilisateur, il *doit les vérifier* avant de les transmettre à la routine de création. La postcondition joue un rôle moins essentiel pour les tests, mais est utile pour la documentation. Ainsi, en supposant corrigée l'erreur signalée ci-dessus, le client de la classe `PERSONNE` est informé que la personne créée est au départ célibataire. Cela lui indique donc qu'il devra ultérieurement ajouter les différents mariages de cette personne s'ils existent. Les assertions exhibent donc les règles d'emploi des classes, et fournissent des ébauches de scénarios d'utilisation.

Si l'on considère maintenant les primitives de la classe `DATE` (cf. Fig. 2.2, p. 42), l'on observe que les postconditions des opérateurs `"-"` et `"-y"` sont des définitions complètes, alors que celle de la fonction `"year_day"` ne fait qu'encadrer le résultat dans le but d'un test. Dans le premier cas, la spécification est totalement équivalente au code. Elle est nécessaire cependant pour la documentation et les contrôles qui doivent par nature être redondants. Souvent, la spécification est plus synthétique et moins efficace que le code, ce qui justifie mieux la différence. Mais même si les deux descriptions sont au même niveau d'abstraction, la duplication reste nécessaire pour vérifier que les redéfinitions éventuelles du code respectent bien l'intention de l'auteur initial. C'est ce qu'on appelle les contrats d'héritage.

```
class interface PERSONNE
creation creer
feature -- attributs
  nom            : expanded NAME
feature -- créations
  creer (n, p : STRING ; s: INTEGER ; d : STRING)
    -- Créer une personne de nom n, prénom p, sexe s et date de naissance d.
    require
      nom_valide       : n /= void and then n.count >=2 and n.count <= 25
      prenom_valide    : p /= void and then p.count >=2 and p.count <= 20
      sexe_valide           : s = masculin or s = feminin
      date_naissance_valide : d /= void and then today.is_date_valid(d)
      pas_trop_jeune        : (today -y date(d)) >=0 -- nouveau né
      pas_trop_vieux        : (today -y date(d)) <= 122 -- Jeanne Calment
    ensure
      nom    = n
      prenom = p
      sexe   = s
      date_naissance.is_equal(date(d))
feature -- accesseurs secondaires
  etat_matrimonial : INTEGER
    ensure
      Result = celibataire or Result = marie or Result = divorce
end -- class PERSONNE
```

FIG. 2.10: *Extrait de la classe* `PERSONNE` *(vue abstraite des clients).*

### Contrats d'héritage

Les contrats d'héritage s'assurent, d'une part que la sémantique générale d'une classe ne sera pas dénaturée lors des héritages ultérieurs *(droit des auteurs)*, et d'autre part que le fonctionnement général est correct *(droit des emprunteurs)*. Ce fonctionnement est d'abord contrôlé par les postconditions qui expriment de manière plus ou moins précise les effets des primitives de modifications *(les modifieurs)* et les définitions des fonctions de consultation *(les accesseurs)*. Pour contrôler que l'enchaînement des primitives respecte bien un scénario minimal, les techniques de spécifications algébriques utilisent des axiomes qui combinent les accesseurs et les modifieurs. L'équivalent simplifié est obtenu en EIFFEL avec les *invariants de classes* qui expriment de manière synthétique l'effet des différentes *primitives exportées*, puisqu'ils doivent être maintenus vrais après chacune de leurs exécutions. Les invariants de classe jouent aussi un rôle important pour contrôler *les cohérences d'ensemble*, aussi bien temporelles (cohérence de l'évolution) que spatiales (contraintes d'intégrité de bases de données).

Ainsi, en reprenant la classe `DATE` (cf. Fig. 2.1, p. 40), on voit que l'invariant exprime que toute date doit être valide, quelle que soit son évolution. Dans le calendrier chrétien, toutes les combinaisons de jours, mois et années ne sont pas valides et il faut tenir compte du rattrapage opéré lors de l'adoption du calendrier grégorien. Ainsi, l'auteur de la routine `make` de création d'une date doit satisfaire à la sortie l'invariant de validité, ce qui suppose qu'on lui fournisse des arguments compatibles avec une date valide, ce qui sera exprimé dans la précondition. Des outils d'aide statiques pourraient aider les programmeurs en faisant ce genre de déduction et établir ainsi, semi-automatiquement, certaines préconditions. De même, on pourrait envisager d'évaluer statiquement certaines assertions, comme cela commence à se faire pour les contraintes d'intégrité de bases de données [Benzaken et Schaefer, 1997].

L'invariant de classe joue aussi un rôle important pour les contrats d'héritage, *puisque toutes les classes héritent des invariants de leurs super-classes.* Dans le cas d'héritage d'inclusion, les super-classes n'ont généralement pas d'invariant (la classe `ETAT_CIVIL` par exemple) ou des invariants indépendants, qui ne peuvent donc être contradictoires. Dans le cas d'héritage de sous-typage au contraire, les propriétés exprimées par les invariants peuvent capturer des propriétés fondamentales, presque taxonomiques, quelquechose qui s'apparente à un « code génétique ». Ainsi, les erreurs d'héritages, comme faire hériter la classe `ARAIGNEE` (huit pattes) de la classe `INSECTE` (six pattes), est détectée par la conjonction contradictoire sur le nombre de pattes. Des exemples plus réalistes et beaucoup plus complexes ne manquent pas et, sans ce mécanisme de contrôle, il serait extrêmement périlleux de construire des hiérarchies d'héritages larges et profondes comme celles qui sont usuelles en EIFFEL. Ainsi, en reprenant les échantillons des classes de la bibliothèque ISE et du projet K2 on obtient, pour le nombre moyen de classes parentes explicitement héritées, les valeurs respectives de 1.38 et de 2.46, avec des maximums de 7 et 13. La profondeur maximum des classes (nombre maximum de parentes jusqu'à la classe `ANY`) est en moyenne de 4.0 (ISE) et de 4.9 (K2), avec des maximums de 11.

En cas de redéfinition d'une primitive, la règle générale est de garantir la substitution polymorphe du nouveau comportement avec le point de vue des super-classes. Il y a donc un lien logique entre les assertions de la version d'origine et celles de la version redéfinie. Depuis la version 3 d'EIFFEL, le mécanisme de contrôle est réalisé automatiquement, en élargissant les préconditions des routines redéfinies par l'emploi imposé du mot réservé **require else**. De même les postconditions de ces routines sont resserrées par l'usage de **ensure then** (cf. Fig. 2.4, p. 53).

Le lecteur trouvera dans [Collet, 1997] une discussion plus complète sur les emplois des assertions et quelques possibilités d'extensions, notamment en introduisant des quantifications et en amorçant un rapprochement avec les requêtes de bases de données.

## 2.9 Les environnements Eiffel

Depuis la première implémentation d'EIFFEL2 par la société ISE en 1988, les environnements se sont considérablement améliorés et diversifiés. Il existe actuellement quatre implémentations principales d'EIFFEL3 qui sont développées par les sociétés ISE (*Interactive Software Engineering*, `http://www.eiffel.com`), OT (*Object Tools*, ex *SIG Computer*, `http://www.sigco.com`), Tower (*Tower Technology Co*, `http://www.twr.com`) et l'implémentation SMALL EIFFEL (*LORIA, Dominique Colnet*, `ftp.loria.fr/pub/loria/genielog/SmallEiffel`) qui est distribuée gratuitement selon la convention des produits *GNU* de la *Free Software Foundation* (F.S.F.).

### 2.9.1 Objectifs généraux

Ces environnements visent tous plus ou moins les mêmes objectifs d'interactivité, mais avec la possibilité de finaliser la fin d'un projet en générant un code optimisé dans les standards actuels (C ANSI, C++ ou *bytecode* JAVA ). La documentation de toutes les classes est testée selon le point de vue des clients ou des héritiers, grâce à des outils comme *short* (§ 2.8.3), *flat* ou *flat-short*. L'outil *flat* donne la vue concrète et complète de toutes les primitives d'une classe, en rapatriant toutes celles héritées ; l'outil *flat-short* applique à cette vue un filtrage qui donne la vue abstraite de toutes les primitives disponible dans une classe donnée.

L'efficacité du code généré peut être voisine, voire même dépasser celle d'un programme écrit à la main en langage C [Collet *et al.*, 1997]. En effet, les optimisations de la phase de finalisation suppriment les primitives héritées non utilisées, les liaisons dynamiques inutiles et expansent le code des petites routines lorsqu'il n'y a, ni récursivité, ni liaison dynamique. Il est donc inutile, comme le font encore certains langages, de demander l'aide du programmeur pour ces optimisations. Si cet avis est encore utile, comme de choisir entre une optimisation de la vitesse d'exécution ou de l'espace mémoire, sa place n'est pas dans le texte des classes. Celui-ci doit en effet être le plus possible indépendant du contexte d'utilisation, pour faciliter sa réutilisation. Dans le cas d'EIFFEL, toutes les informations nécessaires à la génération d'une application sont placées dans un fichier *Ace* (§ 2.3.2). De plus, la pertinence des choix du programmeur ne peut rivaliser avec celle d'un outil automatique qui, seul, peut avoir toute la connaissance nécessaire, fiable et à jour au moment utile pour l'exploiter.

La transportabilité est également un soucis marqué des environnements EIFFEL qui permettent de produire des logiciels qui peuvent être compilés et exécutés sur différentes plates-formes, Unix, Windows, OS2, VMS..., et qui disposent de bibliothèques de composants graphiques pour les présentations favorites de ces différents systèmes. Pour les applications interactives qui font un usage important des mécanismes graphiques, il y a cependant quelques limitations qui sont dues à certaines incompatibilité des environnements, notamment entre X11/Motif et Windows 95.

### 2.9.2 Compilation incrémentale

D'autre part, la puissance actuelle des ordinateurs rend possible la réconciliation des approches *statiques* et *dynamiques* pour analyser et exécuter le code. Il n'est plus nécessaire d'utiliser un langage dont la syntaxe est limitée aux parenthèses, ou absconse comme les langages postfixés, pour avoir un délai acceptable entre l'instant de la dernière modification et celui de l'affichage des premiers résultats. À partir du moment où ce délai est inférieur à dix, voire souvent à deux ou trois secondes, il devient inutile de gagner quelques dixièmes de seconde. Ce court répit suffit maintenant largement pour faire l'analyse syntaxique complète de quelques classes, les vérifications de sémantique statique les plus urgentes et traduire le code dans une représentation interne (réification, *bytecode*, langage intermédiaire...) qui se prête bien à l'interprétation. Ce principe n'exige pas *que tout le système en* *développement soit interprété*, mais seulement les parties en cours d'évolution, normalement réduites à quelques classes. C'est ce que faisait un interprète d'EIFFEL 2 en EIFFEL 2 [Brissi et Gautero, 1991] et maintenant l'environnement EIFFELBENCH d'ISE avec sa *Melting Ice Technology* (métaphore de la *glace fondante*).

Ainsi, le handicap des langages compilés peut-être quasiment éliminé, mais avec le gros avantage de pouvoir bénéficier du *typage statique* qui est utile non seulement pour la fiabilité (§ 2.8), mais aussi pour l'efficacité. S'il est vrai que les langages interprétés peuvent aussi améliorer nettement leur efficacité d'exécution, grâce à des techniques de caches ou de traduction en C, cela est beaucoup plus difficile à faire, à cause des nombreux choix qui restent encore ouverts pendant l'exécution. Pour la fiabilité en revanche, les langages interprétés ne peuvent pas inventer l'information de typage qui leur manque le plus souvent.

De plus, tous les contrôles statiques, en particulier les plus fins, n'ont pas besoin d'être systématiquement réalisés à chaque compilation : ils peuvent souvent être effectués plus tard, avec d'autres vérifications statiques ou dynamiques, comme des analyses globales plus fines que la règle des *catcalls* ou des évaluations d'assertions coûteuses (tests d'intégrité de bases de données, vérifications de quantifications...). Cette perspective de désynchronisation des différentes actions sur un programme en construction, déjà initiée avec les premières implémentations du langage C [22], offre de nombreuses possibilités qui ne sont pas encore toutes exploitées, y compris pour EIFFEL. Ainsi, le langage OQUAL [Collet, 1997] n'est réaliste qu'avec la possibilité d'évaluer ses quantifications de manière incrémentale, au fur et à mesure de la construction des classes, ou de manière différée pour les tests d'intégrité, la nuit ou les week-ends par exemple.

En prenant l'exemple de l'environnement EIFFELBENCH (Fig. 2.11), un texte EIFFEL est traduit selon différents niveaux de finition, dans l'une des trois formes suivantes :

- interprétable pour les parties en cours de mise au point, que Bertrand Meyer appelle « chaudes »,
- compilée, mais sans optimisation pour le reste d'une application en cours de développement (les parties « froides »),
- ou finalisée, c'est-à-dire traduite dans un langage standard (C/C++, JVM de JAVA), et sous la forme optimisée d'un kit portable avec ses procédures d'installation, sa documentation, ses tests...

Les changements de formes peuvent se faire de manière interactive, en utilisant des boutons comme ceux de la figure 2.11, et en adaptant le fichier *Ace* pour choisir le niveau de contrôle statique, l'armement des assertions, la gestion de l'évolution. L'interaction est également indispensable pour tout ce qui concerne l'information du programmeur (*browsing*) sur les classes qu'il réutilise et pour le débogage, d'utilisation assez rare en EIFFEL grâce aux assertions. Ainsi, l'environnement EIFFELBENCH permet de naviguer dans l'univers de classes défini par un fichier *Ace*, d'explorer ces classes selon différents points de vue (concrèts ou abstraits, primitives

22. Comme par exemple d'utiliser une séquence "`cc -O0; lint; cc -O3`" pour compiler rapidement, plus tard faire une vérification statique et enfin produire un code optimisé.

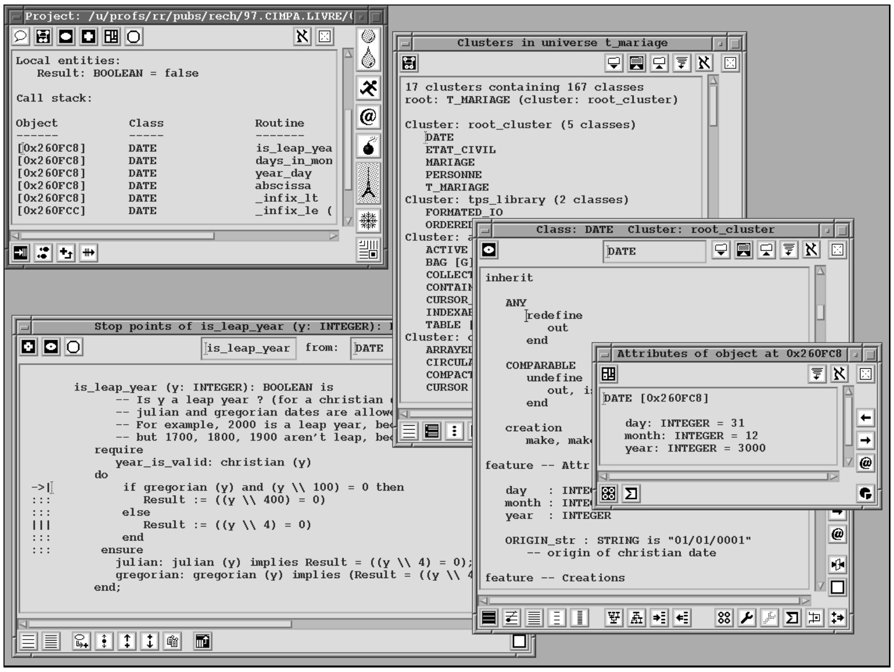



FIG. 2.11: *Un exemple de session avec l'environnement* EIFFELBENCH

propres à la classe ou héritées...), de connaître les super-classes ou les sous-classes d'une classe, ses clients ou ses fournisseurs. Les fonctions de débogage sont classiques et permettent des exécutions en pas à pas et des analyses de l'état des objets (cf. Fig. 2.11).

S'il est vrai qu'il existe de bons environnements interactifs pour d'autres langages (notamment pour SMALLTALK), même si nombre d'idées utilisées dans les environnements EIFFEL sont connues et sous-exploitées, nous ne connaissons pas d'autres environnements actuels qui les appliquent *toutes ensembles* aussi bien. Cela explique les excellents taux de productivité que nous avons constaté lorsqu'on développe en EIFFEL [Rousseau *et al.*, 1994b]. Ainsi, un modèle récent d'estimation des coûts de développement avec une approche par objets [Giliberti *et al.*, 1997] a été étalonné avec un bon coefficient de corrélation[23] sur des projets développés en SMALLTALK et C++ [Lorenz et Kidd, 1994]. En appliquant ce modèle au projet K2 [Lahire et Jugant, 1995], nous obtenons un effort d'environ 90 hommes×mois au lieu de 30 en réalité. Même si l'on peut contester la fiabilité de ce modèle d'estimation des coûts, cela indique clairement une productivité très élevée pour ce projet qui a été développé avec l'environnement EIFFELBENCH. De plus, l'effort réel comprend la fourniture d'une documentation interne de 2 500 pages et de tests assez complets, grâce aux assertions.

23. En comptant les routines exportées.

## 2.10 Conclusion

L'approche objet apporte des solutions potentielles à *tous* les problèmes que se pose le génie logiciel, pour obtenir à moindre coût des logiciels évolutifs, réutilisables, fiables, efficaces, bien documentés, portables et interfaçables. Bien que cette approche manque encore sérieusement de maturité, elle intégrera probablement tous les paradigmes actuels. C'est d'ailleurs une erreur de considérer l'approche objet sur le même plan que la programmation impérative, fonctionnelle ou logique. Elle se situe en effet sur une autre dimension, celle de la structuration des programmes, ce qui permet de la comparer aux principes de la « programmation structurée » des années soixante-dix ou de la « modularité classique » des années quatre-vingt. Dans les deux cas, elle dépasse de loin, et sans doute de manière définitive, les mérites des approches classiques.

Dans la famille des langages à objets dédiés au génie logiciel, le langage EIFFEL tient une place paradoxale. C'est à notre avis le langage actuel qui offre la solution la plus complète et la plus cohérente, ce qui en fait déjà un excellent support pour l'enseignement [Boussard et Rousseau, 1995]. Les environnements actuels pour EIFFEL sont interactifs, performants et conviviaux, avec des bibliothèques conséquentes de milliers des primitives réutilisables ; ils sont certainement à l'origine des taux élevés de productivité que nous avons constatés. Pourtant, le nombre des utilisateurs d'EIFFEL reste très inférieur à ce que l'on pourrait imaginer, car depuis FORTRAN et COBOL, la plupart des utilisateurs suivent les standards du moment aujourd'hui C++ et JAVA. Il est pourtant possible de développer en EIFFEL du logiciel évolutif et bien documenté et de le traduire dans ces derniers standards.

Toutes les constructions d'EIFFEL nous paraissent utiles, sans redondance. Les bases du langage sont solides, ce qui lui permet d'évoluer ou de servir de support à des extensions [Rousseau *et al.*, 1994a] pour acquérir de nouvelles aptitudes [Rousseau, 1995], pour les applications persistantes [Lahire, 1992], parallèles [Caromel, 1993] ou réparties [Jézéquel, 1993a] ou pour la spécification des programmes [Collet, 1997]. En revanche, les possibilités de métaprogrammation n'ont pas encore été suffisamment explorées.

# 3

# Introduction à C++ et comparaison avec Eiffel

CONÇU PAR BJARNE STROUSTRUP, le langage C++ est devenu un standard de fait en matière de programmation par objets, que ce soit dans le monde industriel ou dans le monde académique. Ce chapitre n'a pas la prétention de faire le tour complet de ce langage. Il en présente simplement les principaux mécanismes permettant la définition et l'utilisation d'objets, dans le but de mettre l'accent sur leurs avantages et leurs inconvénients comparativement au langage EIFFEL, avec lequel C++ est souvent mis en compétition.

Pour bien comprendre ce qui suit ou approfondir ses connaissances sur C++, le lecteur peut consulter [Stroustrup, 1992], qui constitue de fait le manuel de référence du langage, et [Stroustrup, 1994], dans lequel Bjarne Stroustrup explique les principes qui ont guidé la définition du langage et qui permet d'apprécier son évolution durant ces dix dernières années. Ces deux ouvrages ne sont cependant pas conçus pour apprendre à programmer en C++. Parmi les nombreux livres qui sont précisément consacrés à cette question, [Lippman, 1991] est considéré comme l'un des meilleurs ; [Musser et Saini, 1996] est un complément indispensable pour apprendre à utiliser la bibliothèque standard du langage, appelée STL [1].

Le langage EIFFEL, quant à lui, est présenté dans le *chapitre 2*. Les principes dont s'est inspiré son créateur, Bertrand Meyer, sont décrits et discutés dans [Meyer, 1990]. Bien qu'assez ancien maintenant, cet ouvrage sert encore de référence. EIFFEL a évolué depuis, et sa dernière version, qui porte le numéro 3, est décrite en détail dans [Meyer, 1994a]. Un certain nombre d'ouvrages sont consacrés à la programmation en EIFFEL, comme [Switzer, 1995] [Jézéquel, 1996] et [Gautier *et al.*, 1996], sans oublier [Wiener, 1995], qui présente aussi une comparaison entre EIFFEL et C++.

1. *Standard Template Library*.

## 3.1 Un peu d'histoire

Le langage C a été imaginé au début des années 1970, dans les laboratoires Bell de la compagnie AT&T. Selon l'avis même de ses concepteurs, B.W. Kernighan et D.M. Ritchie, c'est un langage de bas niveau, censé permettre au programmeur d'utiliser au mieux les ressources de son installation [Kernighan et Ritchie, 1988]. Conçu pour le système Unix, le langage s'est rapidement répandu sur tous les systèmes d'exploitation usuels. Ce succès s'est accompagné d'une prolifération de dialectes, qui a rendu nécessaire un travail de normalisation effectué de 1983 à 1988 par un comité de l'Ansi [Kernighan et Ritchie, 1990].

L'histoire de C++ commence, elle aussi, au centre de recherche des laboratoires Bell. En 1979, Bjarne Stroustrup développe le langage C *with classes*, une extension de C, incluant, entre autres, la définition de classes (§ 3.2) et l'héritage simple (§ 3.5.1). La liaison dynamique (§ 3.5.2) n'est introduite qu'en 1983, pratiquement en même temps que le langage change de nom pour devenir C++. Lorsque la version 1.0 du compilateur est commercialisée, en 1985, le langage correspond à la description donnée dans [Stroustrup, 1986], qui inclut notamment la surcharge des fonctions et des opérateurs (§ 3.4), les notions de constante et de référence (§ 3.3.1), ainsi que les opérateurs de gestion dynamique de la mémoire, `new` et `delete` (§ 3.3.2). Durant la période 1986–1989, l'utilisation de C++ se développe et différents compilateurs, dont certains pour PC, sont commercialisés par les éditeurs de logiciels. Le compilateur GNU C++ est même disponible dans le domaine public à la fin de 1987. La version 2.0 du compilateur AT&T, autorisant l'héritage multiple (§ 3.5.3), apparaît en 1989. Le comité X3J16 de l'Ansi est chargé d'établir une norme et retient le manuel de référence rédigé par Margaret A. Ellis et Bjarne Stroustrup [Ellis et Stroustrup, 1990] comme base de travail. Les extensions proposées dans ce manuel, la généricité (§ 3.6) et la gestion d'exceptions, sont acceptées. Depuis 1991, les travaux de normalisation sont menés conjointement avec le groupe de travail WG-21 de l'Iso.

Dès l'introduction de son ouvrage relatant la genèse et l'évolution de C++ [Stroustrup, 1994], Bjarne Stroustrup affirme clairement sa volonté d'intégrer les moyens de structuration des programmes offerts par SIMULA(classes et héritage) [Kirkerud, 1989] au langage C. L'auteur est conscient des défauts de C, mais considère qu'il offre néanmoins de multiples avantages : souplesse, efficacité, portabilité, disponibilité sur un grand nombre de plates-formes, etc. Dans l'esprit de l'auteur, C++ doit rester compatible avec C tant que cette compatibilité n'empêche pas d'atteindre les objectifs fixés [Koenig et Stroustrup, 1989] : simplifier le travail des programmeurs en enrichissant C de facilités permettant l'abstraction de données et la programmation par objets. C++ est un langage et ne doit pas devenir un système intégrant un environnement de programmation, tel que, par exemple, SMALLTALK [Goldberg, 1984]. Il ne doit pas non plus obliger les programmeurs habitués à C à changer radicalement leurs habitudes. Les constructions de C ont donc été intégrées à C++ sans chercher une compatibilité totale et en procédant, quand il le fallait, à des modifications syntaxiques et des extensions. Certaines d'entre elles ont d'ailleurs été retenues dans la norme C ANSI.

## 3.2 Les classes

### 3.2.1 La définition d'une classe

C++ est un langage fortement typé : les variables, les paramètres et les (résultats de) fonctions doivent faire l'objet d'une déclaration qui précise leur type, autrement dit le domaine auquel appartient leur valeur pendant l'exécution. Des règles de typage définissent la validité des instructions du code source en fonction des types des entités qu'elles manipulent. Le compilateur peut ainsi contrôler statiquement, avant l'exécution, le respect de ces règles, afin de garantir une exécution exempte d'erreur de type [Aho *et al.*, 1989, chapitre 6].

Une classe définit un nouveau type, qui s'ajoute aux types fondamentaux prédéfinis (entiers, réels, booléens, caractères). La déclaration de la classe mentionne toutes les informations nécessaires pour que le compilateur puisse traiter les déclarations des variables de ce type, tant en ce qui concerne l'allocation mémoire que les contrôles de types : structure physique des objets, sous forme d'attributs appelés *données membres* (Fig. 3.1, lignes 2–4), et signatures[2] des opérations applicables aux objets, appelées *fonctions membres* (lignes 9–14). La filiation avec C apparaît clairement, l'analogie entre la déclaration d'une classe C++ et celle d'une structure C étant flagrante.

Le texte constituant la définition complète d'une classe n'est pas nécessairement regroupé dans un seul fichier source : les corps des fonctions membres peuvent être donnés à part, dans différents fichiers, pour être compilés séparément. Cependant, comme la déclaration de classe doit figurer dans tous les fichiers où le nom de la classe est utilisé, il est habituel de la conserver dans un fichier d'inclusion[3]. La directive `#include`, reprise de C, permet ensuite d'inclure cette déclaration partout où elle est nécessaire (ligne 17). Les mêmes noms pouvant servir à désigner des membres de classes différentes ou des fonctions ordinaires, chaque fonction est rattachée à sa classe grâce à l'opérateur de *résolution de portée* :: (lignes 18 à 22).

Les membres d'une classe peuvent être privés (ligne 1) ou publics (ligne 5), les membres publics constituant l'interface de la classe. Les données membres publiques peuvent non seulement être consultées mais aussi modifiées par tout client de la classe. Afin de préserver le masquage d'information, il est donc préférable de garder privés les champs de la représentation et de définir des fonctions d'accès pour les consulter (lignes 9–11). Pour satisfaire les objectifs apparemment contradictoires d'efficacité et de sécurité, ces fonctions peuvent être indiquées comme devant être *expansées en ligne*, c'est-à-dire traitées par le compilateur à la manière d'une macro : les appels de la fonction sont remplacés par les instructions du corps de la fonction, dans lesquelles les paramètres formels ont été remplacés par les paramètres effectifs de l'appel.

Une fonction expansée en ligne peut être définie de deux façons différentes, en plaçant la définition complète de la fonction, corps y compris, dans la déclaration de la classe (lignes 14–16) ou en faisant précéder la définition de la fonction du

2. Selon une syntaxe conforme à la norme C ANSI.
3. *Header file*.

```
   // Ceci est un commentaire : fichier Article.h
   class Article {
1    private:  // Partie privee
2      char* _designation;    // Nom de l'article
3      int _quantite;         // Quantite en stock
4      float _prixHT;         // Prix de vente hors taxes
5    public:   // Partie publique
6      friend class Grossiste;
7      friend void Droguerie::vendre();
8      friend void tester();
9      char* designation();   // Consulter la designation
10     int quantite();        // Consulter la quantite en stock
11     float prixHT() const;  // Consulter le prix de vente HT
12     float prixTTC();       // Calculer le prix de vente TTC
13     void retirer(int q);   // Retirer 'q' articles du stock
14     void ajouter(int q) {  // Ajouter 'q' articles au stock
15       _quantite += q;
16       }
     };
```

```
   // Implantation des fonctions membres de la classe Article
17 #include "Article.h"
18 inline char* Article::designation() {
     return _designation;
     }
19 inline int Article::quantite() {
     return _quantite;
     }
20 inline float Article::prixHT() const {
     return _prixHT;
     }
21 inline float Article::prixTTC() {
     return (1.206 * _prixHT);
     }
22 inline void Article::retirer(int q) {
     _quantite -= q;
     }
```

FIG. 3.1: *Déclaration et implantation de la classe* `Article` *décrivant un article du stock d'un magasin.*

mot clé `inline` (lignes 18 à 22). Ainsi, la consultation des valeurs des membres privés `_designation`, `_quantite` et `_prixHT` ne provoque pas de surcoût de temps d'exécution. Il est tout à fait normal de procéder pareillement pour les fonctions `ajouter` (ligne 14), `prixTTC` (ligne 21), et `retirer` (ligne 22), étant donné leur simplicité.

Ce mécanisme d'expansion en ligne est plus simple et plus fiable que le mécanisme de définition de macro-fonctions de C par la directive `#define`, car, contrairement à ce dernier, il n'opère pas par substitution de texte. Le compilateur fait en sorte que l'effet d'un appel soit rigoureusement identique à celui du texte expansé.

### 3.2.2 Comparaison avec Eiffel

Si, tout comme EIFFEL, C++ est un langage fortement typé, il s'en distingue cependant sur deux points importants. D'une part, EIFFEL impose un style de programmation par objets : une application est un *système* de classes [Meyer, 1990] et toute routine (c'est-à-dire procédure ou fonction) appartient à l'une de ces classes. Il n'en est rien en C++, conformément aux intentions de son auteur : une application peut inclure des fonctions (en particulier le programme principal `main()`) et des variables qui ne sont pas attachées à des classes. D'autre part, le texte d'une classe EIFFEL constitue une unité syntaxique indépendante. Au contraire, comme nous venons de le montrer, la définition d'une classe C++ ne constitue pas obligatoirement une seule unité syntaxique et peut être réparties dans plusieurs fichiers sources. La notion de classe est le concept central en EIFFEL, alors qu'elle n'est qu'un outil parmi d'autres en C++.

Le masquage d'information est plus rigoureux en EIFFEL qu'en C++, car l'unité de protection est l'objet : seules les primitives[4] privées de l'*objet courant* sont accessibles dans le corps d'une routine. En C++, l'unité de protection est la classe : les membres privés de *tout objet* d'une classe sont accessibles dans le corps des fonctions membres de cette classe. En outre, une donnée membre publique d'une classe C++ peut être directement modifiée par n'importe quel client de la classe. Ceci est impossible en EIFFEL : un attribut public ne peut être consulté qu'en lecture et il faut impérativement définir une routine spécifique pour être en mesure de modifier sa valeur.

En revanche, les mécanismes d'exportation d'EIFFEL et C++ sont complémentaires. EIFFEL autorise l'exportation de chaque primitive d'une classe de façon différenciée vers une ou plusieurs classes explicitement nommées (dans une clause **feature**). C++ ne permet d'exporter que l'*ensemble* de la partie privée d'une classe vers les *amies* de la classe, qui peuvent être d'autres classes (cf. Fig. 3.1, ligne 6), mais aussi des fonctions membres d'autres classes (ligne 7), voire des fonctions ordinaires (ligne 8).

## 3.3 Les objets

### 3.3.1 La représentation des objets

La déclaration d'une variable d'un type défini par une classe, par exemple `Article` (Fig. 3.2, ligne 1), a pour effet d'allouer l'espace nécessaire à la représentation des trois données membres de l'objet (une chaîne de caractères, `_designation`, un nombre entier, `_quantite`, et un nombre réel, `_prixHT`, cf. Fig. 3.1). La déclaration d'un pointeur sur un objet de type `Article` (ligne 2) n'alloue que l'espace nécessaire à la représentation d'une adresse, sans qu'un objet de type `Article` ne soit créé. C'est la représentation standard en EIFFEL. La déclaration d'une *référence* à un objet de type `Article` (ligne 3), notion inconnue en C, définit un *alias*, c'est-à-dire une nouvelle désignation pour un objet existant : une telle déclaration doit donc

4. Une primitive (*feature*) est l'équivalent d'un membre C++, donnée ou fonction.

```
  #include "Article.h"
  main() {
1   Article art1, art2, art3, art4;
2   Article *p1, *p2;
3   Article &ref1 = art1;
    ...
    p1 = p2;      // Affectation d'adresses (pointeurs)
    art1 = art2;  // Affectations d'objets
4   ref1 = art3;  // Equivalent a : art1 = art3
5   art4 = ref1   // Equivalent a : art4 = art1
    }
```

FIG. 3.2: *Exemples de déclarations et d'utilisations de pointeurs et de références.*

impérativement être accompagnée d'une initialisation. Ici, la référence `ref1` est en quelque sorte un synonyme de `art1`. La représentation d'une référence est identique à celle d'un pointeur, mais l'accès à la valeur désignée par une référence s'effectue sans utiliser l'opérateur de déréférence `*` (lignes 4–5).

Lorsque des paramètres formels sont déclarés de type référence (Fig. 3.3, ligne 1), le passage des paramètres s'effectue par adresses. L'écriture du corps de la fonction s'en trouve notablement simplifiée, puisqu'il n'y a plus besoin d'utiliser l'opérateur de déréférence pour accéder aux objets désignés par les paramètres (lignes 2–4). Il en est de même pour les appels de la fonction (ligne 6) : il devient inutile de donner explicitement des adresses en paramètres, comme en C, en utilisant l'opérateur *adresse-de* `&` (ligne 5). La lisibilité du texte source est accrue.

L'adjectif `const` peut être appliqué à un type, y compris un type défini par l'utilisateur, pour déclarer des constantes. Le type garde ses propriétés initiales, mais les valeurs des variables de ce type ne peuvent pas être modifiées. A l'instar des fonctions expansées en ligne, cette nouvelle construction permet de se passer de la

```
  void swapC (int *i, int *j) {
  /* Style C : echanger les valeurs de 'i' et 'j'. */
    int tmp = *i;
    *i = *j;
    *j = tmp;
    }
1 void swapCC (int &i, int &j) {
  // Style C++ : echanger les valeurs de 'i' et 'j'.
2   int tmp = i;
3   i = j;
4   j = tmp;
    }

  main() {
    int m = 1, n = 2;
5   swapC(&m, &n);
6   swapCC(m, n);
    }
```

FIG. 3.3: *L'utilisation de paramètres formels de type référence (ligne 1) simplifie l'écriture du code source.*

```
1   int estDans(const int el, const int tab[]) {
    // ‘el’ appartient-il au tableau ‘tab’?
    // Les parametres ‘el’ et ‘tab’ ne peuvent etre modifies
    // dans le corps de la fonction.
      ...
2     tab[i] = 0;  // Instruction illicite
      ...
      }
```

FIG. 3.4: *Exemple de paramètres formels constants.*

directive `#define`. Elle est notamment utile pour les tableaux, qui sont représentés par des pointeurs, comme en C. Passer un tableau en paramètre effectif revenant à passer un pointeur, il est toujours possible de modifier intempestivement les valeurs du tableau dans le corps de la fonction appelée, sauf si le paramètre formel correspondant est déclaré constant (Fig. 3.4, ligne 1). Le compilateur peut alors vérifier que le paramètre ne figure pas dans une instruction susceptible de changer la valeur d'un des éléments du tableau (ligne 2). Si cette précaution est superflue pour le paramètre formel `el`, de type entier, elle reste cependant utile pour se prémunir contre des erreurs pendant la mise au point et pour documenter la fonction.

Le pointeur `this` désigne l'objet courant, comme l'identificateur **Current** d'EIFFEL. Dans le texte d'une fonction membre d'une classe, `Article` par exemple, `this` est, par défaut, un pointeur constant sur un objet du type défini par la classe :

```
Article *const this;
```

Ceci interdit bien de modifier la valeur de `this` en tant que pointeur, mais autorise la modification de l'objet (courant) qu'il désigne, sans passer par une fonction prévue à cet effet. Pour y remédier, `this` doit être un pointeur constant sur une constante de type `Article` :

```
const Article *const this;
```

De cette façon, ni la valeur pointée, ni la valeur même du pointeur, ne peuvent être modifiées. Cette protection est effectivement garantie dans une fonction dite constante, c'est-à-dire déclarée avec la spécification `const`. En toute rigueur, les fonctions de consultation des membres privés, par exemple `prixHT` (cf. Fig. 3.1, lignes 11 et 20), doivent être déclarées constantes, d'une part à titre de garde-fou pendant les phases de mise au point et de maintenance, et, d'autre part, à titre de documentation.

### 3.3.2 La création des objets

Contrairement à EIFFEL, C++ autorise la manipulation explicite de pointeurs sur des objets. La déclaration d'un pointeur (`Article*`) n'allouant que l'espace mémoire nécessaire à la représentation d'une adresse, il faut fournir les moyens d'allouer et de libérer dynamiquement des zones d'espace mémoire. En C, ce sont les fonctions `malloc()`, pour l'allocation, et `free()`, pour la libération, qui remplissent ce rôle. En C++, ce sont les opérateurs `new` et `delete`. Ainsi, l'expression

```
   // Fichier Article.h : declaration de la classe Article
   class Article {
     private:
       char *_designation;   // Nom de l'article
       int _quantite;        // Quantite en stock
       float _prixHT;        // Prix de vente hors taxes
     public:
1      Article(char *des, int qte, float prix) {
2        _designation = des;
3        _quantite = qte;
4        _prixHT = prix;
5        }
       ...
     };
```

```
     #include "Article.h"
     ... {
6      Article *tab = new Article[100];
       Article *savon;
7      Article soude("Soude caustique", 20, 39.95);
       ...
8      savon = new Article("Savonnette", 100, 4.95);
9      qte = soude.quantite();
10     savon->retirer(10);
11     delete savon;
12     delete [] tab;
       }
```

FIG. 3.5: *Exemples de créations et de destructions d'objets.*

`new Article` (Fig. 3.5, ligne 8) alloue l'espace nécessaire à la représentation d'un objet de type `Article` et retourne l'adresse de la zone allouée. Cette zone peut être explicitement libérée grâce à l'opérateur `delete` (ligne 11). Dans l'instruction de libération de la ligne 12, les crochets indiquent que l'espace *complet* alloué au tableau `tab` (ligne 6), et non pas seulement son premier élément, doit être restitué.

Lors de sa création, un objet est initialisé par un *constructeur*, qui est une fonction membre qui porte le même nom que sa classe d'appartenance et n'a pas de type résultat (cf. Fig. 3.5, lignes 1–5). Si une classe n'est pas explicitement dotée d'un constructeur, et seulement dans ce cas, un constructeur sans paramètre, baptisé *constructeur par défaut*[5], est automatiquement généré pour la classe. Dès qu'un constructeur différent du constructeur par défaut est défini, il doit impérativement être utilisé, tant pour déclarer une variable du type correspondant (ligne 7), que pour allouer dynamiquement un objet avec l'opérateur `new` (ligne 8).

Pour une classe donnée, il n'y a pas lieu d'utiliser de constructeur pour déclarer une donnée membre, car l'exécution d'un constructeur de la classe débute *implicitement* par l'appel des constructeurs par défaut de chacune des données membres. Si la classe définissant le type d'une donnée membre n'a pas de constructeur par défaut ou si ce dernier ne convient pas, il faut remplacer l'appel implicite au constructeur par défaut par un appel explicite (et paramétré) à un constructeur existant.

5. Cette dénomination désigne également tout constructeur sans paramètre défini explicitement par l'utilisateur.

```
  #include "Article.h"
  class Rayon {
      int _numero;
1     Article _articleExpose;
    public:
2     Rayon(int num, char *nom, int qte, float prix)
3       : _articleExpose(nom, qte, prix), _numero(num) { }
    ...
    };
```

FIG. 3.6: *Exemple de constructeur avec une liste d'initialisations.*

Ces appels figurent dans une liste d'initialisations (Fig. 3.6, ligne 3), qui suit suit la liste des arguments du constructeur de la classe contenant la donnée membre (ligne 2). Cette liste indique ici que le membre `_articleExpose` (ligne 1) est initialisé par le constructeur de la classe `Article`, qui reçoit `nom`, `qte` et `prix` en paramètres, et que le membre `_numero`, de type entier, est initialisé avec la valeur du paramètre `num`. Comme il n'y a pas d'autre action à effectuer, le corps proprement dit du constructeur est vide.

Une fois qu'un objet est créé, l'utilisation d'un de ses membres se note de la même façon que l'accès à un champ d'une structure C, grâce à une notation pointée (cf. Fig. 3.5, ligne 9). Lorsque la cible est un pointeur, l'opérateur `->` remplace le point (ligne 10).

### 3.3.3 La destruction des objets

Comme en C, l'espace mémoire nécessaire à la représentation d'une variable locale (dans une fonction ou un bloc d'instructions entre accolades) déclarée avec le spécificateur de classe d'allocation `auto`[6] est alloué à chaque activation du bloc où figure la déclaration. L'espace est implicitement libéré à la sortie du bloc. Si la variable est un pointeur, l'espace peut être explicitement libéré à tout moment avec l'opérateur `delete`. Si la classe de l'objet désigné par la variable comporte une fonction membre appelée *destructeur*, celle-ci est implicitement exécutée avant la libération de l'espace mémoire occupé par l'objet, que cette libération soit provoquée par la sortie du bloc ou l'usage de l'opérateur `delete`.

Un destructeur ne peut en aucun cas être invoqué explicitement. Il n'a ni paramètre ni résultat. Son nom est formé par la concaténation du caractère `~` et du nom de sa classe (Fig. 3.7, lignes 5–8). L'exemple présente un extrait de la définition d'une classe décrivant une pile d'entiers. Le constructeur (ligne 4) alloue l'espace nécessaire au stockage de `max` entiers. La destruction d'un *objet* de type `PileInt`, c'est-à-dire d'une zone mémoire formée d'un pointeur (`_pile`, ligne 1) et de deux entiers (`_max` et `_sommet`, lignes 2 et 3), est automatique. En revanche, la libération de la zone désignée par le pointeur `_pile` (le tableau des éléments proprement dit) n'est pas automatique. C'est donc bien le destructeur qui doit s'en charger (ligne 7) et il est appelé chaque fois qu'un objet de type `PileInt` est détruit (lignes 9 et 10).

6. Ou sans spécificateur, `auto` étant le défaut.

```
// Fichier PileInt.h : declaration de la classe PileInt
#include <iostream.h>  // Pour les entrees-sorties
class PileInt {
  private:
    int *_pile;  // Tableau implantant la pile
    int _max;    // Capacite maximale de la pile
    int _sommet; // Indice du sommet de la pile
  public:
    PileInt(int max) : _sommet(-1), _max(max), _pile(new int[max]) {}
    ~PileInt() {
      if (!estVide()) cout << "*** Attention : pile non vide !\n";
      delete [] _pile;
      }
    ...
  };
```

```
#include "PileInt.h"
... { // Debut de bloc
  PileInt pint(500);            // Creation d'une pile de 500 elements
  PileInt *pp = new PileInt(5);  // Pointeur sur une pile de 5 elements
  ...
  delete pp;  // Destruction explicite de la pile designee par 'pp'
  }   // Fin du bloc : destruction de la pile 'pint'
      // et des autres objets locaux (le pointeur pp)
```

FIG. 3.7: *Constructeur et destructeur de la classe `PileInt`.*

### 3.3.4 Comparaison avec Eiffel

Les points de vue adoptés en C++ et en EIFFEL sur la représentation et la création des objets sont radicalement différents.

C++ autorise la manipulation explicite de pointeurs sur des objets, alors que cette notion n'existe pas *explicitement* en EIFFEL. En effet, une variable EIFFEL est par défaut une *référence*, c'est-à-dire un pointeur : l'affectation `x := y` a pour effet d'affecter à `x` la référence (l'adresse) contenue dans `y`, les deux variables désignant ensuite le même objet. Toutefois, il n'existe ni opérateur de déréférence ni opérateur adresse-de : le fait de manipuler des pointeurs reste totalement transparent au programmeur. En cas de besoin, il est toujours possible de déclarer une variable d'un type expansé (avec le mot clé **expanded**), pour indiquer que la variable n'est pas une référence mais qu'elle représente directement un objet du type nommé, comme dans une déclaration de variable d'un type défini par une classe en C++.

A l'inverse, en C++, les pointeurs sont explicitement déclarés et manipulés comme tels : l'affectation à un pointeur modifie l'adresse qu'il contient et l'accès à l'objet désigné nécessite dans tous les cas l'usage de l'opérateur de déréférence `*`. Quant aux références, elles doivent impérativement être initialisées à leur déclaration, *sans qu'il soit possible de modifier ultérieurement l'adresse qu'elles contiennent*. Il n'y a donc aucune équivalence entre un pointeur C++ et une référence au sens d'EIFFEL, pas plus qu'entre une référence C++ et une référence EIFFEL.

En C++, un objet est initialisé à sa création par un *constructeur* : cette convention rend superflue l'utilisation d'une syntaxe et d'un mot clé spécifiques pour la définition des constructeurs, comme c'est le cas en EIFFEL, où un objet doit être explicitement créé par une instruction d'instanciation (notée `!!`). En revanche, un objet EIFFEL n'a pas à être explicitement détruit : la mémoire allouée dynamiquement aux objets est récupérée *automatiquement* par un ramasse-miettes lorsqu'elle n'est plus accessible, comme dans un interpréteur LISP, sans que le programmeur ait à s'en soucier. La notion de destructeur demeure inconnue. En C++ , ce travail est laissé à la charge du programmeur, qui doit explicitement prévoir les instructions adéquates.

C++ ne respecte pas le principe de référence uniforme appliqué en EIFFEL, où l'accès à un attribut (une donnée membre) et l'appel d'une routine (une fonction membre) sans paramètre se notent pareillement. En C++, l'appel d'une fonction sans paramètre se note, comme en C, en faisant suivre le nom de la fonction d'un couple de parenthèses (cf. Fig. 3.5, ligne 9). Pour être en mesure d'utiliser correctement un service, le client d'une classe doit donc nécessairement savoir si ce service est implanté par une donnée ou par une fonction sans paramètre. Afin d'éviter qu'un changement dans l'implantation d'une classe ne provoque des modifications chez ses clients, il est une fois de plus souhaitable de définir systématiquement des fonctions d'accès aux données membres, si possible expansées en ligne.

## 3.4 La surcharge et les conversions

La plupart des opérateurs [7], peuvent être surchargés, mais sans en modifier l'arité et la priorité telles qu'elles sont fixées par le langage. De la même façon, plusieurs fonctions, qu'elles soient non membres, membres d'une même classe ou membres de classes différentes, peuvent porter le même nom, à condition que chaque signature soit unique, par le nombre ou le type des paramètres, sans que cela soit exclusif.

La surcharge est particulièrement attractive pour les constructeurs (Fig. 3.8, lignes 1–3), car elle permet d'enrichir les possibilités de déclarations et d'initialisations (lignes 5–7), sans avoir à utiliser plusieurs constructeurs aux noms différents, comme ce serait le cas en EIFFEL. En effet, ce dernier interdit de définir plusieurs routines de même nom dans une même classe.

Comme en C, il existe des opérations de conversion implicites entre valeurs des types fondamentaux (`int`, `float`, etc.), mais il est également possible de définir des opérations de conversion entre deux valeurs de types quelconques : un constructeur à paramètre unique définit en fait une opération de conversion du type du paramètre vers le type défini par la classe du constructeur. À ce titre, le constructeur de la ligne 2 permet non seulement d'initialiser une instance de la classe `Temps`, mais encore de convertir implicitement un entier en une telle instance (ligne 8).

Une classe peut également être explicitement pourvue d'opérateurs de conversion sous forme de fonctions membres. Ici, l'opérateur `int` (ligne 4) convertit un temps en une valeur entière (ligne 9). Cet opérateur est implicitement appelé pour convertir un opérande ou un paramètre dont le type est inadéquat (ligne 10).

7. A l'exception de `::`, `.*`, `.`, `?:` et des opérateurs s'appliquant aux types fondamentaux, l'addition des entiers par exemple.

```
// Fichier Temps.h : declaration de la classe Temps
class Temps {
  private:
    int _cts;  // Un temps est exprime en centiemes de secondes
  public:
    Temps(int mn, int sec, int cts) { _cts = (mn * 6000) + (sec * 100) + cts; }
    Temps(int cts) { _cts = cts; }
    Temps() { _cts = 0; }          // Constructeur par defaut
    operator int() const { return _cts; }
    ...
};
```

```
#include "Temps.h"
int main() {
  Temps t0(5, 30, 0);
  Temps t1(60);
  Temps t2;
  int i;
  ...
  t2 = 100;      // Equivalent a : t2 = Temps(100)
  i = int(t1);   // Equivalent a la notation C : i = (int)t1
  i = t0;        // Equivalent a : i = int(t0)
}
```

FIG. 3.8: *Exemple de surcharge avec le constructeur de la classe* `Temps`.

L'existence d'opérations de conversion pouvant être invoquées implicitement complique notablement le mécanisme de sélection de l'opération à exécuter en cas de surcharge. En effet, s'il n'existe pas de définition dont la signature corresponde exactement aux types des paramètres effectifs d'un appel d'une fonction surchargée, l'opération sélectionnée est l'intersection des ensembles des opérations « qui conviennent le mieux » pour chacun des paramètres pris séparément, *tout en prenant en compte les possibilités de conversions de types*. Si plusieurs opérations « conviennent aussi bien », l'appel est considéré comme ambigü et rejeté [8].

Il est donc fortement recommandé de ne pas abuser de la surcharge des opérateurs et des conversions. Les programmes deviennent rapidement illisibles et leur maintenance se transforme en véritable casse-tête, quand on sait qu'il est possible de surcharger l'opérateur d'affectation = et l'opérateur d'appel de fonction `()`, entre autres.

## 3.5 L'héritage

### 3.5.1 L'héritage simple

Dans la terminologie C++, une sous-classe, `Vetement`, *dérive* d'une classe de base, `Article` (Fig. 3.9, ligne 2). Une dérivation est qualifiée par un mot clé précédant le nom de la classe de base. Avec la qualification `private`, tous les membres publics hérités de la classe de base deviendraient privés dans la classe

8. Pour l'énoncé exact et commenté de cette règle, voir [Stroustrup, 1992, § r.13.2].

```
  // Fichier Article.h : declaration de la classe Article
  class Article {
1   protected:
      char *_designation;  // Nom de l'article
      int _quantite;       // Quantite en stock
      float _prixHT;       // Prix de vente hors taxes
    public:
      ...
    };
```

```
  #include "Article.h"
2 class Vetement : public Article {
    protected:
      int _taille;      // Taille du vetement
      char *_coloris;   // Coloris du vetement
    public:
      Vetement(char *des, int qte, float prix, int taille, char *color)
3       : Article(des, qte, prix), _taille(taille), _coloris(color) { }
      int taille() const {return _taille; }
      char *coloris() const {return _coloris; }
4     void solder(float remise) { _prixHT *= (1. - remise); }
    };
```

FIG. 3.9: *La classe `Vetement` dérive de la classe `Article`.*

dérivée. Dans l'exemple, le mot clé `public` (ligne 2) indique que les membres hérités conservent leur statut d'origine, public ou privé. En particulier, le membre `_prixHT` reste une donnée privée de la classe `Article`. La fonction `solder`, membre de la classe `Vetement` (ligne 4), ne peut donc pas y accéder directement, car ce droit est réservé aux seules fonctions membres ou amies de la classe `Article`, à l'exclusion des classes dérivées. Le problème est résolu en déclarant *protégées* (`protected`) les données membres de la classe `Article` (ligne 1) : un membre protégé est accessible aux fonctions membres de la classe, aux amies de la classe, ainsi qu'aux classes dérivées et aux amies de ces classes dérivées. Lorsqu'une dérivation est qualifiée avec le mot clé `protected`, les membres publics et protégés de la classe de base deviennent des membres protégés de la classe dérivée.

Un objet d'une classe dérivée est construit en exécutant d'abord le constructeur de sa classe de base, puis les constructeurs des données membres, dans l'ordre de leur déclaration, et enfin le corps du constructeur proprement dit de la classe dérivée. Pour la classe de base et les données membres, ce sont les constructeurs par défaut qui sont invoqués, à moins que la liste d'initialisations du constructeur de la classe dérivée ne mentionne des appels à d'autres constructeurs, comme à la ligne 3 de la figure 3.9 : le constructeur de la classe `Article` qui est invoqué est celui qui est défini avec trois paramètres à la figure 3.5 (lignes 1–5) et les données membres `_taille` et `_coloris` sont explicitement initialisées.

C++ est un langage à structure de bloc et obéit aux règles d'identification classiques dans les langages à structure de bloc inspirés d'Algol [Woodward et Bond, 1974]. Ces règles sont néanmoins compliquées par le fait que les règles qui régissent les conversions implicites et la surcharge des noms de fonctions (autorisant la coexistence de plusieurs définitions d'une même fonction dans un même bloc) se com-

```
  class Base {
    public:
1     int f(int i);
2     int f(float r);
    };
  class Derivee : public Base {
    public:
3     int f(int i);
    };
  ...
  int main() {
    Derivee d;
4   int j = d.f(3.14159);  // Equivalent a :
    }                      // int j = d.Derivee::f(3)
```

FIG. 3.10: *Exemple de redéfinition de fonction.*

binent aux règles de portée des identificateurs. A moins d'utilisation explicite de l'opérateur de résolution de portée ::, la recherche de la déclaration d'un identificateur est fondée sur les règles de portée : le bloc courant, puis, si nécessaire, les blocs englobants successifs, sont examinés dans l'ordre. S'il s'agit d'un identificateur de fonction, il faut considérer les éventuelles surcharges figurant dans *le même bloc* que la déclaration trouvée, sans oublier les conversions implicites pouvant s'appliquer aux paramètres.

Dès qu'une fonction est redéfinie dans une classe dérivée, la nouvelle définition masque *toutes* les définitions héritées de la classe de base. Donc, si cette nouvelle définition est elle-même surchargée, seules les définitions de la classe dérivée sont prises en compte pour sélectionner la définition qui convient le mieux lorsque la fonction est appliquée à un objet de la classe dérivée. Par exemple, la redéfinition de la fonction `f()` à la ligne 3 de la figure 3.10 masque les deux définitions de la classe de base (lignes 1 et 2). C'est la seule version candidate pour résoudre l'appel de la ligne 4, puisque la variable `d` est de type `Derivee`. Bien qu'elle ne semble pas convenir, car elle admet un paramètre de type entier, elle est quand même utilisée, une conversion implicite transformant le paramètre effectif, de type réel, en valeur entière !

### 3.5.2 La liaison dynamique

L'héritage va de pair avec la liaison dynamique et le polymorphisme de variable. Dans l'exemple de la figure 3.11, le tableau `commandes`, déclaré contenir des pointeurs de type `Article` (ligne 3), peut bien entendu contenir des pointeurs de ce type (ligne 4), mais aussi des pointeurs d'un sous-type, défini par une classe dérivée comme `Alimentation` (ligne 5). En fait, le polymorphisme n'est effectif qu'à condition que la variable soit déclarée comme un pointeur ou une référence de classe. Dans ce cas, elle peut désigner des objets appartenant à la classe elle-même ou à une de ses classes dérivées. Dans le cas contraire, une conversion implicite est effectuée. Par exemple, lors de l'affectation d'un objet de type `Alimentation` à une variable de type `Article`, seules les données membres de la « partie `Article` »

```
  class Article {
    public:
1     virtual float prixTTC() {return (_prixHT * 1.206);}
    ...
    };
  class Alimentation : public Article {
    public:
2     float prixTTC() {return (_prixHT * 1.055);}
      ...
    };
  main() {
3   Article *commandes[500];
    float totalTTC;
    ...
4   commandes[13] = new Article("Lessive aux enzymes", 19.95, 50);
5   commandes[14] = new Alimentation("Caviar", 499.95, 1000, ...);
    ...
6   for (int i = 0; i < 500; i++)
7     totalTTC += commandes[i]->prixTTC();
    }
```

FIG. 3.11: *Exemple de liaison dynamique : les objets du tableau* `commandes` *sont des instances de la classe* `Article` *ou de la classe* `Alimentation`*.*

de l'objet de type `Alimentation` sont recopiées dans l'objet de type `Article`. Cela suppose toutefois que l'opérateur d'affectation n'est pas surchargé et qu'aucun opérateur de conversion de `Alimentation` vers `Article` n'est défini !...

Le calcul du prix TTC d'un article, qui figure à la ligne 7 de l'exemple, n'est bien sûr correct qu'à condition que la liaison soit dynamique, puisque la fonction `prixTTC` est redéfinie dans la classe `Alimentation` (ligne 2). Il y a deux conditions à cela : la fonction `prixTTC` doit être déclarée *virtuelle* dans la classe de base `Article` (ligne 1) et l'objet auquel est appliquée la fonction doit être désigné par un pointeur ou une référence (ligne 3). Si l'une de ces conditions n'est pas vérifiée, la liaison est statique.

Que le mot clé `virtual` soit spécifié ou non, la redéfinition d'une fonction virtuelle est elle-même gérée comme une fonction virtuelle. Le mot clé doit seulement être spécifié pour la première définition de la fonction, dans la classe de base, `Article` dans l'exemple.

Comme en EIFFEL, une classe abstraite C++ contient une ou plusieurs fonctions dont les implantations ne peuvent pas être décrites dans la classe parce qu'elles varient selon les classes dérivées. Une classe abstraite ne peut évidemment pas être instanciée et ne peut pas servir de spécificateur de type, sauf en tant que pointeur ou référence, liaison dynamique oblige.

L'exemple de la figure 3.12 reprend la classe `Temps` de la figure 3.8. Deux classes en dérivent désormais, `TempsC` et `TempsMSC`, qui correspondent à deux implantations différentes d'un temps, en minutes, secondes et centièmes, d'une part, en centièmes seulement, d'autre part. La définition de la fonction `print()`, qui imprime la valeur d'un temps, ne peut pas être donnée dans la classe `Temps`, puisqu'elle dépend justement de l'implantation. La signature de la fonction est suivie

```
#include <iostream.h>

class Temps {
  public:
    Temps() {}
    virtual void print() const = 0;
    virtual Temps* plusCst(const int c) = 0;
    virtual Temps* plus(const Temps* const tps) = 0;  // Erreur
    ...
  };
class TempsC : public Temps {
  private:
    int _cts;
  public:
    TempsC() {}
    TempsC(int cts) : _cts(cts) {}
    void print() const;
    Temps* plusCst(const int c);
    Temps* plus(const Temps* const tps);
    ...
  };
class TempsMSC : public Temps {
  private:
    int _mn, _sec, _cts;
  public:
    TempsMSC() {};
    TempsMSC(int mn, int sec, int cts)
      : _mn(mn), _sec(sec), _cts(cts) {}
    void print() const;
    ...
  };

void TempsC::print() const {
  cout << "Centiemes = " << _cts << endl;
  }
void TempsMSC::print() const {
  cout << "Minutes = " << _mn << " secondes = " << _sec
       << " centiemes = " << _cts << endl;
  }

Temps* TempsC::plusCst(const int c) {
  return new TempsC(_cts + c);
  }

Temps* TempsC::plus(const Temps* const tps) {
  return new TempsC(_cts + tps->_cts);
        // Erreur : 'tps' est de type 'Temps'.
  }
... ...
... ...
```

FIG. 3.12: *La classe abstraite* `Temps` *et deux de ses classes dérivées,* `TempsC` *et* `TempsMSC`*, correspondant à deux implantations différentes d'un temps.*

d'une initialisation à zéro (ligne 1) et la fonction est dite *virtuelle pure*. La classe `Temps` devient, de ce fait, abstraite : les implantations de la fonction doivent être données dans les classes dérivées (lignes 4–6 et 7–10).

La définition effective d'une fonction virtuelle pure doit réaliser une adéquation *parfaite* avec la signature donnée dans la classe de base. Si l'adéquation n'est pas parfaite, la définition effective ne fait que surcharger le nom de la fonction. Il n'y a pas de problème avec la fonction `print()` (ligne 1), qui ne retourne pas de résultat et n'admet aucun paramètre. Il n'y en a pas non plus avec la fonction `plusCst`, qui réalise l'addition d'un temps et d'un entier (ligne 2) : elle peut être définie avec le même profil (ligne 11), par exemple pour la classe `TempsC`, puisque le type effectif du résultat, un pointeur de type `TempsC` (ligne 12), est un sous-type du type déclaré, un pointeur de type `Temps`[9].

En revanche, la fonction `plus`, qui effectue l'addition de deux temps, ne peut être déclarée virtuelle pure (ligne 3). En effet, la réalisation de l'addition dans la définition effective de la classe `TempsC` (lignes 13–15) nécessite l'accès à la donnée membre `_cts` du temps passé en paramètre, `tps`. Comme ce dernier est déclaré de type `Temps`, l'instruction d'accès est rejetée par le compilateur (ligne 14). Le problème est le même avec les données membres `_mn`, `_sec` et `_cts` pour la fonction de la classe `TempsMSC`.

### 3.5.3 L'héritage multiple

En C++ comme en EIFFEL, l'héritage est en fait multiple. Une classe peut donc dériver de plusieurs classes de base, l'ordre d'énumération des classes de base déterminant l'ordre d'appel des constructeurs des classes de base lors de l'exécution d'un constructeur de la classe dérivée (l'ordre est inversé pour les destructeurs).

Les conflits d'héritage sont réglés grâce à l'opérateur de résolution de portée. Si plusieurs classes de base possèdent un membre de même nom, la classe dérivée les hérite tous. Cependant, chaque fois que ce nom est utilisé dans la classe dérivée, il faut indiquer explicitement, en utilisant l'opérateur de résolution de portée, la classe d'origine du membre concerné.

Même s'il n'existe pas de classe jouant le rôle de racine implicite du graphe d'héritage[10], une classe peut parfois hériter plusieurs fois, directement ou indirectement, d'une même classe, comme le montre la figure 3.13 (en EIFFEL, l'héritage est alors dit répété). Par défaut, la structure d'un objet de type `Vin` est obtenue en assemblant la structure d'un objet de type `Alimentation` et celle d'un objet de type `Liquide` avec la structure propre à un objet de type `Vin`. Les données membres de la classe `Article` existent donc en deux exemplaires dans la structure résultante : celles qui proviennent de la structure de l'objet de type `Alimentation` et celles qui proviennent de la structure de l'objet de type `Liquide`.

En revanche, si la classe `Article` est déclarée classe de base *virtuelle* dans les définitions des deux classes dérivées `Alimentation` et `Liquide` (Fig. 3.14, lignes

9. L'usage d'un pointeur est impératif : déclarer le résultat de type `Temps` provoquerait une erreur de compilation puisque, étant abstraite, la classe `Temps` ne peut être instanciée.

10. Comme la classe `ANY`, en EIFFEL.

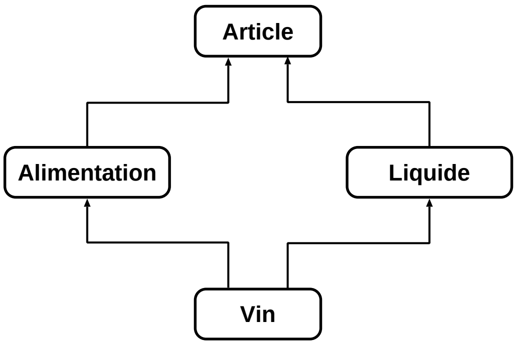



FIG. 3.13: *Exemple d'héritage répété : la classe `Vin` hérite deux fois de la classe `Article`, par la classe `Alimentation` et par la classe `Liquide`.*

4 et 8), la structure des objets de la classe `Vin` ne contient alors plus qu'un seul exemplaire des données membres héritées de la classe de base virtuelle, `Article`.

Les constructeurs des classes de base virtuelles sont invoqués *avant* les constructeurs des classes de base non virtuelles, tout en respectant l'ordre de la liste de dérivation. Si l'une des classes ne possède pas de constructeur, c'est son constructeur par défaut qui est invoqué. La classe `Vin` de notre exemple n'a qu'une seule classe de base virtuelle, `Article`. Puisque cette classe possède un constructeur explicite (lignes 1–2), les données membres correspondantes doivent être initialisées dans la liste d'initialisations (ligne 13). La classe `Vin` possède aussi deux classes de base non virtuelles, `Alimentation` et `Liquide` (ligne 12). La liste d'initialisations devrait par conséquent invoquer leurs constructeurs (lignes 5–6 et 9–10) et s'écrire :

```
Vin(char *des, int q, float prix, char *date, char *cru) :
  Article(des, q, prix),
  Alimentation(des, q, prix, date),
  Liquide(des, q, prix, 0.75),
  _cru(cru) {}
```

Cependant, ces constructeurs font double emploi avec celui de la classe `Article`. Pour éviter la redondance, il est préférable de recourir aux constructeurs par défaut (ligne 13) et, donc, doter les classes `Alimentation` et `Liquide` de constructeurs par défaut *vides* (lignes 7 et 11). Il faut faire de même pour la classe `Article` (ligne 3), car un constructeur par défaut invoque implicitement les constructeurs par défaut de ses classes de base. Les données membres héritées des classes `Alimentation` et `Liquide` sont alors explicitement initialisées dans le corps du constructeur (lignes 14–15).

### 3.5.4 Comparaison avec Eiffel

**L'héritage**

Le mécanisme d'héritage de C++ se distingue principalement de celui d'EIFFEL par la façon dont est géré le statut des membres (primitives en EIFFEL) héritées. En

```
     class Article {
       protected:
         char *_designation;
         int _quantite;
         float _prixHT;
       public:
1        Article(char *des, int q, float prix)
2          : _designation(des), _quantite(q), _prixHT(prix) {}
3        Article() {}
         ...
       };
4    class Alimentation : virtual public Article {
       protected:
         char *_date;  // Date limite de consommation
       public:
5        Alimentation(char *des, int q, float prix, char *date)
6          : Article(des, q, prix), _date(date) {}
7        Alimentation() {}
         ...
       };
8    class Liquide : virtual public Article {
       protected:
         float _volume;  // Volume d'une bouteille (en litres)
       public:
9        Liquide(char *des, int q, float prix, float vol)
10         : Article(des, q, prix), _volume(vol) {}
11       Liquide() {}
         ...
       };
12   class Vin : public Alimentation, public Liquide {
       protected:
         char *_cru;
       public:
         Vin(char *des, int q, float prix, char *date, char *cru)
13         : Article(des, q, prix), _cru(cru) {
14             _date = date;
15             _volume = 0.75;
           }
         ...
       };
```

FIG. 3.14: *La classe* `Vin` *dérive des classes* `Alimentation` *et* `Liquide`.

C++, qu'une dérivation soit déclarée privée, publique ou protégée, elle ne permet, de toute façon, que de conserver ou de renforcer la protection des membres protégés et publics hérités de la classe de base. Les membres publics de la classe de base peuvent devenir protégés dans la classe dérivée, mais il n'existe aucun moyen de rendre public ou protégé un membre initialement privé, ni de rendre public un membre initialement protégé. EIFFEL, en revanche, n'impose aucune restriction de cette sorte, puisque l'option **export** de la clause d'héritage permet de modifier à volonté le statut d'une primitive héritée. En outre, toutes les primitives privées d'une classe EIFFEL sont accessibles aux objets des classes dérivées (sous-classes) de cette classe, alors que les membres privés de la classe de base restent inaccessibles aux objets des classes dérivées. Le statut privé des primitives EIFFEL n'est donc pas équivalent à celui conféré par le mot clé `private` en C++.

D'une manière générale, les mécanismes d'héritage d'EIFFEL sont plus simples dans leur conception et, surtout, dans leur utilisation. Il n'y a nul besoin de déclarer de classes virtuelles en cas d'héritage répété, car les attributs (c'est-à-dire les données membres) des classes responsables d'héritage répété ne sont pas implicitement dupliqués. Quant aux problèmes de collisions de noms, ils sont réglés grâce à des options de la clause d'héritage (**rename** et **select**), qui portent sur le texte global de la classe dérivée. Elles s'appliquent, par conséquent, aux corps de toutes les routines de la classe.

### La liaison dynamique

La différence majeure entre C++ et EIFFEL tient à la liaison, qui est statique par défaut en C++. Elle n'est dynamique qu'à deux conditions : la fonction doit être déclarée virtuelle dans la classe de base et la variable désignant l'objet auquel est appliquée la fonction doit être un pointeur ou une référence. Rien de tel en EIFFEL : la liaison est toujours dynamique.

Incidemment, si une variable de type **expanded** `Article` en EIFFEL correspond à une variable de type `Article` en C++, elle ne peut toutefois recevoir qu'une valeur de même type : il est interdit d'affecter le contenu d'une variable de type **expanded** `Alimentation` à une variable de type **expanded** `Article`. En C++, comme nous l'avons mentionné, l'affectation est autorisée et donne lieu à une conversion implicite, qui consiste à ne recopier que la « partie `Article` » de l'objet de type `Alimentation`.

### La redéfinition de fonctions

En C++, la signature de la redéfinition d'une fonction doit réaliser une adéquation parfaite avec celle de la fonction redéfinie. Dans le cas contraire, ce n'est pas une redéfinition mais une surcharge. Il n'en est rien en EIFFEL, où la redéfinition correspond à l'idée qu'une classe peut fournir une version plus spécialisée d'un service décrit dans un de ses ascendants. La redéfinition d'une routine doit effectivement posséder le même nombre d'arguments que la routine redéfinie et ne peut être qu'une fonction si la routine redéfinie est une fonction ou une procédure si la routine redéfinie est une procédure. En revanche, le type de chacun des arguments, ainsi que le type du résultat dans le cas d'une fonction, peut être changé selon la règle dite de

```
  class POLYGONE
  feature {ANY}
1   intersection(p: POLYGONE): POLYGONE is ...
    -- Intersection du polygône courant avec le polygône ‘p’
      do
        ...
      end; -- intersection
    ...
  end -- class POLYGONE
```

```
  class RECTANGLE
  inherit
    POLYGONE
2     redefine intersection
    end;
  feature {ANY}
    longueur, largeur: REAL;
3   intersection(r: RECTANGLE): POLYGONE is ...
    -- Intersection du rectangle courant avec le rectangle ‘r’
      do
        ...
      end; -- intersection
    ...
  end -- class RECTANGLE
```

FIG. 3.15: *Un exemple de rédéfinition de fonction avec covariance en Eiffel : le type du paramètre de la fonction* `intersection` *de la classe* `RECTANGLE` *est conforme au type du paramètre de la définition initiale de la classe* `POLYGONE`*.*

*covariance* : le nouveau type doit être *conforme* au type d'origine, c'est-à-dire, dans le cas simple de types non génériques, doit être un sous-type du type d'origine [11].

Considérons, par exemple, la classe `POLYGONE`, dont l'interface offre une fonction calculant l'intersection du polygône courant avec un polygône quelconque (Fig. 3.15, ligne 1). Comme l'algorithme calculant l'intersection de deux rectangles est plus efficace que celui calculant l'intersection de deux polygônes quelconques, la fonction est redéfinie dans la classe `RECTANGLE`, sous-classe de `POLYGONE` (ligne 2). La redéfinition respecte le principe de covariance : le type du paramètre formel, `RECTANGLE` (ligne 3), est conforme au type du paramètre de la définition initiale, `POLYGONE`.

En C++, il n'y aurait pas de liaison dynamique dans ce cas. Ainsi, dans l'instruction :

```
p->intersection(q);
```

où `p` serait déclaré de type `Polygone*`, ce serait toujours la fonction de la classe `Polygone` qui serait appelée, quels que soient les types (`Polygone` ou `Rectangle`) du pointeur contenu effectivement dans `p` et du paramètre effectif `q`.

11. Pour l'énoncé complet et détaillé des règles de conformité de types, voir le chapitre 13 de [Meyer, 1994a].

**Les classes abstraites**

En termes de conception, une classe abstraite permet de procéder incrémentalement à la description des objets, tout en garantissant un maximum de modularité. Un concept est d'abord décrit en tant qu'abstraction générale, par une classe abstraite, sans décider d'une implantation particulière. L'héritage permet ensuite de réutiliser cette description abstraite pour décrire les implantations appropriées aux diverses spécialisations du concept. Idéalement, la définition d'une classe abstraite doit se rapprocher de la spécification abstraite et complète du concept correspondant. Incidemment, une telle classe abstraite sert alors de documentation au programmeur pour définir les différentes implantations du concept.

Tout est fait en ce sens en EIFFEL. En C++, malheureusement, pour les raisons évoquées dans les trois sous-paragraphes précédents, les règles de l'héritage et de la surcharge limitent fortement l'intérêt des fonctions virtuelles pures et des classes abstraites.

## 3.6 La généricité

### 3.6.1 Les patrons de classes et de fonctions

La généricité peut être définie comme la possibilité de paramétrer un module par un ou plusieurs types, le terme de module étant pris dans son sens le plus général. C++ autorise une certaine forme de généricité, grâce aux *patrons*[12] de classes. La figure 3.16 montre un extrait de la déclaration d'un patron de classe qui décrit une liste triée et qui n'admet qu'un seul paramètre, `Type` (ligne 1). Celui-ci peut apparaître dans le texte de la classe, partout où l'usage d'un nom de type est licite, par exemple dans la déclaration d'une donnée membre (ligne 2).

Le nom de la classe (incluant la liste des paramètres formels) sert ensuite de spécificateur de type, comme un nom de classe ordinaire, notamment pour définir les fonctions membres (ligne 4). Ces dernières sont, de ce fait même, paramétrées par le type `Type` et doivent donc être déclarées comme patrons de fonctions (ligne 3).

Un type spécifique est obtenu à partir d'un patron de classe en donnant des valeurs effectives aux paramètres formels : une nouvelle classe est instanciée, en substituant dans le patron chaque occurrence de chaque paramètre formel par le type effectif correspondant. Par exemple, la déclaration de la ligne 6 génère une classe décrivant une liste d'entiers, dont est créée une instance, d'une capacité de 500 éléments. De même, la déclaration de la ligne 7 génère une classe décrivant une liste d'objets de type `Article`, dont est créée une instance, d'une capacité de 100 éléments cette fois. Les instructions des lignes 8 et 9 invoquent la fonction membre `inserer`, pour stocker un élément de type approprié dans chacune des deux piles.

12. Traduction du terme anglo-saxon *template*.

```
      #include <iostream.h>
1     template <class Type>
        class ListeTriee {
          private:
2           Type *_liste;   // Tableau implantant la liste
            int _max;       // Capacite maximale de la liste
            int _dernier;   // Indice du dernier element
          public:
            ListeTriee(int max);
            ~ListeTriee();
            void inserer(Type el);
            ...
          }
      ...
3     template <class Type>
4       void ListeTriee<Type>::inserer(Type el) {
          if (_dernier == _max)
            cout << "*** Erreur : liste pleine\n";
          else {
            // Algorithme rudimentaire d'insertion dans une liste triee
            int i = 0, j;
5           while ((i <= _dernier) && (_liste[i] <= el))
              i++;
            for (j = _dernier; j >= i; j--)
              _liste[j+1] = _liste[j];
            _liste[i] = el;
            _dernier++;
            }
          }
      ...
      main() {
6       ListeTriee<int> li(500);
7       ListeTriee<Article> la(100);
        Article art("Caramels mous", 4.95, 100);
        ...
8       li.inserer(4);
9       la.inserer(art);
        ...
        }
```

FIG. 3.16: *Extraits de la déclaration du patron de classe* `ListeTriee` *et des définitions des patrons de ses fonctions membres.*

### 3.6.2 Comparaison avec Eiffel

Comme en EIFFEL, un patron de classe peut dériver d'autres classes, génériques ou non, et il est possible de composer des types. Ainsi, le patron de classe ListeTriee peut être instancié en :

ListeTriee< Pile<Article> >[13]

en supposant que Pile est aussi une classe générique.

Toutefois, à la différence d'EIFFEL, deux classes instanciées à partir d'un même patron ne sont pas liées par la relation d'héritage lorsque leurs types paramètres effectifs le sont. Par exemple, bien que la classe Vetement dérive de la classe Article, ListeTriee<Vetement> n'est pas un sous-type de ListeTriee <Article>. Le polymorphisme de variable ne s'applique donc nullement aux objets de ces deux types.

Il n'existe pas de facilité syntaxique permettant d'exprimer des contraintes sur les paramètres d'un patron de classe, pour réaliser l'équivalent de la généricité contrainte telle qu'elle existe en EIFFEL. Rien n'empêche pour autant d'appliquer une fonction ou un opérateur à des objets des types paramètres, comme le montre la figure 3.16 : l'opérateur <= est utilisé à la ligne 5 de la définition de la fonction membre inserer pour comparer deux objets du type paramètre Type. L'appel de la fonction inserer à la ligne 8 est correct, puisque l'opérateur <= est bien défini pour deux opérandes de type entier. En revanche, l'appel de la ligne 9 est incorrect si l'opérateur <= n'est pas défini pour des opérandes de type Article. Le manuel de référence de C++ [Stroustrup, 1992] ne précise malheureusement pas quand ce genre d'erreur doit être détectée. Il envisage même que les erreurs relatives à l'utilisation des patrons ne soient signalées qu'à l'édition de liens.

L'absence de consignes précises pour la mise en œuvre de la généricité, lorsqu'elle a été introduite dans le langage, et la relative complexité qu'elle a progressivement atteinte ont retardé son intégration à beaucoup de compilateurs. La généricité n'est encore que partiellement implantée dans un bon nombre d'entre eux, la façon de procéder pour l'utiliser variant même souvent d'un compilateur à l'autre. Comme le reconnaît Bjarne Stroustrup lui-même [Stroustrup, 1994], la possibilité de définir et d'utiliser des patrons pour éviter la duplication de code, sans que le programmeur doive explicitement générer les classes et les fonctions qui lui sont nécessaires, est incompatible avec la notion de compilation indépendante des diverses parties d'une application. Au contraire, le compilateur doit connaître les patrons de classes et de fonctions instanciés lors des compilations de parties séparées.

## 3.7 Les exceptions

### 3.7.1 Les gestionnaires d'exceptions

Comme beaucoup d'autres langages, notamment EIFFEL, mais aussi COMMON LISP [Steele Jr., 1990], CLU [Liskov et Guttag, 1990] et ADA [Barnes, 1995], C++

13. Attention à la syntaxe : les deux symboles > doivent impérativement être séparés par une espace pour ne pas être confondus avec l'opérateur de décalage à droite !

fournit un mécanisme de gestion d'exceptions, permettant de déclencher puis traiter des événements correspondant à des anomalies d'exécution, telles qu'une division par zéro, un dépassement des bornes d'un tableau, etc.

Les exceptions de C++ sont largement inspirés par celles de COMMON LISP. Un bloc `try` regroupe des instructions susceptibles de produire des exceptions de certains types. Il est suivi d'un ensemble de clauses `catch`, qui définissent les gestionnaires des types d'exceptions qui doivent être *attrapées* — c'est-à-dire interceptées — au cours de l'exécution des instructions du bloc `try`. Une exception de type donné est *lancée*, — c'est-à-dire déclenchée — par une expression `throw`. Elle est attrapée par le gestionnaire de type approprié, s'il existe, qui prend alors le contrôle de l'exécution du programme. La recherche du gestionnaire à exécuter est effectuée en parcourant séquentiellement les gestionnaires associés au bloc `try`. Elle s'arrête avec le premier gestionnaire réalisant une adéquation avec l'expression `throw`, que cette adéquation soit parfaite ou qu'elle requière une conversion standard.

Ces principes sont illustrés par un exemple simple reprenant la classe qui décrit une pile d'entiers, `PileInt` (cf. Fig. 3.7). Le paramètre `max` du constructeur de la classe (Fig. 3.17, ligne 7) doit être strictement positif, puisqu'il représente la capacité maximale de la pile initialisée. La violation de cette contrainte est signalée en lançant une exception de type `TailleNegative` (ligne 8). Les objets de ce type (ligne 1) ont une donnée membre privée, `_taille` (ligne 2), qui sert à stocker la valeur responsable de l'erreur et qui est initialisée grâce au constructeur (ligne 3), au moment du lancement de l'exception (ligne 8). Cette donnée peut être consultée avec la fonction membre `taille()` (ligne 4).

De la même façon, la fonction `sommet()` de la classe `PileInt` lance une exception de type `PileVide` lorsque la pile courante est vide (ligne 9). Cette fois, il n'y a besoin de mémoriser aucune information et la classe `PileVide` ne possède donc aucun membre (lignes 5–6).

Si une exception de type `TailleNegative` est lancée au cours de l'exécution des instructions d'un bloc `try` (Fig. 3.18, lignes 1–8), les instructions du gestionnaire correspondant (lignes 9–11) sont exécutées. L'objet de type `TailleNegative` construit au moment du lancement de l'exception est transmis au gestionnaire [14], qui consulte le membre `_taille` grâce à la fonction `taille()`, afin d'afficher la valeur ayant provoqué l'exception (ligne 10). La fonction `exit()` [15] est ensuite invoquée pour arrêter l'exécution du programme (ligne 11).

Lorsque l'exception est de type `PileVide`, le corps du gestionnaire correspondant (lignes 12–14) est exécuté. Comme l'objet construit lors du lancement de l'exception n'est porteur d'autre information que son type, il n'est pas nécessaire d'y accéder lors de l'exécution du gestionnaire. L'argument du gestionnaire n'est donc pas nommé et seul son type (`PileVide`) est spécifié. Puisque le corps du gestionnaire ne contient aucune instruction d'arrêt, l'exécution du programme se poursuit en séquence, *derrière le bloc* `try` (ligne 15), et non pas derrière le point d'où a été lancée l'exception.

14. Comme l'argument du gestionnaire est une référence, seule l'adresse de l'objet est transmise. En l'absence du symbole `&`, c'est une copie qui serait transmise.

15. Cette fonction appartient à la bibliothèque standard.

```
class PileInt {
  private:
    int *_tab;      // Tableau implantant la pile
    int _max;       // Capacite maximale de la pile
    int _iSommet;   // Indice du sommet de la pile
  public:
    PileInt(int max);
    int sommet();
    int estVide();
    ...
  };

1 class TailleNegative {
  private:
2   int _taille;
  public:
3   TailleNegative(int t) : _taille(t) {}
4   int taille() {return _taille;}
  };

5 class PileVide {
6 };

// Les fonctions membres de la classe PileInt

7 PileInt::PileInt(int max) {
  if (max <= 0)
8   throw TailleNegative(max);
  else {
    _iSommet = -1;
    _max = max;
    _tab = new int[max];
    }
  }

int PileInt::sommet() {
  if (estVide())
9   throw PileVide();
  else
    return _tab[_iSommet];
  }
...
```

FIG. 3.17: *Le constructeur lance une exception de type `TailleNegative` si son paramètre est négatif ou nul (ligne 8), tandis que la fonction membre `sommet()` lance une exception de type `PileVide` si la pile courante est vide (ligne 9).*

```
...
try {
  int s, taille = 1000;
  PileInt *pile;
  ...
  pile = new PileInt(taille);
  s = pile->sommet();
  ...
  }
catch(TailleNegative &anomalie) {
  cout << "*** Taille negative : " << anomalie.taille();
  exit(-1);  // Termine l'execution.
  }
catch(PileVide&) {
  cout << "*** Consultation d'une pile vide";
  }
...
```

FIG. 3.18: *Un exemple de bloc `try`, avec des gestionnaires d'exceptions de types `TailleNegative` et `PileVide`.*

Un gestionnaire d'exception ressemble donc à un sous-programme. L'argument du gestionnaire joue le rôle de paramètre formel et permet de spécifier le type des exceptions attrapées par le gestionnaire. Une expression `throw` joue le rôle d'un appel de sous-programme, avec transmission d'un paramètre effectif.

### 3.7.2 Comparaison avec Eiffel

En EIFFEL, les contraintes sur les paramètres d'une routine et sur l'état d'un objet auquel est appliquée une routine sont exprimées de manière formelle, par des expressions logiques, dans les assertions de la routine (la précondition et la postcondition) et de la classe de la routine (l'invariant de classe). Pendant l'exécution, le viol d'une assertion déclenche automatiquement une exception, sans que le programmeur ait à écrire de tests explicites pour cela. Une anomalie d'origine logicielle (par exemple, une division par zéro) ou matérielle (par exemple, une insuffisance en ressource mémoire) déclenche également une exception. Toute exception peut être attrapée en spécifiant une clause **rescue** et l'exécution de la routine responsable de l'exception peut être reprise grâce à l'instruction **retry**.

En C++, le mécanisme de gestion des exceptions est plus limité. Il ne permet notamment pas la prise en compte des signaux externes émis par le matériel ou le système d'exploitation (interruption clavier, signal d'horloge, etc.). Les événements de ce type doivent être traités à l'aide des primitives fournies par le système d'exploitation. Les gestionnaires d'exceptions sont exclusivement conçus pour le traitement des erreurs détectées au cours de l'exécution par des tests explicitement prévus par le programmeur. Ils ne fournissent pas non plus de mécanisme simple permettant de reprendre l'exécution d'un bloc `try` responsable d'une exception à l'endroit où l'exception a été lancée.

## 3.8 Conclusion

Même si C++ et EIFFEL sont tous deux des langages de classes fortement typés, ils diffèrent par de nombreux aspects, car les principes et les objectifs qui ont motivé leurs concepteurs sont radicalement différents.

EIFFEL apparaît comme un langage neuf. Inspiré par les travaux sur les types abstraits de données et les techniques de spécification et de preuve de programmes, il est spécialement conçu pour le développement de gros logiciels en équipe. C'est un langage homogène, qui s'inscrit directement et pleinement dans la technologie objet. La classe, en tant qu'implantation d'un type abstrait de données, est le concept central et le seul moyen de structuration des données : une application forme un système de classes, sans fonctions ni variables globales, comme il en existe en C++. EIFFEL encourage ainsi une approche rigoureuse de la programmation, reposant sur l'abstraction de données. Des assertions (préconditions et postconditions des routines, invariant de classe) permettent d'établir la spécification d'une classe, qui est comparable à celle d'un type abstrait de données et qui sert ensuite de guide pour définir l'implantation physique de la classe. D'autres assertions (invariants et variants d'itérations, instructions **check**), insérées dans le code des routines, indiquent les principales étapes de la preuve de la correction des routines [Gautier *et al.*, 1996, §§ 3.4 et 8.5]. Même si elles ne permettent pas toujours de faire une démonstration formelle complète, la présence d'assertions constituent un gage de qualité. Comme elles sont « exécutables », elles servent au concepteur pour la mise au point et la maintenance du logiciel. Comme elles permettent d'exprimer la sémantique des opérations d'une classe, elles définissent, en quelque sorte, un manuel d'utilisation pour le client de la classe.

D'un point de vue moins conceptuel, EIFFEL ne sacrifie jamais le confort de programmation à l'efficacité :

- La syntaxe est claire et lisible.
- Un objet est implicitement représenté par une référence, ce qui dispense le programmeur de manipuler explicitement des pointeurs.
- Un ramasse-miettes récupère automatiquement l'espace libéré par les objets obsolètes. La notion de destructeur est donc inutile.
- La liaison est toujours dynamique [16]. Le programmeur n'a pas le souci de prévoir la déclaration de fonctions virtuelles.
- Le programmeur n'a pas non plus le souci de prévoir la déclaration de classes virtuelles car, par défaut, l'héritage répété ne provoque pas la duplication des données héritées.

C++, au contraire, reste prisonnier de sa filiation avec C. Il a été conçu dès l'origine comme une extension de C afin d'y inclure, entre autres, la définition de classes, mais sans réel souci d'uniformité. A la différence d'EIFFEL, C++ est construit en deux couches, le langage C d'une part, et les classes d'autre part. L'utilisation de C++ n'implique donc pas l'utilisation d'une technologie objet et ne propose

16. Seul l'appel d'une routine déclarée « gelée » (**frozen**), qui ne peut être redéfinie dans une sous-classe, est résolu statiquement.

d'ailleurs aucune démarche de programmation particulière. Les choix de conception ont été principalement dictés par des critères d'efficacité et s'avèrent radicalement opposés à ceux qui ont été effectués pour EIFFEL :

- Par souci de compatibilité, la syntaxe des constructions de C a été conservée, même lorsque leur lisibilité est douteuse. L'esprit de cette syntaxe a même été retenu pour certaines constructions nouvelles : l'utilisation de symboles, dont certains sont déjà employés dans les constructions syntaxiques de C avec une sémantique différente, a été préférée à celle de mots clés. Le code source perd en lisibilité ce qu'il gagne en concision.
- Un objet n'est pas implicitement représenté par son adresse, ce qui astreint les programmeurs à manipuler explicitement des pointeurs.
- La liaison est statique par défaut, le programmeur devant explicitement prévoir la résolution des appels par liaison dynamique *au moment de la définition d'une fonction*, en déclarant la fonction virtuelle. Ce choix semble d'autant plus contestable que l'opérateur de résolution de portée permet d'imposer la liaison statique quand elle est jugée nécessaire, *au moment de chaque appel de fonction*.
- Comme il n'existe pas de ramasse-miettes, il faut définir explicitement des destructeurs pour récupérer l'espace libéré par les objets obsolètes. Ce point est une source importante d'erreurs, souvent pernicieuses, car il arrive fréquemment que, par le jeu des pointeurs, différents objets partagent des structures de données. Libérer l'espace occupé par une structure présente alors le risque de détruire des parties d'objets qui sont encore utilisés par ailleurs. En outre, les destructeurs de la hiérarchie d'héritage d'un objet étant invoqués en cascade en remontant la hiérarchie, l'écriture des destructeurs peut de ce fait devenir très délicate, particulièrement en cas d'héritage multiple. Il ne faut pas non plus oublier de déclarer les destructeurs avec la spécification `virtual`, afin qu'ils soient bien tous invoqués lorsque la liaison est dynamique...
- Le programmeur doit prévoir la déclaration de classes virtuelles, pour désactiver la duplication des données membres en cas d'héritage répété, qui est le défaut.

C++ est donc plutôt une boîte à outils dans laquelle le programmeur choisit l'instrument qui lui convient le mieux. Cette boîte à outils a l'avantage d'être assez complète : elle autorise la déclaration de constantes et de références, la surcharge des opérateurs et des fonctions, la définition de classes, l'héritage multiple, la généricité et la gestion d'exceptions. La liste n'est pas close, puisque le comité de normalisation Ansi/Iso se réunit chaque année pour faire évoluer la norme.

Cependant, cet avantage est également un inconvénient : la multiplicité des outils disponibles encourage les programmeurs à ignorer les principes fondamentaux de la programmation (par objets) pour écrire au plus vite un code voulu compact et efficace. En outre, les outils disponibles ne sont ni faciles à comprendre ni faciles à utiliser, surtout lorsqu'ils sont combinés entre eux, comme l'ont montré, en particulier, les exemples des paragraphes 3.5.2 et 3.5.3. Comprendre une instruction aussi élémentaire que :

```
x = f(a,b);
```

peut même devenir un véritable casse-tête, puisque l'opérateur = et l'identificateur `f` peuvent tous deux être surchargés, sans compter l'existence possible d'opérations de conversions applicables à `a` et `b`. L'algorithme déterminant quelle fonction (ou quel opérateur) est effectivement invoquée est relativement simple à mettre en œuvre... par une machine. Le nombre de cas à examiner est déjà en lui-même une source d'erreurs pour le programmeur. Par conséquent, acquérir une maîtrise parfaite du langage apparaît bien difficile, si ce n'est impossible.

Il n'existe pas, en C++, de constructions syntaxiques permettant d'exprimer des assertions à la manière d'EIFFEL. Il est quand même possible de définir une fonction évaluant une expression et lançant une exception si cette expression est fausse [17], les directives de compilation conditionnelles permettant d'ignorer les appels de cette fonction, une fois le débogage terminé. Cette technique n'est toutefois qu'un pis-aller. Elle alourdit l'écriture du code et en diminue la lisibilité. Les appels nécessaires à la définition d'un équivalent aux préconditions, postconditions et invariants de classe d'EIFFEL doivent être explicitement prévus par le programmeur. Ils ne sont d'aucun secours à titre de documentation, car ils sont dispersés dans le texte des corps des diverses fonctions membres et ne peuvent donc pas apparaître dans l'interface des classes.

Le fait que C++ soit compatible avec C, lui ouvrant ainsi un marché florissant, est sans doute le principal facteur de son succès. L'opportunité de réutiliser facilement des programmes C existants et l'espoir de convertir rapidement à la programmation par objets les développeurs habitués à la programmation en C sont des arguments séduisants. Ils s'avèrent cependant bien souvent illusoires : la parenté avec C n'encourage pas les programmeurs formés avec C (ou avec un langage apparenté) à utiliser au mieux les outils de programmation par objets. Il est plus facile et plus rapide pour eux, coûts et délais de réalisation obligent, de continuer à programmer en C avec une vague « sauce » objet. Ceci est particulièrement flagrant dans les petites et moyennes entreprises, où la formation des programmeurs se fait la plupart du temps sur le tas. En outre, la complexité des mécanismes disponibles et leur caractère bas niveau rend C++ mal adapté au développement de logiciels en équipe, à moins d'adopter des normes draconiennes. L'usage intensif de l'héritage multiple, de la surcharge et des effets de bord de toutes natures font perdre rapidement au code produit tout caractère d'extensibilité et de réutilisabilité, mais aussi de portabilité.

Pour ces raisons mêmes, C++ n'apparaît pas non plus adapté à l'enseignement de la programmation par objets. Il encourage un style d'écriture cryptique hérité de C et le recours à des astuces de programmation sous prétexte d'efficacité, toutes choses qui n'incitent guère les étudiants à faire preuve de rigueur.

Malgré tout, compte tenu des contraintes et des objectifs fixés à l'origine, C++ peut être considéré comme une réussite de l'ingénierie des langages. Il fait cohabiter sans (trop) d'incohérences apparentes des notions empruntées à divers langages et choisies pour avoir fait leurs preuves. Sa compatibilité avec C et son efficacité en font certainement un langage idéal pour générer du code pour un langage de plus haut niveau... comme EIFFEL, par exemple !

17. Le patron d'une fonction nommée `assert()` est généralement défini à cet effet en bibliothèque.

# 4

# Méthodes d'analyse et de conception par objets

LE GÉNIE LOGICIEL ÉTUDIE les méthodes de développement de systèmes informatiques complexes. L'apparition des langages de programmation par objets dans les années 80 a motivé le développement des méthodes d'analyse et de conception par objets pour prendre en compte ces langages dans les travaux du génie logiciel. L'objectif de ce chapitre est de présenter les méthodes d'analyse et de conception par objets en articulant notre présentation autour de la méthode OMT. Des points de comparaison avec d'autres méthodes (OOD, FUSION, BON, OBJECTORY) seront également donnés. Nous nous attacherons à montrer comment les méthodes d'analyse et de conception par objets tentent de concilier les besoins de représentation de haut niveau et les capacités des langages de programmation par objets. La réutilisation fera l'objet d'une brève présentation autour des *design patterns*, des *frameworks* et des composants logiciels. L'état des évolutions en cours et de la normalisation dans le domaine des méthodes d'analyse et de conception par objets sera également abordé.

## 4.1 Introduction

Les différentes techniques objet n'ont pas été créées *ex nihilo* : les langages de programmation par objets sont pour certains dérivés de langages de programmation existants (C++ de C – voir *chapitre 3* –, CLOS de COMMON LISP, etc.) et bénéficient également de l'évolution de ceux-ci ; les bases de données à objets offrent des fonctionnalités (concurrence d'accès, transactions, etc.) héritées de leurs devancières relationnelles ou hiérarchiques et utilisent des traits d'implémentation (cache, architecture client/serveur, etc.) classiques (voir *chapitre 5*) ; enfin, les représentations par objets trouvent une partie de leurs racines dans les techniques de représentation des connaissances du domaine de l'intelligence artificielle (voir *chapitre 10*). Il en va de même pour les méthodes d'analyse et de conception par objets : elles sont en partie issues des méthodes de génie logiciel existant depuis plus de deux décennies dans ce domaine, auxquelles elles ont ajouté des traits spécifiques aux techniques objet.

C'est durant les années 1970 qu'ont été réalisés les premiers travaux dans le domaine du génie logiciel, dont un des objectifs principaux était de proposer des techniques pour maîtriser la réalisation de logiciels. En effet, l'évolution rapide du matériel informatique rendait possible le développement de logiciels de plus en plus importants, dont la complexité croissante a été une motivation pour l'élaboration de méthodes de développement [Sommerville, 1985]. Une proposition importante a été le découpage du développement et de l'utilisation d'un logiciel en étapes successives : le *cycle de vie* [Boehm, 1976]. Aujourd'hui, une des manières couramment acceptée de découper le cycle de vie d'un logiciel est la suivante : *définition des besoins*, *analyse*, *conception*, *codage*, tests, utilisation et maintenance. Dans ce chapitre, nous ne nous intéresserons qu'aux quatre premières phases du cycle de vie :

- La définition des besoins a pour objectif de répondre aux questions suivantes : quel est le domaine précis de l'application ? Quelles sont les fonctionnalités qui doivent être rendues disponibles ? Quel est le contexte dans lequel l'application sera utilisée ? etc. ;
- La phase d'analyse doit organiser et décomposer les informations issues de la phase précédente. Il s'agit de comprendre et de modéliser le système à réaliser ; les contraintes informatiques ne doivent en principe pas être prises en compte ;
- La phase de conception sert à déterminer précisément quels seront les éléments à implémenter pour réaliser l'application en prenant en compte des contraintes de réalisation (par exemple, utilisation d'une base de données, répartition des traitements sur plusieurs processeurs, etc.) ;
- La phase de codage [1] constitue la réalisation de l'application dans un langage de programmation.

Les différentes phases de ce cycle de vie ont été l'objet de recherches pour proposer des techniques permettant, sinon de les automatiser, du moins de les rendre plus rigoureuses et plus facilement contrôlables. En particulier, pour toutes les étapes qui précèdent le codage, des méthodes dites *structurées* ont été proposées à la fin des années 70 [Schoman et Ross, 1977 ; Yourdon, 1975].

Les méthodes d'analyse et de conception par objets sont apparues à la fin des années 80 pour deux raisons principales. D'une part, les langages à objets commençaient à être relativement répandus [2], et les méthodes qui existaient à cette époque n'étaient pas adaptées aux concepts, tels que la spécialisation, mis en œuvre dans les langages à objets ; la conception notamment était très proche des langages structurés. Il s'agissait donc d'une motivation de génie logiciel issue d'un souci de mieux intégrer les phases de conception et de codage. D'autre part, le concept même d'objet a été perçu comme pouvant permettre d'améliorer la phase d'analyse, dans la mesure où il offrait une représentation plus naturelle des entités du monde réel [Carré *et al.*, 1995].

1. Nous emploierons indifféremment codage, implémentation et programmation.
2. Certains de ces langages avaient quelques années d'existence : SMALLTALK (1980), C++ et OBJECTIVE-C (1983), FLAVORS (1981) et LOOPS (1981/1983), par exemple.

Dans les méthodes d'analyse et de conception par objets, la phase de définition des besoins n'est pas toujours identifiée et on doit souvent s'appuyer sur d'autres méthodologies[3]. Du fait de l'utilisation des mêmes concepts pour les phases d'analyse et de conception, la continuité entre ces phases est *a priori* mieux assurée qu'avec les méthodes structurées par exemple. Si cela peut constituer un avantage (meilleur suivi des entités, retour de la conception à l'analyse plus aisé, etc.), cette relative confusion entre l'analyse et la conception peut aussi amener à faire intervenir durant l'analyse des éléments qui relèvent uniquement de la conception, voire du codage (par exemple, savoir si telle fonctionnalité sera réalisée grâce à une méthode ou à une association).

De façon très schématique, une méthode d'analyse et de conception par objets doit produire un ensemble de *modèles* selon un *processus* défini. Les modèles représentent les différentes vues que l'on peut avoir du *système* en cours de réalisation, avec une ou plusieurs *notations* adaptées (principalement des *diagrammes* et des *formulaires*). Ces modèles constituent le résultat de l'activité d'analyse et de conception. Suivant les cas, un modèle de même type peut être utilisé pour toutes les phases (le modèle à objets de la méthode OMT, par exemple), ou des modèles ou notations spécifiques peuvent apparaître pour certaines phases (diagrammes d'interaction des objets de la phase de conception de la méthode FUSION, par exemple). Le processus décrit la démarche à adopter pour produire les différents modèles. Par ailleurs, un outil est la plupart du temps indispensable pour réaliser ces opérations, et nombre de méthodes n'ont pas survécu à cause de l'absence d'outils évolués et fiables.

Certains auteurs [Pronk et Tercero, 1994] ont répertorié plusieurs dizaines de méthodes d'analyse et de conception par objets, mais la plupart ont été éphémères et n'ont eu aucun outil développé pour les mettre en œuvre. Si l'on s'en tient aux méthodes ayant atteint une certaine maturité et stabilité, et ayant été mises en œuvre dans des outils à diffusion non confidentielle, les méthodes suivantes peuvent être retenues :

- OOA de S. Shlaer et S.J. Mellor [Shlaer et Mellor, 1988 ; Shlaer et Mellor, 1992] ;
- OOAD de P. Coad et E. Yourdon [Coad et Yourdon, 1991a ; Coad et Yourdon, 1991b] ;
- OMT de J. Rumbaugh [Rumbaugh *et al.*, 1991] ;
- OBJECTORY et OOSE de I. Jacobson et al. [Jacobson *et al.*, 1992] ;
- FUSION de D. Coleman et al. [Coleman *et al.*, 1996] ;
- OOD de G. Booch [Booch, 1994a] ;
- BON de K. Waldén et J.-M. Nerson [Waldén et Nerson, 1995] ;
- CLASSE-RELATION de P. Desfray [Desfray, 1996].

3. Il existe des propositions pour définir les besoins de façon structurée, par exemple avec [ODP, 1995], ou dans certaines méthodes d'analyse et de conception par objets, comme la méthode BON. Voir également 4.3.2.

La littérature sur les méthodes d'analyse et de conception par objets est importante. [Henderson-Sellers et Edwards, 1990] propose une discussion sur la notion de cycle de vie dans les méthodes d'analyse et de conception par objets. [Champeaux et Faure, 1992] font un comparatif de plusieurs méthodes dont OOA et OMT. [Monarchi et Puhr, 1992] fait une étude critique des méthodes d'analyse et de conception par objets et propose des axes de recherche. [Graham, 1994] et [Morris *et al.*, 1996] sont des ouvrages généraux sur le développement de logiciels avec les techniques objet. De nombreux articles sont parus dans le JOOP (*Journal of Object-Oriented Programming*) notamment sous la plume des auteurs des principales méthodes. D'autres articles sont également cités tout au long de ce chapitre.

Nous utiliserons la méthode OMT pour notre description car c'est la méthode la plus employée[4], elle a été à l'origine de beaucoup de discussions sur les méthodes d'analyse et de conception par objets, elle a été mise en œuvre dans une grande quantité d'outils et ... c'est celle que nous utilisons. Nous nous appuierons sur [Rumbaugh *et al.*, 1991] pour les caractéristiques de la méthode OMT. Les extensions à cette méthode apporté par J. Rumbaugh dans des articles du JOOP entre 1993 et 1995, appelées « OMT2 », ne sont pas prises en compte dans ce chapitre. La plupart de ces extensions sont présentes dans UML (*Unified Modelling Language* en cours d'élaboration (voir 4.5.1). Nous ferons cependant appel à d'autres méthodes (en particulier aux méthodes BON, FUSION et OOD) lorsque qu'il nous semble que certaines de leurs fonctionnalités sont intéressantes à présenter, qu'elles soient absentes ou traitées de façon différente dans OMT. Une grande partie des différences entre les méthodes d'analyse et de conception par objets sont en fait superficielles et le travail sur UML (voir 4.5.1) a pour objectif de proposer un méta-modèle commun.

Ce chapitre est organisé de la manière suivante. La section 4.2 décrit les modèles produits lors de l'utilisation de la méthode OMT en comparant certains aspects de ces modèles avec ceux d'autres méthodes. La section 4.3 présente le processus de la méthode OMT, et, plus rapidement, celui de quelques autres méthodes. La section 4.4 propose une discussion sur la réutilisation, qui est à notre sens un des problèmes importants pour lequel les méthodes d'analyse et de conception par objets devraient proposer des solutions. Enfin, nous évoquerons les perspectives dans le cadre de l'émergence d'UML et de la normalisation.

## 4.2 Les modèles et leurs notations

L'objectif de chacune des phases d'analyse et de conception est de produire un ensemble de *modèles*. Chaque modèle permet de représenter tout ou partie des différentes vues que l'on peut avoir sur le système à réaliser :

- la *vue structurelle* qui décrit l'ensemble des entités utilisées dans le système ;
- la *vue comportementale* qui décrit les modifications du système à la réception d'événements externes ou internes ;

4. Fin 1995, près de la moitié des utilisateurs d'une méthode d'analyse et de conception par objets utilisaient OMT (source : IDC, marché nord-américain).

- la *vue architecturale* qui décompose le système en sous-systèmes afin d'en faciliter la compréhension.

Chacun des modèles est élaboré en utilisant une ou plusieurs notations constituées de formulaires et de diagrammes. Ces notations favorisent la communication entre les différents intervenants, soit au sein d'un outil, soit par l'intermédiaire de copie sur papier. Si ces notations sont mises en œuvre dans un outil informatique, les modèles obtenus peuvent être l'objet de vérifications automatiques et servir de base à la production du code de l'application et de la documentation. Nous décrivons ci-dessous les notations des modèles proposés par la méthode OMT.

Dans la méthode OMT, chacune des vues est représentée par un modèle différent : le *modèle à objets* pour la vue structurelle, le *modèle dynamique* pour la vue comportementale et le *modèle fonctionnel* pour la vue architecturale.

Dans la suite, nous utiliserons le terme *méta-modèle* pour désigner l'ensemble des concepts permettant de produire un certain type de modèles. En particulier, nous appellerons *modèle objet* l'ensemble des concepts communs aux différentes techniques objet : les *objets* regroupés en *classes* possédant des *propriétés* (*attributs* ou *méthodes*), les classes étant hiérarchisées par une relation de spécialisation permettant un *héritage* des propriétés [Carré *et al.*, 1995] (voir *chapitres 1 et 12*).

### 4.2.1 Le modèle à objets en OMT

Pour construire des modèles à objets, OMT propose le modèle objet enrichi du concept d'association, issu du méta-modèle entité-association (ER : *Entity-Relationship* [Chen, 1976]). Ce dernier concept n'existe pas dans les méta-modèles des langages à objets « purs », comme SMALLTALK ou EIFFEL (voir *chapitre 2*). En revanche, le concept d'association est présent dans des langages hybrides comme YAFOOL (voir *chapitre 14*, section 4), ainsi que dans les systèmes de représentation des connaissances par objets comme TROEPS (voir *chapitre 10*, section 2).

La notation des modèles à objets propose deux diagrammes : le *diagramme de classes* et le *diagramme d'instances*.

Le diagramme de classes décrit les classes, c'est-à-dire leurs attributs et leurs *opérations* (les méthodes), ainsi que leurs relations (spécialisation, agrégation, association). Le diagramme d'instances montre comment des objets sont reliés afin de décrire une utilisation typique ou des cas particuliers. Ces diagrammes sont utilisés pour montrer l'existence de classes, d'objets et de relations dans un système. À chaque classe peuvent être associés des diagrammes d'états (dynamique interne) et des diagrammes de trace d'événements (dynamique externe) (voir 4.2.2). Les deux composants essentiels de la notation sont les classes et leurs relations. La figure 4.1 montre quelques exemples de notations qui sont commentés ci-dessous ; les notations complètes peuvent être consultées dans [Rumbaugh *et al.*, 1991].

#### Les classes et les instances

Une classe est représentée par un rectangle composé de trois parties : le nom de la classe, ses attributs, ses opérations (voir par exemple les classes `Faisceau` ou

`Reseau` de la figure 4.1). Une instance est représentée par un rectangle aux coins arrondis comportant le nom de la classe de l'instance, et la liste des attributs valués (voir figure 4.2).

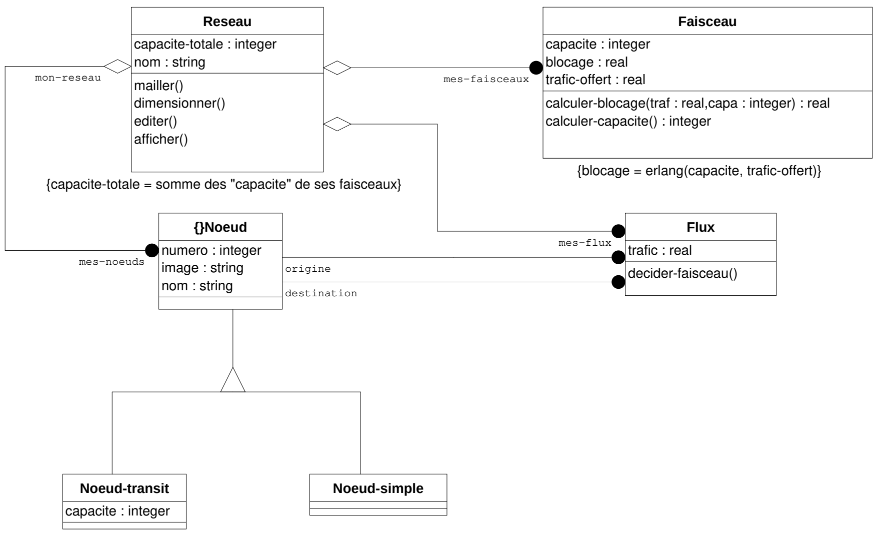



FIG. 4.1: *Un exemple de diagramme de classes* OMT

Une classe porte un nom et possède éventuellement des attributs et opérations :

- les attributs de classe ont un nom et un type, voir par exemple l'attribut `capacite` de la classe `Faisceau`) ;
- les opérations de classe ont un nom et une signature (des paramètres typés en entrée, et un type de résultat en sortie), voir par exemple la méthode `calculer-blocage` de la classe `Faisceau`).

On voit donc que les opérations et les attributs sont traités pratiquement uniformément comme des *propriétés* dans le modèle à objets.

La figure 4.2 est un exemple de diagramme d'instances. Il montre comment sont liées les instances constituant un réseau. L'instance de la classe `Flux` est liée à deux instances de la classe `Noeud-simple` qui ont les rôles `origine` et `destination`. L'instance de la classe `Reseau` est liée aux deux instances précédentes de la classe `Noeud-simple`, et à une instance de la classe `Noeud-transit` ; ces trois instances ont le rôle `mes-noeuds` et dans chacune de ces instances, l'instance de la classe `Reseau` a pour rôle `mon-reseau`. Ainsi, les diagrammes d'instances ne peuvent guère servir qu'à représenter de petits exemples d'utilisation du modèle à objets.

### Les relations

Les relations entre les classes peuvent être de type association, agrégation (ou composition) ou spécialisation. L'association traduit un lien sémantique entre deux

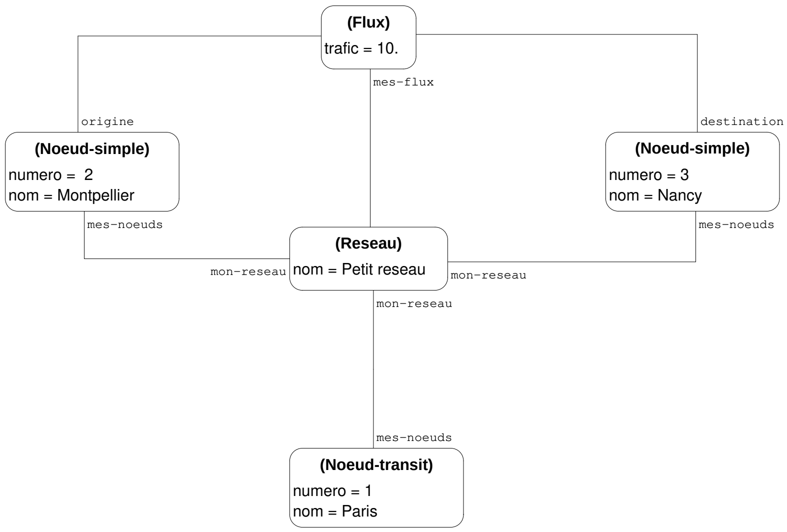



FIG. 4.2: *Un exemple de diagramme d'instances* OMT

classes, alors que l'agrégation traduit en plus une relation partie/tout ou de composition [Magnan, 1994]. Ce type de relation peut comporter des *rôles* joués par les classes dans la relation. La *cardinalité* des rôles peut être exactement un (1), illimitée (n), zéro ou un (0,1), zéro ou plus [0..n], un au moins [1..n], rang spécifique [3..7] et rang spécifique et nombres exacts ([1..3], 7). Les relations sont représentées par des traits entre les classes et portent optionnellement un nom ; une notation graphique permet, dans certains cas, de spécifier la cardinalité pour chaque rôle de la relation (voir ci-dessous). La relation d'agrégation est notée par un losange du côté de la classe composite et la relation de spécialisation par un triangle dont la base est orientée vers les sous-classes.

En plus de la cardinalité, une relation de type association ou agrégation peut être enrichie des notions de *qualificatif* ou *clé* prise par un attribut dans la relation et de *contrainte* conditionnant la relation (ou la classe). Une association peut également avoir des propriétés et peut être modélisée comme une classe, c'est-à-dire posséder des attributs, des opérations ainsi que des relations avec d'autres classes. OMT permet de déclarer une classe *abstraite*, c'est-à-dire une classe qui ne pourra pas être instanciée, et dont l'utilité est de déclarer des attributs ou opérations communes à ses sous-classes (voir *chapitre 3*, section 5, *chapitre 2*, *chapitre 1*) ; dans notre exemple la classe `Noeud` est abstraite (préfixée par {} dans l'outil utilisé ici). De plus, OMT permet de déclarer *dérivée* une classe, une relation, ou un attribut d'une classe. Cette notion de dérivation exprime une idée de « calcul » ou de « contrainte » sur l'existence des objets, liens ou valeur d'attribut. Cette notion est différente de la notion de dérivation en C++ où elle correspond au concept de spécialisation (voir *chapitre 3*, section 5).

Dans notre exemple de la figure 4.1, on indique qu'une instance de la classe `Flux` est reliée à deux instances de la classe `Noeud` par l'intermédiaire de deux associations, et qu'une instance joue le rôle d'`origine` et l'autre le rôle de `destination`. Le rond noir à l'extrémité droite de ces associations indique qu'une instance de la classe `Noeud` peut être l'origine ou la destination de 0 ou plus instances de la classe `Flux`. Les relations entre la classe `Reseau` et les classes `Faisceau` et `Noeud` sont des relations d'agrégation (ou de composition). Comme pour les asociations, une cardinalité et un rôle peuvent être indiqués au moyen des mêmes notations. Enfin, les dernières relations du diagramme montrent que les classes `Noeud-transit` et `Noeud-simple` spécialisent la classe `Noeud`.

Il faut noter que si les cardinalités des associations peuvent être pertinentes en phase d'analyse, il n'est pas toujours facile d'utiliser ces définitions durant les phases de conception et de codage pour effectuer des contrôles de cardinalités. Par exemple, pour une association dont la cardinalité est 1 (voir les associations de rôles `origine` et `destination` entre les classes `Flux` et `Noeud`), il est clair que immédiatement après la création d'un instance de la classe `Flux` les cardinalités des deux associations avec la classe `Noeud` seront nulles. Dans ce cas, il serait uniquement possible de contraindre l'association à ne pas dépasser une cardinalité maximale égale à 1.

Une solution possible dans les langages possédant des *constructeurs*[5] serait de forcer la création des instances par un constructeur ayant au moins comme paramètres les instances à associer à l'instance à créer, le constructeur assurant la mise en relation. Par exemple, la classe `Flux` devrait avoir un constructeur ayant au moins comme paramètres les deux instances de la classe `Noeud` origine et destination de l'instance de la classe `Flux` à créer. Pour forcer l'utilisation du bon constructeur, on peut, au niveau de la programmation, interdire l'utilisation d'autres constructeurs (en levant une exception, par exemple – voir *chapitre 3*, section 7), ou, dans un outil de génération de code, générer toujours l'appel au bon constructeur.

### Les modules

À un niveau d'abstraction plus élevé, OMT propose le concept de *module*, afin de regrouper les classes. Un module matérialise une perspective ou un point de vue sur une situation. Un modèle à objets peut être découpé en plusieurs modules. À l'intérieur d'un module, les noms de classes et d'associations doivent être uniques, mais une classe peut être référencée dans plusieurs modules (ce qui permet de lier les modules entre eux). Cette notion de module est équivalente à la notion de *catégorie* dans OOD de Booch [Booch, 1994a], mais en OMT il n'y a pas de déclaration spécifique des relations entre modules, bien qu'il soit précisé dans [Rumbaugh *et al.*, 1991] qu'il doit y avoir moins de liens entre des modules qu'à l'intérieur des modules. La notion de module n'a pas de représentation graphique officielle.

La méthode BON propose un concept équivalent aux modules OMT : les *clusters*. Ces entités, comme les classes, peuvent être décrites par des formulaires textuels et par une notation graphique spécifique. La notation textuelle permet de décrire

5. Il s'agit de fonctions de création d'instances associées aux classes et ayant des paramètres permettant d'initialiser les attributs, comme en JAVA ou C++ (voir *chapitre 3*).

ces entités en termes de définition des besoins ou d'analyse, alors que la notation graphique, plus précise, est plus adaptée à la conception.

Le concept de module est similaire, du point de vue structurel, aux *perspectives* des systèmes de représentations de connaissance par objets comme TROEPS ; mais il n'offre pas de sémantique ni de mécanisme d'utilisation de ces perpectives, comme les passerelles ou la multi-instanciation de TROEPS (voir *chapitre 10*, section 4).

### Les contraintes

La méthode OMT permet de déclarer des *contraintes* (relations fonctionnelles) entre des entités du modèle à objets : les objets, les classes, les attributs et les relations. Une contrainte sert à restreindre les valeurs qu'une entité peut prendre. La syntaxe de ces contraintes n'est pas définie par la méthode qui spécifie uniquement que celles-ci doivent apparaître entre accolades « près » de l'entité contrainte. Nous donnons dans la figure 4.1 deux exemples de contraintes : le premier, sur la classe `Reseau`, indique que la capacité totale d'un réseau doit être égale à la somme des capacités de ses faisceaux ; le second, sur la classe `Faisceau`, lie les attributs `blocage`, `capacite` et `trafic-offert` de cette classe (la fonction d'Erlang permet de calculer la probabilité de perte sur un faisceau en fonction de sa capacité et du trafic qu'il doit écouler).

Si l'intérêt de ces contraintes est évident pour la documentation d'une application, il l'est moins pour la production de code. En effet, dans la méthode OMT, il n'existe pas de syntaxe pour ces contraintes, et il n'est donc pas envisageable de générer automatiquement le code permettant de les définir et maintenir. La notion d'*assertion*[6] de la méthode BON permet de déclarer des propriétés que doivent vérifier les objets d'une classe à certains moments : par exemple des préconditions et des postconditions de routines, ou des invariants de classes. Les assertions sont définies avec un langage à la fois textuel et graphique comprenant des opérateurs arithmétiques et logiques. Une extension d'OMT, la méthode SYNTROPY [Cook et Daniels, 1994], offre des fonctionnalités équivalentes. Le problème de ces extensions est l'impossibilité de produire automatiquement le code correspondant sauf dans des cas relativement limités ; une possibilité est l'utilisation de systèmes de propagation de contraintes fonctionnelles sur des attributs d'objets pour implémenter certains types de contraintes. Nous utilisons un tel système pour maintenir les deux contraintes (et d'autres) de la figure 4.1 [Myers *et al.*, 1990 ; Myers *et al.*, 1992 ; Pelenc, 1994]. Malheureusement, il n'y a pas de traduction automatique entre les contraintes décrites dans le modèle à objets OMT et l'application.

Ces possibilités sont à rapprocher du domaine des *techniques* ou *méthodes formelles* [Monin, 1996]. Celles-ci cherchent à offrir un cadre formel rigoureux pour la spécification de systèmes informatiques pour pouvoir produire automatiquement le code correspondant ainsi que les programmes de tests. Ces techniques restent assez difficiles à utiliser car elles demandent de maîtriser des formalismes très complexes. Il existe des tentatives pour intégrer les formalismes de ces méthodes et les objets

6. Les assertions proviennent du langage EIFFEL (voir *chapitre 2*), qui est le langage cible privilégié de la méthode BON.

[Ruiz-Delgado *et al.*, 1995 ; Moreira et Clark, 1994], mais elles sont plutôt des apports pour la structuration des informations dans les méthodes formelles que des enrichissements de méthodes d'analyse et de conception par objets.

### 4.2.2 Le modèle dynamique en OMT

Un *modèle dynamique* est constitué de *diagrammes d'états* et de *diagrammes de trace d'événements*. Il exprime tous les aspects de contrôle décrits par des séquences d'opérations qui se réalisent sur les objets du système en réaction à des événements internes ou externes.

**Les diagrammes d'états**

Les diagrammes d'états sont utilisés pour montrer les changements d'états occasionnés par des événements pour les objets d'une certaine classe. La réunion des diagrammes d'états des classes traduit l'activité du système vu dans son ensemble. Le diagramme d'états décrit le cycle de vie des instances d'une classe (dynamique interne), c'est-à-dire comment les instances d'une classe sont créées, détruites et modifiées au cours de l'activité du système. Il montre comment des événements déclenchent des transitions d'un état vers un autre. Chaque état peut se décomposer en sous-états ; les transitions sont déclenchées par des événements sous certaines conditions ; des actions peuvent être attachées aux transitions comme aux états, et des messages peuvent être envoyés à des objets.

Un *état* correspond aux valeurs d'attributs et de relations d'un objet à un instant donné. Une *transition* est un changement d'état occasionné par un *événement*. Un événement se passe à un instant donné et n'a pas de durée ; il peut être interne au système (le début ou la fin d'une méthode, par exemple), ou externe (une action à la souris, ou un choix d'item de menu). Un *message* est l'invocation d'une opération d'un objet.

Nous reprenons notre exemple de réseau pour illustrer ce type de diagramme dans la figure 4.3. Ce diagramme décrit les différents états dans lesquels peut être une instance de la classe `Reseau`, et les événements qui déclenchent les changements d'états, durant les opérations de maillage (détermination des faisceaux existants du réseau), et de dimensionnement (calcul de la capacité des faisceaux existants).

Certains états (`initial`, `maillé`, `dimensionné`) du diagramme correspondent effectivement à des états des instances au sens où, par exemple, on peut sauvegarder l'ensemble des instances constituant le réseau (instances des classes `Reseau`, `Flux`, `Noeud` et `Faisceau`), avec toutes les valeurs d'attributs et de relations cohérentes. Cela étant, il est difficile de caractériser un tel état : par exemple pour l'état `maillé`, on peut définir cet état par le fait que le réseau possède au moins un faisceau, mais en cours de maillage le réseau sera également dans cet état. Un artifice consiste à déclarer un attribut booléen dans la classe `Reseau` (`maillé?`, par exemple), qui prendra la valeur `vrai` à la fin de la méthode `mailler`.

Les transitions d'états peuvent également déclencher des actions (voir `afficher` pour l'événement `creer`), ou être « gardées », c'est-à-dire n'avoir lieu que si une

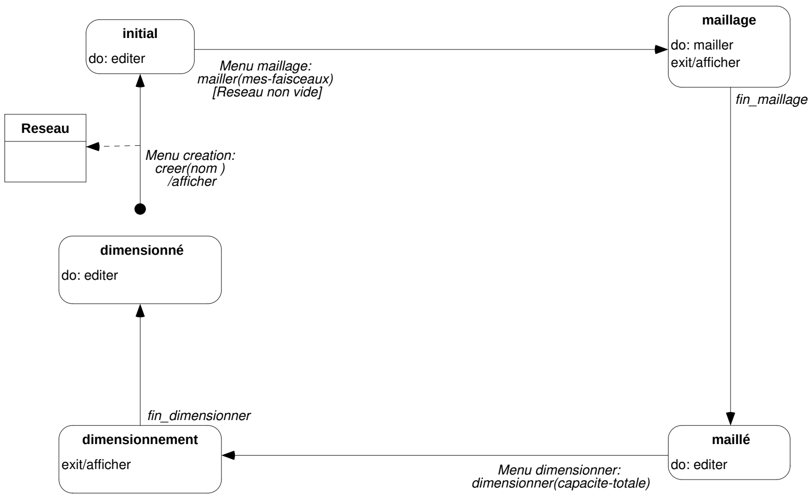



FIG. 4.3: *Un exemple de diagramme d'états* OMT

condition est satisfaite ; par exemple, le passage de l'état `initial` à l'état `maillage` ne peut être réalisé que si le réseau est non vide. Il est également possible d'indiquer des créations d'instances de classe (voir la classe `Reseau` liée à l'événement `creer`). La notation complète peut être consultée dans [Rumbaugh *et al.*, 1991].

D'autres états du diagramme correspondent au fait qu'une opération est en cours d'exécution (`maillage`, `dimensionnement`) ou au fait qu'il n'est ni maillé, ni dimensionné (`initial`). Les événements du diagramme correspondent soit à des activations, soit à des fins d'opérations. Si les activations d'opérations peuvent effectivement être des événements extérieurs (choix dans un menu, par exemple), en revanche, les fins d'opérations n'en sont jamais. Il n'est pas toujours immédiat de représenter par ce type de diagramme les changements d'états des instances durant le déroulement d'algorithmes. En effet, il n'est par exemple pas envisageable d'avoir autant d'états que de valeurs possibles pour un attribut d'objet durant le déroulement d'un algorithme. Ces diagrammes permettent donc de représenter un modèle global du comportement des classes du système à réaliser ; ils sont mieux adaptés à la description du fonctionnement de systèmes à base d'automates.

### Les diagrammes de trace d'événements

Les diagrammes de trace d'événements traduisent les échanges entre objets, mais ils s'intéressent surtout au séquencement des échanges et non à leur contenu. On peut considérer que les diagrammes de trace d'événements sont aux diagrammes d'états du modèle dynamique, ce que les diagrammes d'instances sont aux diagrammes de classes du modèle à objets. Ils sont utilisés pour tracer l'exécution d'un scénario (dynamique externe), c'est-à-dire comment les instances des classes interagissent, notamment par l'intermédiaire des activations de méthodes. Un *scénario* est une séquence d'événements qui se réalise entre des objets dans un contexte particulier du système.

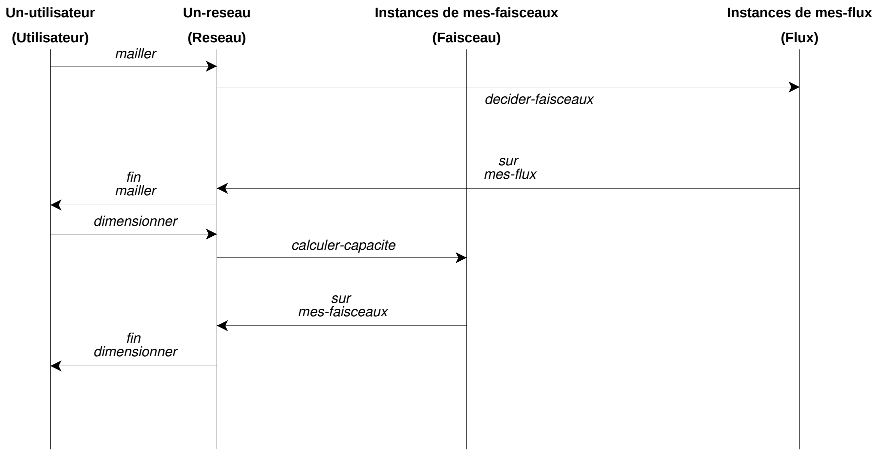



FIG. 4.4: *Un exemple de diagramme de trace d'événements* OMT

La figure 4.4 décrit les échanges qui doivent avoir lieu entre certains objets constituant un réseau pour les opérations de maillage et dimensionnement (il s'agit d'une version très simplifiée). Pour le maillage par exemple, l'utilisateur doit activer[7] l'opération `mailler` sur une instance de la classe `Reseau`, et cette instance doit activer l'opération `decider-faisceaux` sur toutes les instances de la classe `Flux` auxquelles elle est reliée par l'association `mes-flux`. La notation indiquant le parcours de la liste (sur la flèche de retour) ne fait pas partie de la méthode OMT : il n'existe pas de moyen dans les notations de la méthode pour indiquer ce type de parcours. Cette possibilité est offerte par la méthode FUSION dans les diagrammes d'interaction des objets où il est possible d'indiquer l'envoi d'un même message à une collection d'objets.

### D'autres modèles pour la vue comportementale

Pour la phase d'analyse, la méthode FUSION propose des modèles similaires à ceux de la méthode OMT pour la modélisation des comportements. Deux modèles sont regroupés sous l'appellation *modèle d'interface* : le modèle des opérations (ou opératoire), et le modèle de cycle de vie. Le premier correspond aux diagrammes d'états, et le second aux diagrammes de trace d'événements, et ils ont tous les deux une notation non graphique avec des formulaires. Pour la phase de conception, deux

7. En fait, il sélectionne l'article de menu `mailler` dont l'activation va appeler la méthode `mailler`.

notations permettent de représenter des diagrammes d'interaction des objets et des graphes de visibilité. Les premiers offrent la possibilité de représenter graphiquement les envois de messages entre objets, et les seconds la visibilité des différents objets entre eux. Les diagrammes d'interaction des objets sont comparables aux diagrammes d'instances de la méthode OMT avec une notation plus riche.

La méthode BON offre un grand nombre de formulaires pour décrire la vue comportementale d'un système : formulaires d'événements, qui décrivent les événements générés et reçus par les classes ; formulaires de scénarios, qui montrent des scénarios de fonctionnement du système, et qui sont associés à des diagrammes d'objets ; formulaires de création d'objets qui indiquent quelles classes génèrent des instances de quelles autres. La figure 4.5 donne un exemple de formulaire de création d'objets où il est indiqué que les instances des classes `Flux` et `Noeud` sont créées dans une méthode de la classe `Reseau` et que les instances de la classe `Faisceau` sont créées dans une méthode de la classe `Flux` ; les méthodes dans lesquelles les instances sont créées (méthode d'initialisation de la classe `Reseau`, méthode de maillage de la classe `Flux`) ne sont pas indiquées dans le formulaire. Les diagrammes d'objets (diagrammes dynamiques) de la méthode BON regroupe les diagrammes d'instances et les diagrammes de trace d'événements de la méthode OMT. Des scénarios sont explicitement associés à ces diagrammes.

| **CREATION** | *DIMENSIONNEMENT* | **Partie** : 1/1 |
|---|---|---|
| **COMMENTAIRE**<br>*Liste des classes créant des objets* | | **INDEXATION**<br>**créé le** 17-06-1997 MD |
| *RESEAU* | *FLUX, NOEUD* | |
| *FLUX* | *FAISCEAU* | |

FIG. 4.5: *Un exemple de formulaire de la méthode* BON

La méthode OOD comporte également des diagrammes d'états et des diagrammes d'objets qui peuvent indiquer des envois de messages entre objets.

La méthode OOSE propose un type de diagramme particulier, les *use cases* (cas d'utilisation) qui sont des diagrammes d'objets où des acteurs extérieurs au système peuvent également être représentés. Ce type de diagramme est utile en phase d'analyse pour appréhender des comportements des objets du système ; il peut également être utilisé lors de la définition des besoins en faisant intervenir des utilisateurs comme des acteurs. Le méta-modèle des *use cases* fait partie de UML (voir 4.5.1).

### 4.2.3 Le modèle fonctionnel en OMT

Un *modèle fonctionnel* se compose de diagrammes de flux de données (DFD, [Yourdon, 1975]). Il décrit l'activité à l'intérieur d'un système. Il montre les dépendances qui existent entre les données et les fonctions, sans se préoccuper du séquencement temporel de ces fonctions. Le diagramme de flux de données peut être utilisé à différents niveaux de la modélisation. En analyse, le diagramme de flux réalise la décomposition fonctionnelle du système étudié pour montrer les fonctions de haut niveau, alors qu'en conception il traduit les opérations des classes du modèle à objets et les actions des états du modèle dynamique qui sont complexes et peuvent être décomposées en sous-actions. Comme dans une modélisation de type SA (*Structured Analysis*, [Schoman et Ross, 1977]), les diagrammes de flux utilisent quatre concepts : *acteur* (objet actif), *traitement*, *réservoir* (objet passif) et *flux* (contrôle, donnée, objet).

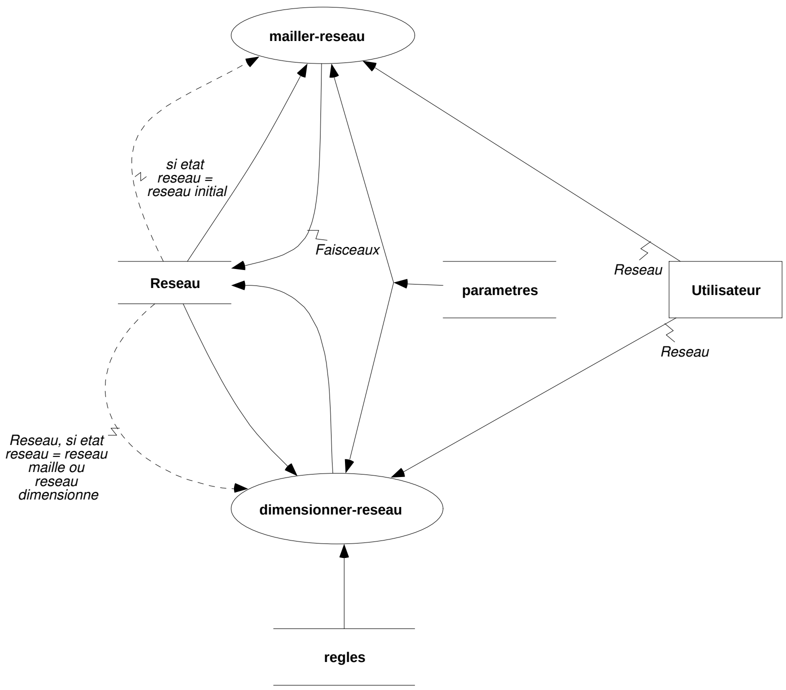



FIG. 4.6: *Un exemple de diagramme fonctionnel* OMT

La notation utilisée pour représenter les diagrammes de flux est directement inspirée des DFD de la méthode SA. Un traitement (noté par une ellipse) transforme des données. Un flux de données (noté par un trait plein muni d'une flèche, creuse si des données sont créées) est la connexion entre la sortie d'un objet ou d'un traitement, et l'entrée d'un autre. Un flux de contrôle (noté par un trait pointillé muni d'une flèche) est une valeur booléenne qui indique sous quelles conditions un traitement est activé. Un acteur (noté par un rectangle) est un objet actif qui dirige un diagramme de flux de données en produisant ou consommant les données. Un enregistrement ou réservoir de données (noté entre deux barres horizontales) est un objet passif qui enregistre des données pour un accès ultérieur. Un traitement peut être décomposé en sous-traitements, et les traitements feuilles peuvent correspondre à une méthode d'une classe du modèle à objets. La notation permet également la représentation de flux simples et composites, ainsi que la duplication et la décomposition de flux de données.

La figure 4.6 est un diagramme fonctionnel décrivant, de façon simplifiée, les opérations de maillage et de dimensionnement déjà mentionnées. Les réservoirs de données correspondent ici soit à des fichiers (`parametres`, `regles`), soit à des classes (`Reseau`). Il y a deux flux de contrôle ; par exemple, celui entre `Reseau` et `mailler-reseau` indique que le traitement `mailler-reseau` ne peut être activé que si l'instance de la classe `Reseau` est dans l'état `Reseau-initial`. La flèche creuse de `mailler-reseau` vers `Reseau` avec l'indication `Faisceaux` indique que des instances de la classe `Faisceau` sont créées par le traitement `mailler-reseau`. Enfin, on trouve l'acteur `Utilisateur` (comme sur le diagramme de trace d'événements de la figure 4.4).

Un modèle fonctionnel peut être utile lorsque l'on désire avoir une vision justement fonctionnelle d'un système : quelles sont les fonctions principales du système, et comment elles se décomposent. Dans une activité de rétro-conception [8] d'un système réalisé avec des techniques non objet, un modèle fonctionnel permet de comprendre le système plus facilement, et de lier les classes candidates à des traitements détectés lors de la définition du modèle fonctionnel.

Il peut être souvent nécessaire de représenter, sur un même modèle à objets, plusieurs ensembles de traitements qui agissent chacun de façon indépendante sur les objets. Cela peut être considéré comme l'existence de plusieurs modèles fonctionnels pour un même modèle à objets. Ce besoin de mettre en évidence plusieurs « ensemble de fonctions » sur un même ensemble d'objets a été le sujet de quelques travaux. [Harito Shteto, 1997] propose un modèle pour l'implémentation d'un ensemble d'algorithmes sur un même ensemble de classes ; les algorithmes sont réifiés et décomposés en plusieurs niveaux jusqu'à un dernier niveau qui correspond à des méthodes des classes du modèle à objets. [Vanwormhoudt *et al.*, 1997] propose de regrouper les propriétés des objets suivant les différents aspects fonctionnels du système.

### 4.2.4 Liens entre les modèles

Comme on l'a vu dans les trois sections ci-dessus, de nombreux liens peuvent exister entre le modèle à objets, le modèle dynamique et le modèle fonctionnel. En fait, on peut considérer que l'utilité des modèles dynamique et fonctionnel est de permettre de détecter de nouvelles classes et de nouvelles associations à inclure dans le modèle à objets. Par exemple, le diagramme de trace d'événements de la figure 4.4 met en évidence les associations `mes-flux` et `mes-faisceaux`. Comme on le verra par la suite (voir 4.3.1), le diagramme de classes du modèle à objets est le seul qui soit actuellement exploité pour générer le code de l'application. La démarche est donc différente de celle utilisée en représentation des connaissances par objets : la détermination des classes et des relations entre celles-ci est en grande partie guidée par les besoins fonctionnels et dynamiques du système à réaliser.

8. La rétro-conception consiste à définir un dossier de conception d'un système à partir du système déjà existant.

## 4.3 Le processus d'une méthode d'analyse et de conception par objets

Le processus d'une méthode d'analyse et de conception par objets décrit les activités nécessaires à l'obtention des différentes modèles sous forme de diagrammes, formulaires et textes, ainsi que l'enchaînement de ces activités. Autant les outils peuvent être utiles (et même nécessaires) pour l'obtention des modèles, autant, pour le processus, il est difficile d'aider l'utilisateur et d'automatiser certaines tâches. Cela réduit bien souvent la mise en œuvre du processus à l'application de quelques règles de bon sens, car sans support informatique il est très difficile de respecter tous les détails. Dans le travail sur UML, le processus n'est d'ailleurs pas pris en compte pour l'instant (voir 4.5.1)[9]. Comme pour les modèles et leurs notations, nous décrivons ci-dessous le processus de la méthode OMT.

### 4.3.1 Le processus de la méthode OMT

La méthode OMT s'intéresse aux phases d'analyse, de conception et de codage d'un système. La figure 4.7 résume le processus de cette méthode.

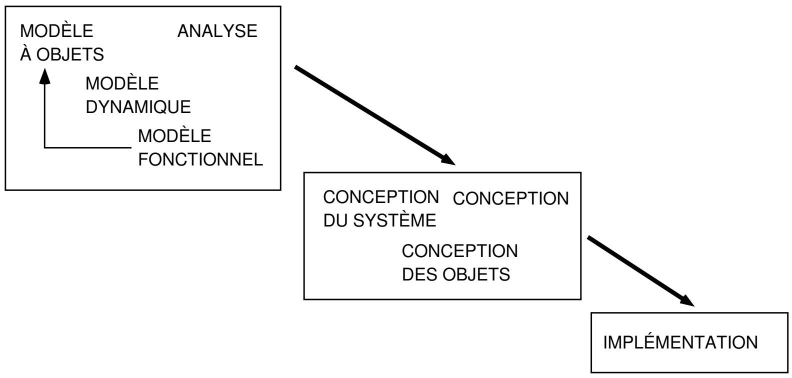



FIG. 4.7: *Le processus de la méthode* OMT

**La phase d'analyse**

La phase d'analyse consiste à décrire le système (à partir d'une définition des besoins obtenue précédemment) en produisant des modèles décrits dans la section 4.2 : le modèle à objets, le modèle dynamique et le modèle fonctionnel. La flèche de retour dans la cas ANALYSE indique que cette description peut (et doit) être améliorée jusqu'à l'obtention d'un ensemble de diagrammes satisfaisants. Pour produire chaque modèle, la méthode OMT donne une liste de tâches à réaliser et des conseils

9. Il faut d'ailleurs noter le changement d'appellation : au début des travaux sur l'unification des méthodes, UML s'appelait *méthode unifiée*.

pratiques pour y parvenir. Nous donnons ci-dessous à titre d'exemple les tâches proposées par OMT pour construire le modèle à objets :

- identifier les objets et les classes ;
- établir un dictionnaires de données ;
- identifier les relations (dont les agrégations) entre les objets ;
- identifier les attributs des objets et des relations ;
- organiser et simplifier les classes en utilisant l'héritage ;
- vérifier que toutes les données pourront être accédées correctement ;
- améliorer le modèle par itérations successives ;
- regrouper les classes dans des modules.

Il s'agit simplement de la liste de ce qui est réalisé naturellement quand on cherche à produire un modèle à objets d'une situation réelle. Pour quelqu'un qui aborde ce type d'activité, les conseils qui sont associés à ces tâches peuvent permettre de gagner un peu de temps. Par exemple, pour la première tâche il est préconisé d'éliminer les classes vagues (pour lesquelles on n'est pas capable de donner une définition claire et non ambigüe), redondantes (qui ont par exemple des attributs en commun) ou hors domaine.

**La phase de conception**

Dans la méthode OMT, la phase de conception se décompose en deux étapes : la conception du système et la conception des objets, qui doivent constituer des affinements successifs des modèles issus de la phase d'analyse, en prenant en compte des contraintes d'implémentation.

*La conception du système*

La conception du système traite de la modularité de l'application résultante : elle doit organiser les vues du système issues de la phase d'analyse, en les décomposant en sous-systèmes selon des critères fonctionnels ou physiques. Cette décomposition a également pour objectif d'obtenir une bonne modularité du système. Si cela est pertinent pour le système, c'est durant cette étape que les problèmes de concurrence et d'allocation de processeurs sont traités. Le stockage des données doit également être abordé. Il s'agit de problèmes d'architecture informatique du système, assez indépendants des modèles à objets.

*La conception des objets*

L'objectif de la conception des objets est de définir précisément les classes d'objets avec leurs attributs et méthodes. Cette définition se fait principalement à partir du modèle à objets obtenu durant la phase d'analyse, mais les modèles dynamiques et fonctionnels doivent aussi être utilisés. Elle doit prendre en compte les possibilités et les limites du langage qui sera utilisé pour l'implémentation. Par exemple, si le langage ne gère pas l'héritage multiple, et que celui-ci est utilisé dans le modèle à objets de la phase d'analyse, il faut modifier le diagramme de classes en conséquence. Le modèle à objets obtenu doit être le plus proche possible de celui qui sera implémenté dans le langage. C'est également durant cette étape que sont réalisés les optimisations de la hiérarchie des classes (par exemple, la création de classes abstraites regroupant des attributs et des méthodes abstraites communes à plusieurs classes), la conception des algorithmes implémentés dans les méthodes, les choix d'implémentation des associations (attributs, classes, etc.). La description des activités de cette étape dans la méthode OMT est du type « conseils généraux ». Par exemple, pour le choix et la mise en œuvre d'un algorithme la méthode OMT préconise de :

- choisir des algorithmes qui minimisent le coût d'implémentation des opérations ;
- utiliser des structures de données adaptées aux algorithmes ;
- définir de nouvelles classes et de nouvelles opérations si nécessaire ;
- mettre les opérations dans les bonnes classes.

C'est à partir des modèles obtenus durant cette phase, et principalement du modèle à objets, que le code sera généré par les outils proposant cette fonctionnalité.

**La phase d'implémentation**

La phase d'implémentation des méthodes d'analyse et de conception par objets consiste à produire le code à partir des modèles issues de la phase de conception. Deux aspects sont à considérer pour cette phase : les conseils donnés par les auteurs des méthodes, et l'utilisation d'outils pour produire du code automatiquement.

En ce qui concerne OMT et le premier aspect, [Rumbaugh *et al.*, 1991] donne notamment des exemples pour trois langages à objets, C++ (voir *chapitre 3*), SMALLTALK et EIFFEL (voir *chapitre 2*). Il explique comment sont réalisées les fonctionnalités suivantes dans ces trois langages : la définition des classes, la création des objets, l'appel de méthodes, l'utilisation de l'héritage, l'implémentation des associations. Les auteurs donnent également des conseils pour réaliser une implémentation dans un langage sans objets comme C, ADA, ou FORTRAN, par exemple.

Le modèle à objets peut également être utilisé pour générer un schéma de base de données. Pour les bases de données à objets (BDO), la génération peut être délicate sur des points comme la gestion des associations ou l'héritage multiple si ces fonctionnalités n'existent pas dans la base. Pour les bases de données relationnelles, il existe des manières de faire correspondre des définitions de classes et de tables relationnelles [Waldén et Nerson, 1995 ; Rumbaugh *et al.*, 1991 ; Demphlous et Lebastard, 1996].

La production automatique de code au sein d'outils[10] se réduit à la génération de squelettes de définition de classes ; dans le meilleur des cas (suivant le langage

10. appelés AGL – Atelier de Génie Logiciel, en français ; *CASE tool – Computer Aided Software Engineering*, en anglais.

cible), la traduction des associations est également réalisée. Cela s'explique par le fait que les éléments du modèle à objets (les classes, les attributs, les opérations, et, avec certains langages, les associations) sont les seuls qui existent dans les langages à objets. Par exemple, l'état d'un objet n'est pas manipulable en tant que tel dans les langages à objets. D'où la difficulté des concepteurs d'AGL pour réaliser des outils permettant de produire du code à partir d'une description issue d'une méthode.

La notion d'association n'existe pas dans les langages C++ (voir *chapitre 3*) ou EIFFEL (voir *chapitre 2*) : l'implémentation doit être faite au moyen de pointeurs, dont la gestion doit être assurée soit de façon *ad hoc* par le programmeur, soit grâce à des bibliothèques de classes spécifiques. Les AGL tentent de pallier ce manque en intégrant par exemple des bibliothèques de classes C++ offrant les fonctionnalités nécessaires. Cette approche pose deux problèmes ; d'une part, toutes les fonctionnalités ne peuvent pas être implémentées sous forme de bibliothèques externes : il faudrait parfois intervenir sur le langage lui-même (par exemple en ajoutant des mots-clés pour la définition des classes) ; d'autre part, cette approche lie le concepteur à un outil, voire à un type de machine. Certains langages offrent des fonctionnalités équivalentes aux associations (YAFOOL [Ducournau, 1991], voir *chapitre 14*), ou peuvent les offrir grâce à des extensions du langage (CLOS) [Masini *et al.*, 1989]. Les traits de représentation des méthodes d'analyse et de conception par objets sont donc plus proches de ceux des langages dits hybrides (voir [Masini *et al.*, 1989]) utilisés pour représenter des connaissances de façon plus naturelle. De plus, bien des traits de représentation sont absents des méthodes actuelles : les modalités (ou facettes) des langages de *frames*, les points de vue ou perspectives, etc. (voir *chapitre 10*, section 3).

Il y a donc une différence importante entre la représentation définie dans le cadre de la méthode, et sa traduction dans un langage de programmation. Le code produit doit être très enrichi pour obtenir un système complet, car le modèle dynamique n'est pas pris en compte lors de la génération automatique de code. L'utilisation de techniques formelles (voir 4.2.1), bien que très lourde, peut être une solution, mais la production de code directement à partir des diagrammes d'états serait certainement préférable [André, 1996]. Il est donc extrêmement difficile de maintenir une cohérence complète entre les modèles issus de la méthode, d'une part, et le code du système, d'autre part. Certains outils insèrent des commentaires particuliers dans le code produit pour retrouver les parties qu'ils ont générés, mais ces commentaires peuvent être détruits ou déplacés lors de l'édition des fichiers source. D'autres, comme OBJECTEERING qui met en œuvre la méthode CLASSE-RELATION, intègrent des fonctions d'édition de code, ce qui leur permet de mieux contrôler les modifications apportées par le programmeur.

### 4.3.2 Les processus d'autres méthodes

Le processus de la méthode FUSION est constitué des trois phases classiques : analyse, conception et implémentation. La phase d'analyse doit élaborer le modèle à objets [11] et le modèle d'interface (modèles des opérations et du cycle de vie). La

11. La traduction française de l'ouvrage de référence sur la méthode FUSION utilise en fait « modèles des objets ».

phase de conception réalise les diagrammes d'interaction des objets, et les graphes de visibilité ; elle complète également le modèle à objets par des descriptions de classe (en utilisant les informations des diagrammes d'interaction des objets et de visibilité), et des graphes d'héritage. Les auteurs donnent aussi un ensemble de conseils pour les trois phases.

La méthode BON propose un processus très détaillé en neuf tâches regroupées en trois phases :

- *collecte* : déterminer les limites du système, faire la liste des classes candidates, sélectionner des classes et les regrouper en *clusters* ;
- *description* : définir les classes, ébaucher les comportements du système, définir les interfaces ;
- *conception* : affiner les entités existantes, généraliser, compléter et réviser le système.

Des modèles (sous forme de diagrammes et de formulaires) sont créés ou modifiés par chaque tâche ; par exemple, la troisième tâche de la première phase (sélection et regroupement des classes) doit créer l'architecture statique (vue graphique des *clusters* et des classes) et le dictionnaire des classes, et doit modifier le formulaire du système (description générale du système) et les formulaires des *clusters*. Comme la méthode OMT, la méthode BON donne par ailleurs un ensemble de conseils pour trouver les « bonnes classes ».

Comme on l'a vu, les méthodes d'analyse et de conception par objets s'intéressent peu à la phase de définition des besoins. Dans le meilleur des cas (voir la méthode BON ci-dessus), elles identifient des actions dans leur processus. Pour pallier cela, des utilisations conjointes de la méthodologie d'acquisition des connaissances KADS [Schreiber *et al.*, 1993] et de la méthode OMT ont été réalisées. Dans un cas [Ayache *et al.*, 1995], la méthode KADS a été utilisée préalablement à la méthode OMT afin de définir les besoins du système ; dans l'autre cas [Siboni, 1994], les deux méthodes ont été utilisées simultanément pour des aspects différents du système. Une utilisation séquentielle est une démarche assez naturelle : acquisition des connaissances avec KADS, puis analyse et conception avec OMT ; cette démarche a de plus l'intérêt de limiter l'interface entre les deux méthodes. La deuxième approche est plus discutable car l'intégration de KADS et de OMT doit être beaucoup plus étroite : il faut établir des correspondances entre les entités internes des deux méthodes, ce qui n'est pas toujours possible selon [Siboni, 1994].

## 4.4 La réutilisation

C'est durant la phase de conception que doivent être déterminés les moyens informatiques nécessaires à la réalisation du système visé. C'est donc lors de cette phase que se pose la question de la réutilisation éventuelle d'éléments existants. Les concepts du modèle objet doivent *a priori* favoriser la réutilisation :

- la *spécialisation* est essentiellement un concept qui permet le partage d'information et de comportements ;

- la *modularité* permet de répartir les informations et les traitements dans différentes entités ;
- l'*encapsulation* donne une interface unique et bien définie aux objets.

Malgré ses qualités intrinsèques, l'utilisation du modèle objet n'implique pas de pouvoir réutiliser immédiatement et facilement. Ces dernières années plusieurs propositions tendant à faciliter la réutilisation lors de la réalisation d'applications avec les techniques objet ont été proposées. Nous décrivons brièvement ci-dessous trois types de réutilisation : les *design patterns*, les *frameworks* et les *composants logiciels*, ainsi que les travaux autour de CORBA.

### 4.4.1 Les *design patterns*

L'idée des *design patterns* [Gamma *et al.*, 1994] est de capitaliser et de documenter des solutions à des problèmes de conception qui apparaissent de façon récurrente dans les applications. Un *design pattern* est constitué d'une part d'un diagramme représentant les classes génériques qui le constitue, ainsi que les associations et les envois de messages nécessaires, et d'autre part d'un texte décrivant le *design pattern* avec un exemple et des conseils d'utilisation. UML (voir 4.5.1) propose une notation spécifique pour représenter les *design patterns*.

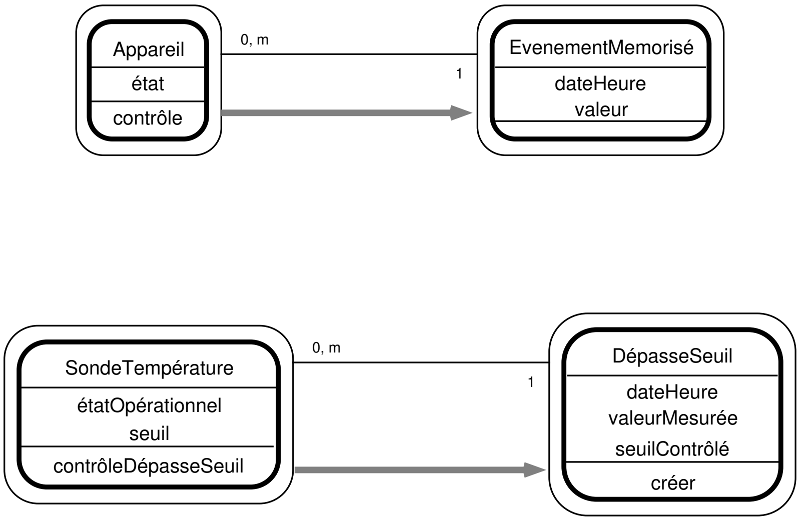



FIG. 4.8: *Un exemple de design patterns*

La figure 4.8 montre un exemple de *design pattern*. Il s'agit du problème de contrôle d'un appareil au moyen de mémorisation d'événements. Deux classes sont définies : la classe `Appareil` et la classe `EvénementMémorisé` (voir partie supérieure de la figure 4.8). Il existe une association entre ces deux classes : une instance de la classe `Appareil` peut être associée à un nombre quelconque d'instances de la classe `EvénementMémorisé`. Ce *design pattern* peut être appliqué au contrôle d'une sonde de température avec des dépassements de seuils de température. Les propriétés `état`, `contrôle` et `valeur` sont transposées dans ce cas respectivement en `étatOpérationnel`, `contrôleDépasseSeuil` et `valeurMesurée` (voir partie inférieure de la figure 4.8).

L'utilisation de ces techniques peut être utile pour la conception si on est capable de retrouver dans un catalogue la description du *design pattern* qui correspond au problème posé. Mais il peut être très difficile de retrouver le bon *design pattern*, même s'il existe, si la formulation du problème est éloignée de la définition du *design pattern*. Ensuite, si un *design pattern* a été considéré comme pertinent, il faut transposer la solution proposée dans le contexte particulier, ce qui doit être réalisé par le programmeur. Concernant la recherche d'un *design pattern* répondant à un problème particulier, l'indexation par mots-clés est une solution possible mais bien limitée. Pour la transposition d'un *design pattern* donné, une proposition consiste à enrichir le méta-modèle d'un langage à objets par des constructions permettant d'exprimer des *design patterns*. Par exemple, [Ducasse, 1997] propose d'intégrer un méta-modèle d'interaction dans un langage à objets.

### 4.4.2 Les *frameworks*

Un *framework* est constitué d'un ensemble de classes offrant des fonctionnalités pour un domaine particulier. L'intérêt des *frameworks* est de proposer des points de personnalisation (*hot spots*), tout en faisant bénéficier de fonctionnalités communes (*frozen spots*). Cette partie commune peut être réutilisée sans adaptation par le système construit dans le *framework*. Le mécanisme d'héritage peut être un moyen de spécialiser les points de personnalisation et de bénéficier des fonctionnalités communes.

CASSIS [Dao et Richard, 1990] est un exemple de tel environnement. Il définit un ensemble de classes représentant une structure générale de réseau (des graphes contenant des sommets et des arêtes, qui peuvent eux-mêmes contenir des graphes, et peuvent être partagés par plusieurs graphes), sur laquelle sont implémentées des fonctionnalités d'édition et de manipulation graphiques. Réaliser une application avec CASSIS, c'est définir ses propres classes de graphes, de sommets et d'arêtes, en personnalisant certains aspects ; il est par exemple possible de préciser quels sommets et quelles arêtes peuvent appartenir à certains graphes, de particulariser la visualisation graphique des sommets et des arêtes, etc. Les classes de la figure 4.1 (`Reseau`, `Faisceau` et `Noeud`) qui font partie d'une application qui a été réalisée avec CASSIS héritent directement des classes correspondantes de l'environnement (`Graphe-k6`, `Arete-k6` et `Noeud-k6`).

L'avantage d'un *framework* est que, si celui-ci est bien adapté à l'application à réaliser, le développement peut être très rapide. En revanche, l'utilisateur est très contraint dans la réalisation de son application avec un *framework*. De plus, la réalisation d'un *framework* est une opération coûteuse.

### 4.4.3 Les composants logiciels

Les composants logiciels sont des éléments logiciels réutilisables offrant un service particulier. Ils peuvent être des bibliothèques de classes spécialisées dans une technique particulière (graphique, structures de données, etc.) ou de classes *métier*, c'est-à-dire spécifiques d'un domaine d'activité (télécommunications, gestion bancaire, etc.). Ces composants peuvent être réutilisés directement (par héritage ou par association) en intégrant les classes nécessaires au système à réaliser, ou en créant et utilisant des instances de ces classes sur un réseau, par exemple. Dans le premier cas, il peut s'agir d'une utilisation conjointe avec les techniques de *design pattern* ou de *framework* : par exemple, CASSIS intègre un ensemble de classes implémentant son modèle de réseau. Le deuxième type d'utilisation peut être mis en œuvre en utilisant les techniques de distribution d'objets telles que CORBA.

### 4.4.4 CORBA

L'*Object Management Group* [12] est un consortium de plus de 600 compagnies informatiques et utilisatrices, dont l'objectif est de promouvoir les techniques objet pour le développement de systèmes informatiques distribués. Cette promotion se fait par la production de spécifications dont la plus importante est CORBA (*Common Object Request Broker Architecture*) [Object Management Group, 1996 ; Otte *et al.*, 1996]. Il s'agit d'une spécification permettant à des applications réparties sur un réseau informatique de communiquer selon une « syntaxe objet » : c'est-à-dire que les échanges entre ces applications se font au moyen d'envois de messages et d'accès à des attributs. Chaque application doit publier son interface (les classes, les opérations et les attributs qu'elle rend visibles) dans un langage unique (IDL : *Interface Definition Language*). L'objectif est de permettre une meilleure réutilisation des objets, puisque chaque application présente sur le réseau publie son interface, et donc les services qu'elle peut offrir, et peut utiliser les services offerts par les autres applications. Des correspondances entre des langages et l'IDL ont déjà été spécifiées et adoptées (ADA, C++, SMALLTALK), et d'autres sont en cours (COBOL, JAVA). Des services généraux au-dessus de CORBA ont également été spécifiés : service de concurrence, service de notification, service de cycle de vie (création et destruction d'objets), service de nommage, service de persistance, etc. Construits à partir de ces services des ensembles de fonctionnalités (*facilities*) sont en cours de spécification. Il s'agit soit de fonctionnalités générales (impression, méta-objets, analyse et conception – voir 4.5.2 –, etc.), soit dans des domaines applicatifs (télécommunications, finances, etc.).

CORBA est plus l'aboutissement d'un travail de normalisation qu'un réel apport technologique. Son impact peut néanmoins être important dans le domaine des télécommunications où il peut favoriser une normalisation et une meilleure interopérabilité des logiciels de gestion de réseaux.

12. L'ensemble des travaux de l'OMG peut être consulté à l'adresse : `http://www.omg.org`.

## 4.5 Évolution des méthodes d'analyse et de conception par objets

Les méthodes d'analyse et de conception par objets ont connu une phase de développement relativement courte, à peu près cinq ans, au cours de laquelle un grand nombre de propositions issues soit du domaine du génie logiciel, soit de celui de techniques objet ont vu le jour. Nous avons rapidement décrit dans les sections précédentes une grande partie de ces propositions. La tendance actuelle est la recherche d'une homogénéïsation de toutes ces propositions. Deux activités étaient en cours dans cette direction jusqu'à l'automne 1997 : le travail sur UML autour de G. Booch, I. Jacobson et J. Rumbaugh au sein de la société Rational, et la normalisation en cours à l'OMG. Ces deux activités ont fusionné en novembre 1997 avec l'acceptation de UML comme norme au sein de l'OMG.

### 4.5.1 Vers un méta-modèle unique pour la conception ?

Les travaux sur UML ont commencé à partir du constat que les méthodes les plus utilisées (OMT et OOD notamment) convergeaient sur les points principaux, et que les différences étaient souvent superficielles (sur la notation ou la terminologie entre autres). Par exemple, OOD avait repris des idées de OMT (les associations, les diagrammes de trace d'événements, etc.) et vice-versa (les sous-systèmes sont par exemple définis dans OMT2 par J. Rumbaugh). Le travail proprement dit sur UML a commencé quand en 1994 J. Rumbaugh a rejoint G. Booch au sein de la société Rational (en 1995, I. Jacobson, auteur de la méthode OOSE, s'est joint à eux). Une version préliminaire est parue à l'automne 1995, augmentée d'un addendum à la mi-96. En janvier 1997, la version 1.0 d'UML a été publiée [13] et soumise à l'OMG (voir 4.5.2).

L'objectif du travail sur UML est de proposer un standard de fait unifiant les méthodes OOD et OMT et incluant des propositions jugées intéressantes d'autres méthodes, notamment les *use cases* de la méthode OOSE et les diagrammes d'interaction d'objets de la méthode FUSION. Deux idées principales président à la spécification de cette méthode : ne spécifier que ce qui peut être implémenté dans des outils, c'est-à-dire ne pas spécifier le processus pour l'instant ; spécifier un méta-modèle permettant de définir différents modèles qui seront effectivement utiles pour un outil ou une méthode donnés. Ces modèles (éléments de base, éléments logiques, types, classes, relations entre classes, etc.) sont décrits en utilisant le modèle objet muni d'une notation sous forme de diagramme de classes. D'autres diagrammes existent : diagramme des *use cases*, diagrammes de comportements (diagramme d'états, diagramme d'activité, diagramme de séquence, diagramme de collaboration), diagrammes d'implémentation (diagramme de composants, diagramme de déploiement). Le diagramme de classes doit bien sûr servir aussi à exprimer les modèles à objets des utilisateurs de UML. Il s'agit de la mise côte à côte d'un ensemble de modèles et de diagrammes, unifiés par un méta-modèle commun. Mais les relations entre ces modèles sont encore moins claires que dans le cas de la méthode

13. Les documents sur UML peuvent être consultés à l'adresse : `http://www.rational.com`

OMT. Il est donc peu probable que des outils UML performants et complets puissent être disponibles rapidement.

### 4.5.2 Normalisation des méthodes

Au sein de l'OMG, l'« Object Analysis and Design Task Force » est chargée de spécifier un méta-modèle qui représente la sémantique des méthodes d'analyse et de conception par objets. Les modèles statiques (par exemple, modèles de classes), dynamiques (par exemple, modèle à transition d'états), architecturaux (modèles de sous-systèmes) doivent entre autres être pris en compte par ce méta-modèle. Comme les autres activités de l'OMG, cette spécification doit être réalisée grâce aux propositions reçues après l'émission d'une RFP (Request For Proposal), dont la date limite était fixée au début de l'année 1997. UML a constitué la réponse principale à cette RFP, et, après discussion interne, a été accepté au troisième trimestre 1997.

### 4.5.3 Les méta-modèles objet

UML n'est pas la seule proposition de modèle objet en cours d'élaboration ou de normalisation. L'OMG a défini un méta-modèle abstrait qui est concrétisé dans celui utilisé dans CORBA. Ce méta-modèle définit les concepts d'objet, de type d'objet, de non-objet (valeur atomique), d'opération, de sous-typage et d'héritage. On voit donc que le modèle objet de l'OMG, c'est-à-dire celui concrétisé par IDL, est plus simple que celui de UML Cette différence a deux raisons : il s'agit essentiellement du dénominateur commun minimum des modèles des langages à objets (C++ et SMALLTALK à l'origine) et les contraintes de performances de distribution sur un réseau imposent que le méta-modèle soit le plus simple possible. On a donc d'une part UML qui est un méta-modèle issu des travaux sur l'analyse et la conception, et qui est très riche et très complexe, et d'autre part le méta-modèle abstrait de l'OMG, donc la conception a été guidée par des contraintes d'implémentation, qui est beaucoup plus simple. Ces deux méta-modèles sont pourtant normalisés au sein de la même organisation !

Par ailleurs, le méta-modèle de l'ODMG [14] reprend celui de l'OMG en y ajoutant en particulier le concept d'association (voir *chapitre 5*, section 4).

Bien sûr, nous ne mentionnons ici que les méta-modèles faisant l'objet d'une normalisation, ou étant au moins définis indépendamment de tout outil ou langage informatique. D'autres méta-modèles sont présentés ou évoqués dans plusieurs chapitres de cet ouvrage : ceux des langages C++ (*chapitre 3*), EIFFEL (*chapitre 2*) et YAFOOL (*chapitre 14*), ceux des langages à prototypes (*chapitre 8*), ceux de la représentation des connaissances par objets, en particulier celui de TROEPS (*chapitre 10*), celui des logiques de descriptions (*chapitre 11*), et enfin celui des systèmes classificatoires (*chapitre 12*).

14. *Object Database Management Group* est le consortium homologue à l'OMG pour les bases de données à objets. Les travaux de l'ODMG peuvent être consultés à l'adresse : `http://www.odmg.org`.

## 4.6 Conclusion

Dans ce chapitre, nous avons présenté tout d'abord les méthodes d'analyse et de conception par objets. Ces méthodes ont une double origine : les méthodes dites « structurées » issues du génie logiciel et le modèle objet. L'introduction du modèle objet a permis d'améliorer la continuité dans la vision d'une application pour les phases de définition des besoins, d'analyse, de conception, et, dans une certaine mesure, d'implémentation.

À l'heure actuelle, les méthodes d'analyse et de conception par objets et les AGL associés sont considérés à juste titre comme des aides utiles pour le développement d'applications importantes. Les différents modèles de représentation d'un système, et notamment le modèle à objets, et les diagrammes qui leur sont associés, offrent un support appréciable de spécification et de communication. Mais la richesse et la complexité des concepts et des notations a deux inconvénients : d'une part, les utilisateurs ont beaucoup de difficultés à les appréhender en totalité, ce qui conduit bien souvent à quelques désillusions ; d'autre part, les AGL deviennent rapidement très difficiles à maîtriser, mais, paradoxalement, restent assez limités sur la qualité et la quantité de code généré automatiquement. La complexité du nouveau méta-modèle UML ne devrait pas améliorer ce défaut.

En fait, les méthodes d'analyse et de conception par objets souffrent d'un décalage important entre leur puissance d'expression et celles des langages de programmation utilisés pour l'implémentation. La génération des définitions de classes à partir des modèles à objets est assez bien réalisée, sauf pour la gestion des associations dans les langages n'offrant pas cette fonctionnalité. En revanche, la production automatique du corps des méthodes est un problème très difficile pour lequel aucune solution satisfaisante n'existe aujourd'hui. La vision fonctionnelle du système à réaliser, qui a été pratiquement complétement abandonnée dans UML, est pourtant bien utile dans certains cas. Le processus reste aujourd'hui un ensemble de conseils certes censés, mais qui ne trouvent pas de traduction dans les AGL.

Concernant la réutilisation, un *framework* offre un cadre permettant une personnalisation et une réutilisation importantes si l'application à réaliser peut accepter les contraintes imposées par le *framework* (modèle de données, fonctionnalités, etc.). Les *design patterns* n'offrent pour l'instant qu'une aide à la conception et à la documentation, leur mise en œuvre dans un langage étant réalisée au coup par coup. La normalisation entreprise par l'OMG doit tenter de concilier des contraintes d'implémentation en réseau, qui tendent à imposer des modèles simples, avec les besoins de représentation de systèmes complexes qui demandent des modèles très riches.

L'évolution des méthodes d'analyse et de conception par objets est maintenant dans une phase de stabilisation. Les concepteurs de plusieurs des principales méthodes travaillent ensemble (G. Booch pour OOD, I. Jacobson pour OBJECTORY et OOSE, et J. Rumbaugh pour OMT) afin de concevoir un méta-modèle « unifié », dont une première version a été retenu comme norme au sein de l'OMG. Ce travail a l'ambition de rassembler dans un méta-modèle la plupart des modèles proposés dans les méthodes actuelles, mais ne résoudra en rien le problème de la distance avec la programmation évoqué ci-dessus.

# 5

# Les systèmes de gestion de bases de données objet

La communauté des bases de données tend à améliorer l'expressivité des modèles utilisés ainsi que la puissance de calcul des Systèmes de Gestion de Bases de Données (SGBD) en s'appuyant sur les travaux menés dans les domaines des langages de programmation, du génie logiciel et de la représentation de connaissances. Actuellement les Systèmes de Gestion de Bases de Données Orientées Objet (ou plus simplement les Systèmes de Gestion de Bases de Données Objet, SGBDO), constituent un axe important de réflexion tant par la variété des prototypes et outils commerciaux proposés que par les travaux menés (du point de vue théorique ou expérimental). Ce chapitre présente, sans être totalement exhaustif, les principales fonctionnalités de ces systèmes au travers de leur dualité (programmation par objets et bases de données).

## 5.1 Introduction

Les systèmes de gestion de bases de données (SGBD) existent depuis trois décennies. Ils subissent des évolutions notables sous la double influence des avancées technologiques et théoriques. À l'heure actuelle, il serait hasardeux de prédire sur quelle voie définitive vont s'engager ces systèmes (relationnels étendus, déductifs, objets, etc.). Cependant l'approche objet est assez significative pour retenir l'attention.

Depuis leur naissance, les SGBD ont toujours intégré une composante de programmation. Dans la première génération, les types de données manipulés sont les mêmes dans les applications (écrites pour la plupart en COBOL) et dans la base. L'arrivée des SGBD relationnels a nécessité des conversions entre les types de données des applications (écrites en C, PL/1, etc.) et la base, ce qui, nous le relaterons, entraîne des dysfonctionnements aussi bien en phase de développement qu'en phase d'exécution.

Les SGBDO suivent leurs ancêtres et bénéficient des avancées de divers domaines : langages de programmation, représentation de connaissances, logique et bases de données. La volonté affirmée est de présenter des outils permettant le développement d'applications intégrant de manière transparente pour l'utilisateur leurs divers avantages. Le double héritage des langages de programmation et des bases de données pose cependant des problèmes divers que nous pouvons relater simplement. Pour les langages de programmation, données et programmes sont en général intégrés, le modèle de calcul est impératif (ou procédural), les contraintes sont exprimées sous forme de pré et post-conditions, la persistance n'est pas toujours assurée et dans le cas où elle est offerte, l'accès à la mémoire secondaire est explicite [1], l'expressivité se traduit par des notions de modularité, de paramétrisation, de généricité, etc. Pour les bases de données, notamment pour les bases de données relationnelles, programmes et données sont séparés, le modèle de calcul est déclaratif [2], les contraintes d'intégrité sont générales et déclaratives, la persistance est systématique et l'accès à la mémoire secondaire implicite, l'expressivité du langage est celle d'un langage de requête riche mais incomplet [3].

L'objectif majeur des SGBD du futur est d'intégrer les outils de programmation et de gestion de données au sein d'un même système. Cette intégration est à l'heure actuelle très diversement réalisée. Dans l'approche objet, on rencontre des systèmes qui assurent simplement la gestion d'objets persistants (relevant des langages de programmation par objets persistants) ; puis ceux qui intègrent la gestion des objets et la programmation d'applications (relevant de l'approche SGBD) et que nous qualifierons de SGBDO. Autrement dit, l'architecture de systèmes alliant persistance et programmation par objets peut être plus ou moins sophistiquée. Nous pouvons distinguer trois niveaux de complexité croissante :

- si l'on dispose du niveau « gestionnaire d'objets », le système est assimilable à un langage de programmation persistant couplé avec un système de gestion de fichiers (SGF),
- si l'on dispose, au-dessus du niveau précédent, de fonctionnalités de bases de données (concurrence notamment), le système supporte la multi-utilisation de données communes,
- enfin, si l'on dispose d'un édifice complet, avec langage de programmation, langage de requête et fonctionnalités de bases de données, on atteint pleinement la notion de SGBD.

Avant d'aller plus avant, il semble nécessaire d'effectuer quelques rappels qui permettront de préciser le domaine d'étude ainsi que les facteurs qui ont entraîné l'évolution des SGBD vers les SGBDO : c'est le sujet de la section 5.2. Dans la section 5.3, en nous appuyant sur le manifeste des SGBDO [Atkinson *et al.*, 1989], nous définirons les fonctionnalités qu'un système de gestion de bases de données objet doit présenter. Un panorama rapide des SGBDO et des standards actuels sera ensuite développé en section 5.4. La section 5.5 présentera les extensions nécessaires que doivent incorporer les SGBDO pour résoudre divers problèmes encore ouverts.

1. Le développeur doit explicitement signifier ce qui persiste.
2. Les langages de manipulation et d'interrogation décrivent ce que l'on veut obtenir et non comment l'obtenir.
3. Au sens de la calculabilité.

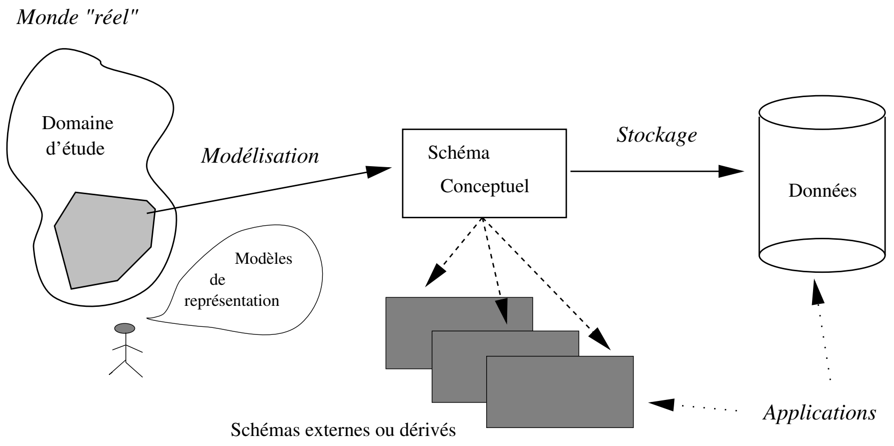



FIG. 5.1: *Étapes de la construction d'une base de données.*

## 5.2 Rappels

### 5.2.1 Base de données et système de gestion de bases de données

Une base de données (BD) est un ensemble *cohérent* de données *permanentes* qui est en général partagé par plusieurs utilisateurs pouvant y accéder simultanément (on parle alors de *concurrence* d'accès). Les systèmes de gestion de bases de données (SGBD) se distinguent des autres systèmes de programmation par deux qualités principales : leur capacité à gérer de grands volumes de données cohérentes et persistantes et leur capacité à accéder à ces données de manière efficace.

Tout SGBD doit assurer la cohérence, la persistance, le partage, la fiabilité et la sécurité des données [Ullman, 1989 ; Gardarin, 1993]. Une base de données est dite *cohérente* si l'ensemble des données persistantes en mémoire secondaire vérifient les diverses *règles d'intégrité* décrites au niveau de sa structure (désignée sous le nom de *schéma*). La persistance des données demande des méthodes de gestion de la mémoire secondaire ainsi que des méthodes d'accès particulières (indexation, regroupement, etc.). L'ensemble des données est partagé par plusieurs utilisateurs : le système doit garantir, par l'intermédiaire d'un mécanisme de *transactions*, des accès corrects et concurrents à chaque utilisateur. Enfin le système doit être *fiable* et *sûr* : il doit assurer la conservation des données en cas de panne logicielle ou matérielle et la sécurité des données en termes de contrôle des accès et de limitation de droits.

### 5.2.2 Étapes de la construction d'une base de données

La construction d'une base de données (FIG. 5.1) comporte une phase de développement qui consiste à analyser, concevoir et implémenter un schéma conceptuel unique et diverses applications adaptées aux besoins des utilisateurs qui s'appuient sur des schémas externes ou dérivés du schéma conceptuel global. Au cours de cette

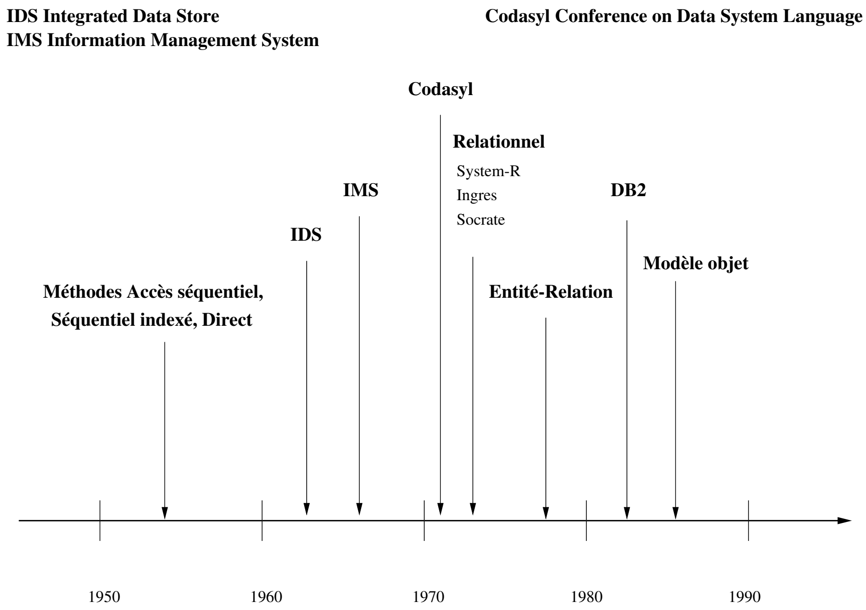



FIG. 5.2: *Évolution de la gestion des données.*

phase, le concepteur fait appel à une méthode d'analyse et de conception (cf. *chapitre 4*) qui lui permet de coordonner les divers aspects du système désiré (comme pour la représentation de connaissances le but est de rendre compte d'un « domaine » : cf. *chapitre 10*). La réalisation effective s'effectue ensuite (le plus fréquemment) après le choix d'un des SGBD existants. La phase ultérieure de la « vie » d'une base de données est celle de l'exploitation durant laquelle on effectue « l'instanciation » du schéma, les données sont stockées sur mémoire secondaire en cohérence avec le schéma et sont manipulables par les diverses applications développées entre temps.

La conception d'une base de données s'appuie sur les trois niveaux de description des données définis par le groupe ANSI/X3/SPARC [Tsichritzis et Klug, 1978] : *niveau conceptuel* (représentant la sémantique des données indépendamment de toute implémentation), *niveau interne* (description des supports physiques des données) et *externe* (vision des utilisateurs selon leur centre d'intérêt).

L'objectif majeur des SGBD est d'assurer l'indépendance des programmes et des données [Date et Hopewell, 1971] : plus précisément ils doivent assurer l'*indépendance logique* (toute modification du niveau conceptuel ne doit pas altérer le niveau externe) et l'*indépendance physique* (les modifications du niveau interne ne doivent pas altérer les niveaux conceptuel et externe).

### 5.2.3 Histoire

Avant les années 60, la gestion des données relève surtout de la gestion de fichiers et les efforts portent sur le développement de systèmes de gestion de fichiers

(SGF) partagés. L'histoire des SGBD débute ensuite par des systèmes « dédiés » conçus pour des applications particulières, puis la première génération de systèmes « génériques » voit le jour. Elle s'appuie sur des modèles dit *navigationnels* (réseau et hiérarchique). Ces modèles manipulent les concepts d'enregistrements logiques (ou entités) et de liens. Le langage de manipulation des données (LMD) offre des primitives qui permettent de parcourir des ensembles d'entités à travers les liens « orientés » existant entre eux. Le principal reproche formulé à l'encontre de ces modèles est leur manque d'indépendance physique. Cependant, de nombreux SGBD conçus alors (IMS[4], IDS[5], CODASYL, SYSTEM 2000) restent encore fort utilisés pour les applications traditionnelles de gestion. La parution de l'article de Codd [Codd, 1970] ouvre l'ère des SGBD *relationnels* (INGRES, DB2, ORACLE). Ceux-ci ont largement prouvé leurs avantages. L'organisation des données repose sur un modèle de données simple et formellement bien défini. Les langages de définition des données (LDD) et de manipulation de données (LMD) ont été standardisés (SQL [ANSI X3.135-1986, 1986]). Ces langages *déclaratifs* permettent la formulation de contraintes d'intégrité au niveau des schémas relationnels impliqués dans un schéma conceptuel et l'expression de requêtes plus ou moins complexes, tout en respectant l'indépendance des données. L'indépendance logique est, elle, assurée par la définition de *vues relationnelles*, c'est-à-dire des structures dérivées du schéma conceptuel et définies par une requête sur celui-ci. Le contrôle de la concurrence et de l'accessibilité aux données est fiabilisé : les SGBD relationnels proposent des mécanismes de transaction et de sécurité d'accès.

### 5.2.4 Les facteurs d'évolution vers les SGBDO

#### Nouveaux besoins en terme d'applications

L'émergence des « réseaux » informatiques (locaux, régionaux, mondiaux) ainsi que l'intérêt croissant que portent de nouveaux domaines de recherche à la gestion de leurs données contribuent à l'essor de nouveaux types d'applications pour lesquels les SGBD constituent un support incontournable. Biologie, médecine, chimie, etc. doivent représenter et manipuler des données complexes. L'information géographique et les sciences de l'environnement doivent faire face à des demandes croissantes en terme de rendus cartographiques et d'aide au diagnostic. Les applications de gestion souhaitent gérer des données multi-media. La conception et la fabrication assistées par ordinateur, les ateliers de génie logiciel (*AGL*), la bureautique et, au-delà, le « suivi des charges de travail » (*Workflow*) au sein des entreprises demandent d'assurer la gestion de versions de configurations (logicielles et matérielles) et se tournent vers les SGBD.

4. *Information Management System* (IBM) dont l'étude commença en 1966 en vue de l'opération spatiale Apollo pour ensuite devenir un des SGBD les plus répandus dans les années 70.

5. Développé par CII Honeywell-Bull.

**Les limites du relationnel**

Si les avantages des SGBD relationnels sont indéniables, leurs faiblesses sont maintenant bien déterminées :

- le modèle de données est trop simple et la contrainte de première forme normale (1FN) demande aux concepteurs des contorsions trop pénalisantes. En effet on ne manipule qu'un seul type de données (table) qui correspond à un ensemble de n-uplets. Le n-uplet est lui-même construit comme une concaténation de types primitifs : entier, réel, chaînes de caractères, booléen, etc.
- l'incompatibilité entre les langages de définition et de manipulation de données (LDD et LMD) et les langages de programmation « hôte » (*impedance mismatch*) nécessitent l'utilisation d'artifices de programmation (curseurs). Les types manipulés par les langages de requête (Structured Query Language (SQL), Query By Example (QBE), etc.) et les langages de programmation impératifs (COBOL, PL/1, PASCAL, C, etc.) sont incompatibles ; le modèle d'exécution de SQL est orienté « ensemble »[6] alors que celui des langages de programmation traditionnels est orienté « élément » ; enfin le modèle de calcul de SQL est incomplet [Date, 1984].

Pour ces multiples raisons, les applications conçues autour des SGBD relationnels sont souvent complexes à mettre en œuvre et à maintenir.

Pour pallier ces faiblesses, sont nés, dans un premier temps :

- des modèles de données complexes (NF2 *Non First Normal Form*) qui ne contraignent plus à la première forme normale [Pistor et Andersen, 1986 ; Abiteboul et Bidoit, 1986 ; Abiteboul *et al.*, 1987],
- des modèles « sémantiques » dérivés de modèles de représentation de connaissances qui enrichissent la variété des liens entre entités [Chen, 1979 ; Abiteboul et Hull, 1987 ; Parent *et al.*, 1989 ; Hammer et McLeod, 1981], et dont certains ont fortement influencé les méthodologies (cf. *chapitre 4*),

puis, progressivement, les modèles liés à l'approche objet ont fait leur apparition.

## 5.3 Fonctionnalités des SGBDO

### 5.3.1 Généralités

L'article « The Object Oriented Database System Manifesto » [Atkinson *et al.*, 1989] définit, ce que l'on est en mesure d'espérer d'un SGBDO. Nous allons nous appuyer sur cette définition pour préciser les différentes caractéristiques de ces systèmes. Les auteurs distinguent les caractéristiques nécessaires (*règles d'or*) que doivent posséder les SGBDO, puis les caractéristiques optionnelles qui feront en partie le sujet de la section 5.5.

6. La seule notion manipulée par SQL est la relation.

Les SGBDO doivent satisfaire aux deux critères essentiels :

- être des *systèmes à objets* et, à ce titre, permettre la gestion d'objets complexes (en fait le terme objet complexe est apparu dès que les attributs ou champs d'un objet n'étaient plus contraints à être de type atomique), l'identité d'objet, l'encapsulation, les notions de types et/ou de classes, d'héritage, de surcharge ;
- être des *systèmes de gestion de bases de données* et, à ce titre, assurer la persistance des données, leur gestion en mémoire secondaire, la concurrence et l'accès aux données via un langage de requêtes.

Les concepts spécifiques aux bases de données ne seront pas détaillés ici. Il existe une multitude d'ouvrages les décrivant [Ullman, 1989 ; Date, 1990 ; Gardarin et Valduriez, 1990 ; Kim, 1990 ; Bancilhon *et al.*, 1991 ; Delobel *et al.*, 1991 ; Gardarin, 1993 ; Benzaken et Doucet, 1993 ; Collet et Adiba, 1993 ; Kemper et Moerkotte, 1994 ; Abiteboul *et al.*, 1995 ; Cattell, 1997]. Les concepts objet sont développés dans les chapitres relatifs aux langages de programmation et aux méthodes d'analyse et de conception (cf. *chapitres 1, 2, 3, 4 et 8*). Nous nous attacherons simplement à montrer les diverses facettes d'un SGBDO, en précisant les spécificités issues de la combinaison des langages à objets et des bases de données.

Plusieurs modèles d'objets (plus ou moins formalisés) ont été proposés pour les SGBDO [Cardelli et Wegner, 1985 ; Maier *et al.*, 1986 ; Abiteboul et Hull, 1987 ; Banerjee *et al.*, 1987 ; Hull et King, 1987 ; Beeri, 1989 ; Hammer et McLeod, 1981] issus des réseaux sémantiques ou des langages de classes [Masini *et al.*, 1989].

Avant d'aborder, plus en détail, les fonctionnalités des SGBDO, nous allons présenter un exemple succinct relatif à la gestion simplifiée d'un institut de formation universitaire, que nous utiliserons dans la suite du propos.

Au sein de l'institut, les personnes regroupent la population étudiante et enseignante. Chaque personne a un nom, une liste de prénoms et une adresse comportant une rue, un numéro et une ville. Les étudiants possèdent de plus un numéro de carte d'étudiant, les enseignants ont un grade et un salaire. L'institut propose des cours. Chaque cours a un intitulé et un numéro, ainsi qu'un volume horaire et un ensemble de cours qui sont prérequis. Un cours est enseigné par un et un seul enseignant et un enseignant peut assurer plusieurs cours. Un étudiant peut suivre plusieurs cours existants, et évidemment un cours doit être suivi par plusieurs étudiants. Enfin certains étudiants peuvent accéder au statut particulier d'Attaché Temporaire de Recherche (Ater) et, à ce titre, ils peuvent enseigner.

En utilisant le formalisme de la méthode OMT [Rumbaugh *et al.*, 1991] (cf. *chapitre 4*), le concepteur propose le diagramme d'objets de la figure 5.3. Ce diagramme constitue la représentation du schéma conceptuel de la base de données. Les classes `Personne`, `Etudiant`, `Enseignant`, `Ater` et `Cours` sont décrites avec leurs attributs et leurs opérations. Les classes `Etudiant` et `Enseignant` sont des spécialisations de la classe `Personne` ; la classe `Ater` est une spécialisation des classes `Etudiant` et `Enseignant`. La classe `Cours` est associée à la classe `Etudiant`, à la classe `Enseignant` et à elle-même. Chaque association est décrite par les verbes qui précisent comment les classes associées sont liées entre elles et par une arité conforme au contraintes données dans le texte initial.

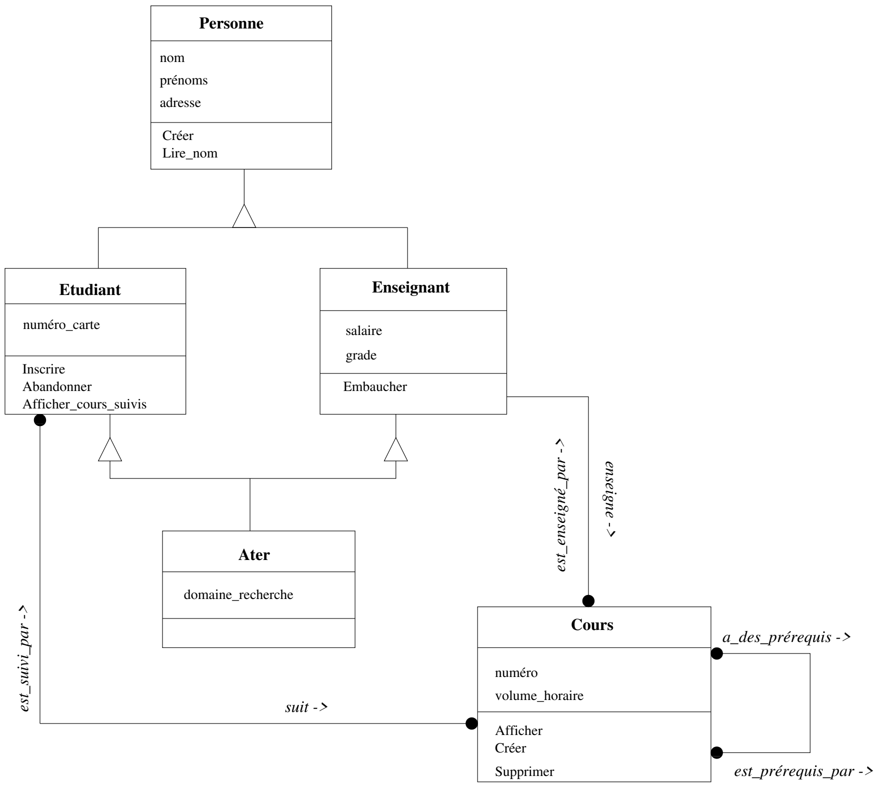



FIG. 5.3: *Modèle objet d'une gestion simplifiée d'institut universitaire.*

La « traduction » de ce modèle dans une syntaxe proche de celle du SGBD $O_2$ [Bancilhon *et al.*, 1991] va servir de base pour étayer les divers points énumérés en introduction (FIG. 5.4).

## 5.3.2 Être un système à objets

Nous allons rapidement présenter en quoi les SGBDO suivent le paradigme *objet*. Les concepts sont hérités des langages de programmation par objets (LPO) (cf. *chapitres 1, 2, 3 et 8*), mais la spécificité des bases de données transparaît.

### Objets et identification

Le concept essentiel partagé par tous les SGBDO est bien sûr celui d'*objet*. Un objet est une collection de données structurées identifié par une référence unique et pouvant réagir à divers stimuli. En d'autres termes, un objet possède une *identité*, un *état* (valuation de ses propriétés statiques ou attributs) et un *comportement* : il peut répondre à des messages provenant d'autres objets.

```
Schema Gestion_Universitaire
Class Personne
tuple   (nom : string,
         prénoms : list(string),
         adresse : tuple(rue : string, num : integer, ville : string))
Method   Créer (n : string) : Personne,
         Lire_nom : string
End Personne
Class Etudiant inherit Personne
tuple   (numéro_carte : integer,
         cours_suivis : set(Cours))
Method   Inscrire (c : Cours),
         Abandonner (c : Cours)
End Etudiant
Class Enseignant inherit Personne
tuple   (grade : string,
         salaire : real,
         cours_enseignés : set(Cours))
Method   Embaucher()
End Enseignant
Class Ater inherit Enseignant,Etudiant
tuple   (domaine_recherche : string)
End Ater
Class Cours
tuple   (numéro : string,
         volume_horaire : integer,
         inscrits : set(Etudiant),
         enseignant : Enseignant,
         ....)
Method   Créer(num : string) : Cours,
         Supprimer,
         Afficher,
         ....
End Cours
```

FIG. 5.4: *Schéma de la base de données gestion universitaire.*

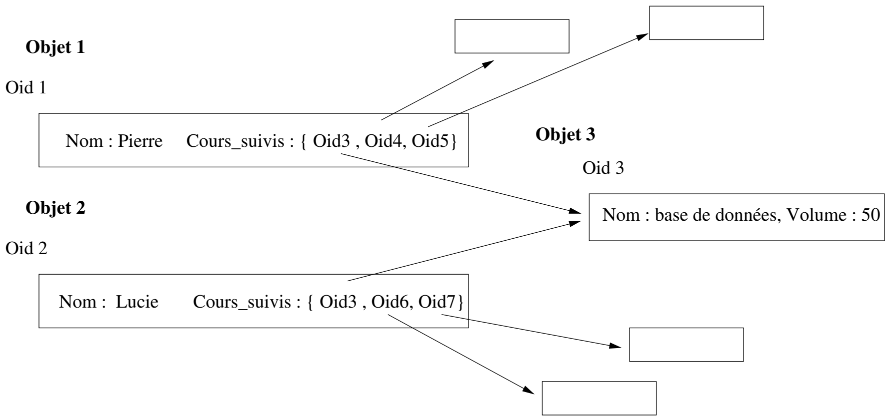



FIG. 5.5: *Pierre et Lucie suivent le même cours.*

L'identité unique (*object identifier* ou *oid*) traduit simplement le fait que l'objet existe : elle est indépendante de l'état de l'objet. Tout objet peut, au sein du système, être associé à un doublet (*oid*, état). Il conserve cette identification immuable au cours du temps tout en admettant que son état puisse être altéré aussi souvent que nécessaire. Ce principe d'identité unique indépendant de l'état, constitue, en soi, une nouveauté par rapport aux principes des SGBD relationnels. Ceux-ci, dénotés *orientés valeur* manipulent, quant à eux, des propriétés *identifiantes* ou *clés* dépendantes de la valeur. La présence d'une clé dans un schéma relationnel assure l'unicité des valeurs associées aux attributs qui la constituent, d'où l'unicité des n-uplets concernés.

Les objets ayant une identité unique, la notion de partage d'objets devient chose naturelle.

À titre d'illustration, supposons que Pierre et Lucie soient des étudiants inscrits au même cours. Dans un système à objets, les deux objets `Pierre` et `Lucie` ont un cours en commun le `cours (oid3, Nom : "base de données", Volume : 50)` (FIG. 5.5). Une mise à jour sur l'état de cet objet `cours` sera « effective » pour tous les étudiants partageant cet objet.

Dans un système relationnel, la représentation du fait que Pierre et Lucie soient des étudiants inscrits dans un même cours peut se traduire par les n-uplets suivants : `("Pierre", "bases de données", 50),("Lucie", "bases de données", 50)`. Le partage du cours n'est traduit, implicitement, que par l'égalité de la valeur des attributs `nom` et `volume`. Les mises à jour peuvent être faites pour les deux n-uplets séparément ou non. Pour pallier cette ambiguïté nous aurions pu dans le modèle relationnel instaurer une « indirection » pour assurer la mise à jour simultanée pour tous les étudiants qui suivent le même cours, ce qui aurait alourdi la représentation. Les systèmes relationnels les plus récents proposent la notion de contrainte *d'intégrité référentielle* exprimée entre deux schémas relationnels. Cette contrainte lie l'existence et par suite toutes les mises à jour d'un n-uplet d'une relation à celles d'un autre n-uplet « référent » dans une autre relation. En fait, grâce

```
Objet 1 : (oid1, [Nom: "Pierre", Cours_suivis: {Oid3, Oid4, Oid5}])
Objet 2 : (oid2, [Nom: "Lucie", Cours_suivis: {Oid3, Oid6, Oid7}])
Objet 3 : (oid3, [Nom: "BD", Volume: 50])
Objet 4 : (oid4, [Nom: "LPO", Volume: 50])
Objet 5 : (oid5, [Nom: "Syst                    eme", Volume: 70])
Objet 6 : (oid6, [Nom: "BD", Volume: 50])
Objet 7 : (oid6, [Nom: "Algo", Volume: 30])
Objet 8 : (oid8, [Nom: "Pierre", Cours_suivis: {Oid6, Oid4, Oid5}])
Objet 9 : (oid9, [Nom: "Lucie", Cours_suivis: {Oid3, Oid6, Oid7}])
```

*Les neuf objets sont distincts c'est-à-dire qu'ils ont des identités différentes.*
*Objets 2 et 9 sont égaux en surface et en profondeur.*
*Objets 1 et 8 sont égaux en profondeur mais différents en surface.*
*Objets 1 et 2 sont différents en surface et en profondeur.*

FIG. 5.6: *Égalité et copie.*

au concept d'objet (et d'identité unique), les SGBD objets permettent de mettre en œuvre un mécanisme de mises à jour cohérent qui prévient de toute anomalie et qui est proche de l'intégrité référentielle des systèmes relationnels.

L'identification d'objets permet de définir, pour des objets instances d'une même classe, des opérateurs de comparaison et de copie :

- *identité* : deux objets identiques correspondent au même objet. L'objet `cours` suivi par `Pierre` est identique à l'objet `cours` suivi par `Lucie`.
- *égalité* (de surface et en profondeur) : deux objets sont égaux en surface s'ils ont les mêmes valuations pour leurs attributs ; ils sont égaux en profondeur s'ils ont des « graphes de composition » isomorphes au niveau de chacun de leurs attributs et ont les mêmes valuations pour leurs attributs terminaux.
- *copie* (superficielle et profonde) en accord avec les opérateurs d'égalité précédents.

Les systèmes relationnels peuvent simuler l'identité au sens objet en rajoutant des identificateurs de n-uplets (notion de *surrogate* introduite par Codd [Codd, 1979]). Le concept d'*identité* indépendante de la valeur avait déjà été introduit dans les SGBD navigationnels qui gèrent pour tout *enregistrement* ou *record* un identifiant système appelé *clé d'enregistrement*.

De leur côté, les systèmes à objets peuvent rajouter la notion de *clé* d'objet analogue à celle des systèmes relationnels traduisant l'obligation d'unicité de la valeur pour les attributs définissant cette clé pour toute instance créée (cf. *chapitre 10*).

**Classes et types**

Sans entrer dans une discussion qui dépasse le cadre de ce chapitre, notons que deux approches différentes existent parmi les SGBDO selon qu'ils dérivent de lan-

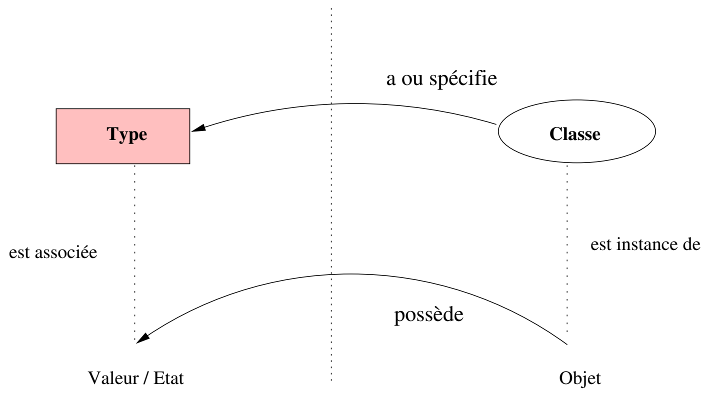



FIG. 5.7: *Classes et types dans les* SGBDO.

gages comme SMALLTALK, LISP, CLOS ou de langages comme C++. Certains relèvent du tout objet : les classes sont des objets et décrivent un type ; les autres différencient classes et types, objets et valeurs[7] (cf. *chapitre 1*).

Comme la terminologie est souvent sujette à controverses, on peut en quelques lignes définir les termes types et classes tels qu'ils sont admis par la communauté bases de données.

La notion de type correspond à la notion de type utilisée dans les langages de programmation tels que PASCAL, C, etc. Un type est « instancié » par des valeurs qui n'ont pas une identité au sens de celle des objets.

La notion de classe, elle, recouvre en fait deux versants :

- Une classe est accompagnée d'une description en *intension* (cf. *chapitre 12*). La spécification de cette description est analogue à celle d'un type construit auquel on adjoint des opérations spécifiques. En phase d'exécution, la classe sert à créer les objets qui sont ses instances.

- Une classe a une composante *extensionnelle*. Tous les objets instances de la classe constituent son extension. On peut distinguer la notion d'extension « propre » correspondant uniquement aux instance propres de la classe et la notion d'extension « globale » correspondant aux instances de la classe et de celles de toutes ses sous-classes. Pour les SGBDO, ce rôle d'« entrepôt » assigné à la classe est essentiel en phase d'exploitation (les instances devant pour la plupart être sauvegardées sur support physique).

Certains SGBD ont un LDD qui définit des types indépendamment des classes et un LMD qui manipule par suite des valeurs indépendamment des objets.

7. Au sens où l'on trouve dans ces SGBDO comme dans certains langages de programmation par objets des types qui ne sont pas des classes et des valeurs qui ne sont pas des objets. Cependant toute classe a un type associé.

```
Class Equipe_enseignante
  set(Enseignant)
End Equipe_enseignante
```

FIG. 5.8: *Classe collection*

```
Method body Lire_nom : string
in Class Personne
 {return self.nom;}
```

FIG. 5.9: *Code pour Lire_nom*

La description intensionnelle d'une classe comporte :

- *Une partie structurelle.* Tout SGBDO offre aux concepteurs des modèles de données des constructeurs de structures et collections complexes : *n-uplet, ensemble, multi-ensemble, liste*. Le constructeur de n-uplet (`tuple`) crée une structure composée d'une suite d'attributs. À chaque attribut est associé un domaine de définition qui peut être (si la distinction classe/type existe) un type ou une classe prédéfinis (entier, réel, booléen, etc.), un type construit ou une classe. Les constructeurs ensemble (`set`), multi-ensemble (`bag`), liste (`list`), tableau (`array`) créent des structures résultant de leur application sur un domaine (même sens que précédemment) (cf. les déclarations de classes de la figure 5.1). Ils permettent la définition naturelle de classes « collections » polymorphes.

  La classe `Equipe_enseignante` (FIG. 5.8) est polymorphique au sens où son extension pourra regrouper des instances de la classe `Enseignant` et de la classe `Ater`.

  Tout constructeur peut être appliqué à tout domaine y compris ceux qui sont définis par d'autres constructeurs (cf. la classe `Equipe_enseignante`) où `set` est appliqué au type de la classe `Enseignant`). Ceci n'était pas possible dans le modèle relationnel ou le constructeur `relation` ne s'applique qu'à la structure créée par le constructeur `n-uplet`.

- *Une partie comportementale.* Les opérations ou méthodes auxquelles les objets pourront être soumis lors de l'invocation de messages sont définies au niveau des classes comme dans un langage de classes ordinaire. Leur spécification est donnée sous forme d'une signature précisant le nom de l'opération (méthode) et la liste des divers arguments ( FIG. 5.4). Le code d'implémentation des méthodes peut être directement attaché à la description de la classe ou défini indépendamment de la partie spécification.

  L'opération `Lire_nom` de la figure 5.9 est une opération *accesseur* permettant à une instance de la classe `Personne` d'accéder à son attribut `nom`.

Enfin pour conclure, dans le cadre des SGBDO, l'ensemble des classes (complété éventuellement d'un ensemble de types) relatif au domaine du monde réel représenté constitue la notion de *schéma conceptuel de base de données*. Ce schéma est conservé dans la base : selon les systèmes, la partie comportementale (méthodes des classes) est sauvegardée avec le schéma, donc persiste dans la base, ou bien est « attachée » uniquement aux applications qui l'utilisent.

Comme dans les LPO, l'approche objet dans le contexte des bases de données supprime la dichotomie programmes/données puisque l'objet les regroupe dans une même entité. L'utilisateur final ne doit plus connaître la structure de données puisqu'il ne manipule l'objet que par l'intermédiaire des méthodes qui constituent son interface (*encapsulation*).

Ce principe d'encapsulation répond à des objectifs précis en phase de développement (simplification, réutilisation) (cf. *chapitres 1, 3 et 8*). Les SGBDO, comme les systèmes de représentation de connaissances, n'appliquent pas tous strictement l'encapsulation (cf. *chapitre 10*) ; la problématique des bases de données reste tout de même de « donner accès aux données le plus aisément possible ». L'encapsulation stricte semble alors être contraire aux nécessités des LMD déclaratifs de type SQL. On trouve donc, selon les systèmes, la définition de divers niveaux d'encapsulation :

- soit l'encapsulation est stricte et tout accès (lecture/écriture) aux données d'un objet se fait par appel d'une méthode de la classe de l'objet concerné (c'est un des garants de la réutilisation),
- soit l'encapsulation est réservée aux écritures, les accès en lecture peuvent être faits directement,
- soit l'encapsulation est partielle : la structure des classes comporte deux parties, l'une visible (publique) et l'autre privée (accessible uniquement par l'appel de méthodes)(cf. *chapitres 2 et 3*).

La notion d'encapsulation, dans les SGBDO, peut être étendue au niveau du schéma. Certains systèmes offrent la possibilité aux concepteurs d'encapsuler les schémas qu'ils définissent. L'objectif est toujours la réutilisation. La spécification d'un schéma comporte donc la définition de classes exportables vers d'autres schémas [Collet et Adiba, 1993]. En fait nous retrouvons ici l'idée de bibliothèque de classes partageables déjà présente dans les LPO (cf. *chapitre 2*).

**Composition et agrégation**

Au sein d'un schéma de base de données, les classes sont impliquées dans plusieurs types d'associations qui peuvent se spécialiser conceptuellement et sont désignées sous les termes de relation de *composition* et de relation de *généralisation/spécialisation*.

La terminologie dans le domaine des objets (objets complexes, objets composites, référence, composition, agrégation) est assez variée, non unifiée et nécessite quelques explications complémentaires. La conception d'un objet du monde réel consiste à décrire les diverses propriétés (attributs ou champs) de l'objet. Nous avions déjà noté, que le terme objet complexe est apparu dès que les attributs ou champs d'un objet n'étaient plus contraints à être de type atomique. Dans le schéma de la figure 5.4, la classe `Personne` définit la structure générale d'objets complexes que seront ses instances.

Les attributs d'un objet peuvent référencer d'autres objets indépendants : on peut parler alors de relations de dépendance « faible » entre objets s'il n'y a pas de contraintes spécifiques définies entre eux. Dans la figure 5.4, la classe `Etudiant` définit la structure d'objets complexes référençant un ensemble d'instances de la classe `Cours`.

```
Class Voiture
tuple (num_im : string,
       moteur : Moteur,
       roues : set (Roue))
End Voiture
```

FIG. 5.10: *Classe composite*

```
Class Personne
tuple (nom: string,
       .....,
       voiture : Voiture))
End Personne
```

FIG. 5.11: *Autre classe composite*

On peut aussi explicitement introduire la notion d'*agrégation* groupant des objets composants pour en faire un *objet composite*. L'objet composite est alors perçu comme une entité de niveau supérieur construite avec des entités composantes (hiérarchie construite avec la relation *partie-de*).

La classe `Voiture` de la figure 5.10 illustre la structure générale d'objets composites. Les classes `Moteur` et `Roue` sont des classes composantes de la classe `Voiture`. Le concepteur, par l'agrégation, veut que les objets instances de la classe `Voiture` soient perçus différemment de tas de pièces détachées correspondant à leurs composantes, instances des classes `Moteur` et `Roue`.

Précédemment, la classe `Equipe_enseignante` de la figure 5.8 définissait aussi une structure particulière d'objets composites : `Enseignant` est la classe composant de la classe `Equipe_enseignante`.

Autour de la notion d'objets composites, les modèles proposent différentes variantes de sémantiques liées à des règles de gestion ou à des contraintes d'intégrité :

- Un objet composant *peut être ou non partagé* entre plusieurs objets de la même classe.

  La classe `Voiture` de la figure 5.11 est une classe d'objets composants pour la classe `Personne`. Le concepteur peut choisir si cette classe constitue ou non un composant partageable, c'est-à-dire si l'application visée utilise la règle de gestion « toute personne peut utiliser une voiture qui peut aussi être utilisée par d'autres personnes » ou si, au contraire « toute personne dispose d'une voiture qui lui est propre ».
- Un objet composant peut être *existentiellement dépendant ou non de l'objet composite*. Nous retrouvons ici la notion d'intégrité référentielle des bases de données relationnelles (cf. § 5.3.2). Ceci demande la mise en œuvre de stratégies particulières lors des phases de création et de destruction de ces divers objets. Certains optent pour la vision « un objet composite ne peut être créé (ou détruit) sans que sa création (ou sa destruction) entraîne la création (ou la destruction) de ses composants », d'autres optent pour une vision plus « libérale » où l'on admet que les composants peuvent exister indépendamment de leur « tout ».

Une bonne définition et discussion de ces divers aspects se trouve dans [Kim, 1990 ; Puig et Oussalah, 1996].

### Généralisation/spécialisation

La généralisation/spécialisation relève d'un mécanisme d'abstraction qui permet de percevoir certains ensembles d'objets du monde réel comme l'union de plusieurs sous-ensembles, chaque sous-ensemble ayant des propriétés particulières. Par exemple, les étudiants ont un nom, une liste de prénoms, une adresse, suivent des cours, peuvent se marier, mourir, etc. ; les enseignants ont aussi un nom, un ensemble de prénoms, une adresse, ils donnent des cours, peuvent se marier, mourir, etc. ; le concept de généralisation/spécialisation permet de décrire les personnes qui regroupent les caractéristiques communes aux étudiants et aux enseignants et de conserver les caractéristiques spécifiques pour spécialiser les étudiants et enseignants. Dans la figure 5.1 : la classe `Personne` généralise les classes `Etudiant` et `Enseignant`, les classes `Etudiant` et `Enseignant` spécialisent la classe `Personne`.

Le concept de généralisation/spécialisation est le même que celui des LPO et de RCO (cf. *chapitres 1, 3, 10 et 12*) : il est réalisé par le mécanisme d'héritage.

Déclarer $C_1$ sous-classe directe de $C_2$ signifie que :

- la population (l'extension) de $C_1$ est incluse dans la population de $C_2$,
- $C_1$ *hérite* des attributs et des méthodes de $C_2$,
- le type associé à la classe $C_1$ est un sous-type de celui associé à la classe $C_2$ (respect des règles de sous-typage),
- tout objet instance de $C_1$ peut jouer le rôle d'un objet instance de $C_2$ (c'est-à-dire qu'il peut se substituer au niveau du comportement à celui de l'instance de $C_2$).

**Surcharge et liaison dynamique**

Les SGBDO permettent la surcharge et le masquage des opérations (méthodes) comme les LPO dont ils dérivent. Les conflits éventuels résultant de généralisations multiples sont gérés comme dans les langages de programmation. Le schéma conceptuel doit être *consistant* [Waller, 1991 ; Zicari, 1991]. Cette notion recouvre deux aspects :

- La consistance *structurelle*. En phase de développement, le schéma est un graphe orienté sans circuit (appellé hiérarchie d'héritage en LPO) ; les définitions des attributs et des méthodes sont conformes aux règles définies par l'héritage. Le domaine des attributs et la signature des méthodes sont conformes aux règles de sous-typage (cf. *chapitre 1*).
- La consistance *comportementale*. Ici il s'agit d'une notion liée à la phase d'exploitation. Si aucune erreur ou aucun résultat aberrant ne se produisent, la consistance comportementale du schéma est vérifiée (cf. *chapitres 1 et 2*). L'intérêt de cette propriété, est de pouvoir disposer d'outils capables de la contrôler : lorsque les méthodes subissent des mises à jour, toute exécution ultérieure utilisant des appels à ces méthodes, reste assurée de terminaison normale [Castagna, 1995].

### 5.3.3 Être un SGBD

Les SGBDO doivent assurer la gestion et les accès concurrents à de larges volumes de données. Nous allons détailler ici en quoi l'approche objet influe sur les fonctionnalités classiques des SGBD.

**Persistance**

Toute donnée même complexe doit pouvoir être manipulée par le système mais aussi doit pouvoir être stockée dans la base c'est-à-dire doit pouvoir devenir *persistante*. Cette fonctionnalité évidente du point de vue des bases de données constitue une fonctionnalité qui, le plus souvent, est ignorée des LPO. En effet, pour ceux-ci, les objets créés ont une durée de vie limitée à la durée de l'exécution du programme qui les fait naître. Quelques langages permettent de conserver en mémoire secondaire, par l'intermédiaire de fichiers, les données relatives aux objets créés en mémoire principale (PS-ALGOL [Atkinson *et al.*, 1983], les versions persistantes de JAVA [Atkinson *et al.*, 1996], SMALLTALK [Straw *et al.*, 1989] et C++ [Channon, 1996], YAFOOL [Ducournau, 1991], etc.).

Pour les SGBDO, d'une part le schéma de la base de données, c'est-à-dire l'ensemble des classes (et types), doit perdurer, puisque c'est sur l'ensemble des définitions qu'il comporte que vont s'appuyer les diverses applications, d'autre part, pour toute application les objets ou valeurs créés et manipulés doivent pouvoir être conservés eux-mêmes, puisqu'ils contribuent à enrichir la base de données proprement dite.

Pour ces objets ou valeurs, le *modèle idéal* de persistance doit présenter les propriétés suivantes :

- *Transparence* : les applications ou programmes doivent manipuler de manière uniforme objets ou valeurs « temporaires » (éphémères ou volatiles) et objets ou valeurs persistants ;
- *Orthogonalité avec type/classe* : la persistance est applicable à tout objet ou toute valeur indépendamment de sa classe ou de son type ;
- *Intégrité* : toutes les références des objets ou valeurs complexes ou composites restent « valides » après toute opération sur la base.

Différentes approches pratiques sont proposées dans les SGBDO pour permettre au concepteur de manipuler la persistance :

- La persistance est associée aux classes lors de leur déclaration au niveau schéma et par suite toute instance d'une classe déclarée persistante est persistante. La solution généralement adoptée consiste à définir une classe racine du graphe des classes persistantes. Le développeur doit manipuler en fait une double hiérarchie, celle des classes permettant la création d'instances persistantes et celle permettant la création d'instances temporaires (FIG. 5.12). Remarquons que cette approche ne respecte pas l'orthogonalité dont nous parlions précédemment. De plus le maintien de deux sous-hiérarchies ne devrait pas exister : les objets persistants n'étant qu'un sous ensemble, choisi par l'utilisateur, des objets temporaires qu'il crée.
- La persistance est réalisée, de manière explicite, à la déclaration ou à la création des instances. C'est le programmeur qui donne le statut persistant ou non, soit lors de la déclaration de la *variable* référençant une future instance, soit lors de la création de l'instance (FIG. 5.13).

```
Class Objet
Class Personne inherit Objet
// classes à instances temporaires ou volatiles
Class Persistant Pobjet
Class Ppersonne inherit Pobjet
// classes à instances persistantes
```

FIG. 5.12: *Classes persistantes et non persistantes.*

- La persistance peut être « dynamique », un même objet pouvant changer de statut au cours de sa vie. La persistance est obtenue lors de l'*attachement direct* ou indirect d'une instance à un *point d'entrée* dans la base (racine de persistance). L'objet peut aussi être attaché explicitement à un support de stockage (fichier), c'est cette démarche qui est le plus souvent adoptée dans les langages de programmation persistants.

  Dans l'exemple de la figure 5.14, l'application dispose des classes `Etudiant` et `Cours` et déclare (ici par la primitive `name`) deux points d'entrée dans la base : `Les_Etudiants` référençant une valeur collection et `Responsable` référençant un objet. Des instances temporaires de la classe `Etudiant` et de la classe `Cours` sont créées (`p`, `q`, `r` et `c`), on prend ensuite en compte le fait que `p` et `q` suivent le même cours `c`. L'instance `r` est promue responsable et persistante après avoir été liée à `Responsable`. Les instances `p`, `q`, `r` sont aussi persistantes car liées directement à `Les_Etudiants` (l'opérateur `+=` met à jour l'ensemble `Les_Etudiants` en réalisant l'union de celui-ci avec l'ensemble construit par le constructeur `set` appliqué aux instances `p`, `q`, `r`) ; l'instance `c` persistera aussi car liée indirectement, puisque référencée dans le champ `cours_suivis` d'instances persistantes.

Pour le *gestionnaire d'objets* du SGBDO qui doit assurer la persistance des objets/valeurs le problème essentiel à résoudre est celui de l'*identification*. Ensuite se posent des problèmes liés aux accés à ces valeurs/objets persistants et à leur optimisation : le *regroupement* (*clustering*) et l'*accès associatif*.

- *L'identification* consiste à passer d'une « poignée de pointeurs » en mémoire principale à des adresses en mémoire secondaire. Pour cela, il faut implémenter des *Oid* : ils peuvent être *physiques* ou *logiques* (pas d'information directe sur la localisation). Ensuite il faut utiliser des mécanismes de *localisation* pour les

```
Personne persistant p; // déclaration de p persistant
Personne q; // déclaration de q temporaire ou volatile
p = new Personne; q = new Personne;
ou
Personne p,q; // déclaration
p = new persistant Personne // p devient persistant
q = new Personne // q est volatile
```

FIG. 5.13: *Instances temporaires et persistantes.*

```
name Les_Etudiants : set(Etudiant);
name Responsable : Etudiant;
Etudiant p,q,r;
p = new Etudiant; q = new Etudiant; r = new Etudiant;
Cours c = new Cours;
p.cours_suivis += set(c); q.cours_suivis += set(c);
Responsable = r;
Les_Etudiants += set(p,q,r);
```

FIG. 5.14: *Racines de persistance.*

objets temporaires ou persistants. L'accès aux objets est le plus fréquemment réalisé par un mécanisme d'indirection qui consiste à aller chercher dans une table la référence à l'objet soit en mémoire principale, soit en mémoire secondaire. De nouvelles techniques sont mises en œuvre pour optimiser les accès : il s'agit de techniques de *conversion* (*pointer swizzling*) qui permettent d'éviter la recherche dans la table d'indirection. Ces techniques consistent à associer à un objet un format externe contenant son *Oid* et un format interne obtenu en remplaçant l'*Oid* par l'adresse en mémoire principale ou secondaire [Kemper et Kossmann, 1993]. En d'autres termes, l'objet présente une interface qui permet d'accéder à son format externe (*Oid*) et son implémentation est en fait une adresse physique en mémoire secondaire ou principale.

- *Le regroupement* consiste, si l'on veut le traduire simplement, à forcer la présence contigüe sur un même support physique des objets qui doivent être accédés simultanément. À partir du schéma (hiérarchie de généralisation-spécialisation et graphes de composition des classes), l'administrateur de la base peut définir des *arbres de placement* (il effectue, en quelque sorte, une classification prédéfinie des objets) [Bancilhon *et al.*, 1991 ; Delobel *et al.*, 1991 ; Kemper et Moerkotte, 1994 ; Cattell, 1997]. La stratégie d'accès aux objets regroupés s'appuie ensuite sur des algorithmes de parcours de ces diverses structures.

- *L'accès associatif* est naturel pour les SGBD relationnels : on peut atteindre la « valeur » des n-uplets par des techniques d'indexation (index par *fonction de hachage*, par *arbre-B, arbre-B+, etc.*) qui *optimisent* les plans d'exécution des requêtes et, par suite, la durée d'exécution de celles-ci lorsqu'elles sont effectuées sur la base [Ullman, 1989]. Ces techniques d'indexation doivent être adaptées au contexte des objets et relèvent encore du domaine de la recherche [Cattell, 1997].

Toute cette variété de mécanismes, destinés à optimiser les accès à la mémoire secondaire, doit rester invisible pour le programmeur d'applications, afin de respecter le critère d'indépendance physique.

```
Etudiant p = new Etudiant;.....
Cours c = new Cours (2,50);......
p.Inscrire(c);
....
Cours cc;
for (cc in p.cours_suivis) cc.Afficher;
name Les_Etudiants : set (Etudiant);
Les_Etudiants += set(p);
for (e in Les_etudiants where e.adresse.ville = "Arles")
e.Lire_nom;
```

FIG. 5.15: *Exemple d'application.*

**Langages de programmation et langages de requêtes**

*Langage de programmation*

En phase d'exploitation, l'interface naturelle des SGBDO est *procédurale* et *navigationnelle*. Les utilisateurs accèdent à la base par l'intermédiaire d'applications que l'on peut considérer comme des procédures ou des modules indépendants et qui leur permettent le plus souvent de naviguer dans divers ensembles d'objets éphémères ou persistants. Les applications sont développées par des programmeurs qui utilisent un langage similaire aux LPO ; les accès et mises à jour des objets de la base sont réalisés par l'envoi de messages. Pour que ce langage de programmation devienne un LMD, il faut cependant lui adjoindre des opérations de manipulation relatives aux constructeurs des collections (`set`, `multi-set`, `list`).

À partir du schéma de la figure 5.4 on peut donner un aperçu de ce style d'application.

Dans la figure 5.15 l'application crée des instances `p` et `c` de la classe `Etudiant` et de la classe `Cours`, celles-ci tout d'abord temporaires sont rendues persistantes par attachement au point d'entrée `Les_Etudiants`. On notera ici que l'accès aux attributs comme aux méthodes se fait de manière unifiée par la syntaxe : `p.cours_suivis` (accès à l'attribut de `p`) et `e.Lire_nom` (envoi du message `Lire_nom`). Afin de visualiser les cours suivis par `p` on utilise l'itérateur `for (.. in..)` pour parcourir la collection `p.cours_suivis`. Pour connaître le nom des étudiants montpelliérains, on utilise l'itérateur-sélecteur `for (.. in .. where ..)`. L'usage de ces opérateurs se distingue de l'activation de méthodes par envoi de messages, qui relève de la pure programmation objet.

*Le langage de requêtes*

Pour rester conforme à l'esprit des SGBD relationnels, les SGBDO doivent aussi offrir, aux utilisateurs éventuels, la possibilité d'interroger la base de la manière la plus déclarative possible (il s'agit alors de déclarer ce que l'on veut obtenir et non de décrire comment y parvenir). Un *navigateur*[8] graphique permet d'accéder à la base, mais il ne satisfait pas aux critères de déclarativité et de haut niveau que revendiquait le *calcul relationnel* (LMD de la théorie relationnelle basé sur un langage prédicatif),

8. Désigné aussi sous le terme de *browser*.

```
//Requête 1
Responsable

//Requête 2
Responsable.Lire_nom
```

FIG. 5.16: *Requêtes de base.*

voire SQL. Certains systèmes proposent donc un langage de requêtes à part entière. Ce langage spécifique est utilisable soit dans un environnement particulier (le mode « interrogation » interactif), soit « immergé » dans le langage de programmation.

Diverses sortes de requêtes sont alors réalisables parmi lesquelles on peut distinguer :

- Les requêtes *de base* qui donnent accès à l'état d'un objet. Dans l'esprit « base de données », cet accès est libre pour tout utilisateur autorisé, ce qui a pour conséquence dans le contexte objet soit de lever l'encapsulation (cf. § 5.3.2), soit de demander au concepteur d'écrire autant d'opérations d'accès en lecture/écriture qu'il y a de champs pour l'objet.

  Les requêtes 1 et 2 de la figure 5.16 sont des requêtes de base correspondant aux interrogations : « qui est le `Responsable` ? » et « quel est le nom du `Responsable` ? »
- Les requêtes dites *d'accès associatif* qui réalisent des sélections d'objets appartenant à des collections, à partir de conditions relatives aux valeurs de leurs attributs. Les langages proposent alors une variante de la forme `Select From Where` de SQL. La clause `Select` définit la structure du résultat désiré, la clause `From` introduit les collections auxquelles on doit accéder, la clause `Where` spécifie le filtre utilisé. Remarquons que les requêtes correspondent, en autorisant plus d'expressivité, au mécanisme de filtrage des langages de représentation des connaissances (cf. *chapitres 10 et 12*).

  Les requêtes 1, 2 et 3 de la figure 5.17, correspondent aux interrogations suivantes : « ensemble des `Étudiants` de nom `Pierre` résidant à `Arles` » ; « en-

```
//Requête 1
Select e
From e in Les_Etudiants
Where e.nom = "Pierre" or e.adresse.ville = "Arles"

//Requête 2
Select e
From e in Les_Enseignants
Where e.nom like "A*" and Count(e.cours_enseignés) > 2

//Requête 3
Select e
From e in Les_Etudiants, c in e.cours_suivis
Where e.numéro_carte > 125 and c.numéro = 45
```

FIG. 5.17: *Requêtes associatives.*

```
//Requête 1
Select tuple (nom : e.nom, ville : e.adresse.ville)
From e in Les_Etudiants

//Requête 2
Group e in Les_Enseignants
by (débutant : e.salaire < 10000,
    moyen : e.salaire >= 10000 and e.salaire < 20000,
    ancien : e.salaire >= 20000)
```

FIG. 5.18: *Requêtes constructives.*

semble des `Enseignants` dont le nom commence par `A` et qui enseignent plus de 2 cours » ; « ensemble des `Étudiants` dont le numéro de carte est supérieur à `125` et qui sont inscrits dans le cours de numéro `45` ». Les clauses `From` et `Where` peuvent utiliser des *expressions de chemins*, c'est-à-dire des séquences de chemins suivis au travers du graphe de composition des classes ou de la hiérarchie de généralisation/spécialisation du schéma (`e.adresse.ville` est une expression de chemins), et des *opérateurs agrégats* prédéfinis (`Count`, `Average`, `Sum`).

- Les requêtes *constructives* dont le résultat est une nouvelle structure construite.

  Dans la figure 5.18, la requête 1 correspond à la demande « afficher l'ensemble des renseignements, `nom` et `ville`, pour tous les `Étudiants` de la base » ; la requête 2 correspond à la demande « partitionner l'ensemble des `Enseignants` de la base selon la valeur de leur salaire en trois catégories désignées par `débutant`, `moyen`, `ancien` » ; elle utilise l'opérateur `Group`.

  Le résultat de toute requête constructive est une collection qui peut, à la manière des requêtes dans les logiques de description, constituer un nouveau « concept » (cf. *chapitre 11*).

Le type du résultat de toute requête est inférable (par le programmeur) de l'expression de celle-ci et correspond à un des types construits potentiels du langage de programmation. Par suite, le problème de dysfonctionnement (*impedance mismatch*) (cf. paragraphe § 5.2.4) est supprimé. Toute requête peut être plongée dans le langage de programmation, le résultat pouvant être affecté à une variable collection de même type. L'optimisation des requêtes devient, de ce fait, non seulement un problème lié à l'optimisation du langage de requête, mais plus globalement un problème d'optimisation du langage de programmation.

### Transactions et répartition

Pour tout SGBD, la ou les bases doivent être mises à la disposition de plusieurs utilisateurs. De plus l'évolution des systèmes tend à prendre en compte la répartition des données et des traitements.

*Transactions*

En phase d'exploitation, les utilisateurs accèdent aux objets via des applications. Une application est composée d'une ou de plusieurs *transactions*. Une transaction est donc définie comme une unité logique de traitement qui, appliquée sur un état *cohérent* de la base, doit restituer un nouvel état cohérent (*a priori* différent du premier). Toute transaction ne peut être qu'exécutée complètement (elle est alors dite *validée*) ou bien elle est *annulée*. Pour les SGBD relationnels, les *transactions* sont définies selon un *modèle plat* : une transaction ne peut contenir une autre transaction. Le modèle de transaction plat garantit les propriétés d'atomicité, de cohérence, d'isolation et de durabilité (ACID) pour chaque transaction effectuée sur la base [Bernstein *et al.*, 1987]. L'*atomicité* signifie que toute transaction constitue un tout et ne peut qu'être validée ou annulée ; la *cohérence* est relative à la définition d'état cohérent de la base sur lequel la transaction agit et qui doit être restitué cohérent ; l'*isolation* signifie que toute transaction doit avoir l'impression d'être seule à accéder aux données (pour cela on utilise des mécanismes de *verrouillage*) ; enfin la *durabilité* exprime le fait que les effets d'une transaction validée perdurent sur la base (un mécanisme de reprise sur faute [9] doit être mis en œuvre).

Le niveau du schéma et le niveau de la base peuvent être soumis à la *concurrence* des utilisateurs :

- plusieurs utilisateurs peuvent accéder au même schéma et la concurrence porte alors sur l'accès aux classes,
- plusieurs utilisateurs peuvent accéder à une même base et la concurrence porte alors sur l'accès aux objets.

Dans le contexte objet, la notion de transaction élargit la notion classique de transaction définie pour les SGBD relationnels. Le modèle de transaction plat s'avère mal adapté. Les objets complexes et composites ont des structures hiérarchiques et peuvent correspondre à de grands volumes de données. Les nouveaux besoins, en terme d'applications, nécessitent des transactions longues (pouvant durer plusieurs heures voire plusieurs jours) :

- annuler toutes les mises à jour, lors d'une panne ou d'une erreur, devient trop contraignant : il faut alors relâcher l'atomicité ;
- bloquer une transaction concurrente sur une ressource verrouillée par une transaction longue devient, de même, trop pénalisant : il faut relâcher l'isolation.

Les transactions portant sur des structures hiérarchiques (graphe des classes, graphe d'objets) demandent des modèles plus adaptés (modèle de transactions hiérarchiques ou emboîtées [Moss, 1985 ; Beeri *et al.*, 1989], modèles de transactions multi-niveaux). Diverses adaptations des protocoles de concurrence sont alors nécessaires [Cart et Ferrié, 1989 ; Delobel *et al.*, 1991 ; Gray et Reuter, 1993 ; Cattell, 1997].

9. Sous le terme générique de « faute » on englobe panne et défaillance de programme.

*Répartition*

Une *base de données répartie* est une « collection de données logiquement corrélées et physiquement réparties sur plusieurs machines interconnectées par un réseau de télécommunication ». C'est un ensemble de bases de données gérées par un ou plusieurs SGBD, chacune résidant sur un site différent, vu et manipulé par l'utilisateur comme une base centralisée [Gardarin et Valduriez, 1990].

Pour réaliser cet objectif une base de données répartie doit fournir divers niveaux d'indépendance :

- *Indépendance vis-à-vis de la localisation.* L'utilisateur de la base ignore où se trouvent les données et les requêtes qu'il peut effectuer ne comportent aucune indication de localisation [10].
- *Indépendance vis-à-vis de la fragmentation.* Pour des raisons d'efficacité, il est parfois souhaitable de diviser les objets persistants en plusieurs collections stockées sur des sites différents. L'indépendance vis-à-vis de la fragmentation des données cache à l'utilisateur le fait que les objets sont fragmentés.
- *Indépendance vis-à-vis de la duplication.* La duplication des fragments stockés sert à augmenter la fiabilité et la disponibilité des données ainsi que les performances d'accès. L'indépendance vis-à-vis de la duplication des données rend la duplication invisible à l'utilisateur.
- *Indépendance vis-à-vis des* SGBD. L'indépendance vis-à-vis des SGBD doit masquer le fait que les SGBD locaux peuvent être différents.
- *Autonomie des sites.* L'autonomie des sites permet à chacun de contrôler et de manipuler ses données locales indépendamment des autres sites.

Les problèmes divers soulevés par la répartition (cf. *chapitres 6 et 7*) ne sont pas encore totalement résolus dans le contexte relationnel et la complexité est accrue dans le contexte objet [Kemper et Moerkotte, 1994 ; Cattell, 1997].

## 5.4 SGBDO et normes

### 5.4.1 SGBDO

Nous citons ci-dessous, sans cependant vouloir être exhaustifs, divers systèmes existants à l'heure actuelle et qui ont contribué au développement de l'approche objet pour les bases de données. Nous donnons pour chacun quelques unes des caractéristiques illustratives des divers points abordés dans la section précédente.

- GEMSTONE de GemStone Systems [Bretl *et al.*, 1989]. Historiquement, les concepteurs de Gemstone ont commencé par définir un modèle de données muni d'une algèbre opératoire (*Set Theoric Data Model*) puis ont tenté de plonger ce modèle dans un langage de programmation général PASCAL. Les difficultés rencontrées et l'analogie entre le modèle décrit et les langages de

10. Ce n'est qu'une extension à l'indépendance physique.

programmation par objets, ont conduit au choix du langage à objets SMALLTALK qui a dû, dans le cadre de ce projet, être enrichi de fonctionnalités propres aux SGBD. Gemstone peut gérer de grandes quantités de données et la manipulation de ces données est effectuée par l'intermédiaire du langage de requêtes OPAL, fruit de la confrontation de l'algèbre de STDM avec SMALLTALK.

- ITASCA version commerciale dérivée d'Orion Itasca Systems Inc [Itasca Systems, 1993]. Le prototype du SGBD ORION initial a été conçu à partir du langage LISP [Kim *et al.*, 1989]. Le langage de manipulation est basé sur l'utilisation de prédicats utilisés à l'intérieur de requêtes correspondant à l'envoi de messages de type `Select` vers un objet de type `Ensemble` correspondant à une classe du système. L'interfaçage avec C, C++, CLOS, ADA et LISP est actuellement disponible.

- $O_2$, initialement développé par le consortium de recherche GIP ALTAIR [Bancilhon *et al.*, 1988], est depuis 1991 développé et maintenu par $O_2$ Technology puis récemment par Unidata Inc. Ce SGBDO offre un ensemble d'outils d'environnement et de développement ($O_2$Tools, $O_2$Graph, $O_2$Kit, $O_2$Look), un noyau SGBD ($O_2$Engine), un langage de programmation spécifique $O_2$C (langage dit *langage de 4ème génération, L4G*), un langage de requête $O_2$SQL et divers traducteurs permettant l'intégration avec C++, LISP, SMALLTALK et JAVA. Le modèle de données formel est basé sur la distinction entre types et classes. La conception d'un schéma de base de données se fait incrémentalement ; l'encapsulation est réalisable au niveau des classes et du schéma (export/import) et la persistance est assurée par nommage de racines de persistance.

- ONTOS, système développé par Ontologic (Billerica, Massachussets) sous le nom de Vbase, est un système multi-utilisateur gérant des objets écrit en C++ [Andrew et Harris, 1987]. Le modèle des objets est à typage fort, mais l'encapsulation n'est pas respectée. Les méthodes ne sont pas stockées dans la base. La persistance est assurée par héritage de la classe racine `Entity`. Les accès aux objets sont réalisés par l'intermédiaire d'un langage de type SQL (OBJS).

Bien d'autres systèmes sont disponibles (OBJECTSTORE, VERSANT, MATISSE, OBJECTIVITY/DB, etc.) la plupart sous Unix et en environnement client/serveur. Dans le domaine de la micro-informatique, les SGBD actuels (ACCESS, DBASE, FOXPRO, etc.) intégrent des caractéristiques objets mais le plus souvent au niveau du langage de programmation des applications, le modèle sous-jacent reste relationnel.

### 5.4.2 Les standards

La standardisation a toujours été importante pour les SGBD, car il semble naturel d'assurer à tout concepteur son indépendance par rapport aux systèmes propriétaires. SQL et ODBC offrent portabilité et interopérabilité pour les SGBD relationnels[11].

### OMG et ODMG

L'*Object Management Group* (OMG) est un consortium qui a pour objectif la définition d'une infrastructure à objets ouverte (cf. *chapitre 4*). Les composants CORBA (*Common Object Request Broker Architecture*) de cette infrastructure ont une interface décrite par IDL (*Interface Definition Language*), langage déclaratif permettant de spécifier l'interface des objets.

L'*Object Database Management Group* (ODMG) créé en 1991 par R. Cattell, regroupe divers participants représentant des grandes sociétés développant des SGBDO (Object Design, Ontos, $O_2$ Technology, Versant, Objectivity, Poet Software, Itasca, Servio logic, ADB), des constructeurs informatiques (Digital Equipement, Hewlett-Packard, Texas Instrument, Sunsoft, etc.). Ce groupe, affilié à l'*Object Management Group*, s'est fixé comme objectif la publication du standard ODS (*Object Database Standard*) pour les SGBDO.

Le standard [Cattell *et al.*, 1994] comporte un modèle objet (OM) basé sur celui proposé par l'OMG, un langage de définition des données (ODL) sur-ensemble d'IDL, un langage de requêtes (OQL), la liaison avec divers langages de programmation par objets (SMALLTALK, C++, JAVA, etc.).

- **Modèle Objet**

  Le modèle d'ODMG présente de nombreux points communs avec ceux décrits dans la section précédente. La principale originalité est la vision proposée qui s'appuie sur les types abstraits de données (TAD). Les types définis dans le modèle sont scindés en `Literal_type` (correspondant à des *valeurs*) et `Object_type` (correspondant aux *objets* proprement dits).

  Les propriétés d'un TAD sont séparées en deux catégories :

  – *les propriétés de type* partagées par tous les objets du type concerné. Parmi celles-ci, `extent`, qui permet de gérer l'extension du type, et `key`, qui permet de rajouter la notion de clé d'objet correspondant à l'unicité de valeurs pour les propriétés d'instance associées,

  – *les propriétés d'instance*, qui peuvent être des attributs ou des opérations. Les attributs ont un domaine associé qui peut être un `litteral` (valeur) ou une `relation` (référence vers un type objet ou un type collection d'objets). Les opérations sont définies par une signature et peuvent être complétées par des exceptions.

  Les classes sont ensuite définies par la spécification correspondant au TAD (interface) et une implémentation[12].

11. Des SGBD interopérables sont des systèmes hétérogènes mais qui peuvent communiquer entre eux pour échanger des données : exemple ORACLE avec ACCESS.
12. On peut donner plusieurs implémentations à une même interface. Un type abstrait n'a pas d'implémentation.

```
Interface Cours : Object {
//type properties
    extent Les_Cours;
    keys numéro;
// instance properties
    attribute string numéro;
    attribute integer volume_horaire;
    relationship Set<Etudiant> est_suivi_par
    inverse Etudiant :: cours_suivis;
    relationship Enseignant est_enseigné_par
    inverse Enseignant :: cours_enseignés;
    .......
// instance operations
    Creer (in string nc) raises (existant);
    Supprimer (in string nc) raises (non existant);
    ...........}
```

FIG. 5.19: *Définition d'un exemple d'interface en* ODL.

- **Le langage de description des données ODL**

  La syntaxe ODL est basée sur IDL (*Interface Definition Language*). ODL n'est pas un langage de programmation mais un langage de spécification qui permet de décrire l'ensemble du schéma des TAD de la base. Dans l'exemple ci-dessous, on décrit l'interface de la classe `Cours` de la figure 5.4 selon la syntaxe ODL.

  L'implantation d'un schéma spécifié en ODL dans un SGBDO se fait ensuite par précompilation du source ODL vers le langage de définition cible du SGBDO concerné.

- **Le langage de manipulation OQL**

  Le langage OQL est très proche du langage de requête $O_2$SQL proposé par $O_2$. La dernière version du SGBD $O_2$ présente OQL comme langage de requête. Il permet de définir les divers types de requêtes, que nous avons présentées dans le § 5.3.3 et qui peuvent être, si besoin, « encapsulées » dans un langage de programmation par objets.

### SQL3

SQL3 intègre des aspects objets (après la norme SQL éditée en 1986, puis SQL2 éditée en 1989 et 1992) dans le modèle relationnel classique. Pour cela SQL3, supporte les types abstraits de données, l'identité d'objet, l'héritage mais incorpore aussi divers aspects de programmation (structures itératives et opérations récursives comme `While` et `For` par exemple). Un TAD pour SQL3 spécifie des attributs (plus ou moins encapsulés selon le choix de leur déclaration `public`, `private`, `protected`) et des routines (constructeurs, destructeurs et acteurs). Les valeurs sont déclarées par des `TAD without oid`. Nous retrouvons ici, la notion de valeur sans identité du § 5.3.2.

Comme dans tous les autres SQL, on retrouve la notion de *table* correspondant à la description de la structure d'un schéma relationnel. TAD et tables devraient être totalement intégrés, mais la norme n'est pas encore définitive à l'heure actuelle. Les deux standards proposés (ODMG et SQL3) reflètent une volonté de donner aux utilisateurs de SGBDO des langages intégrant les aspects langage de programmation par objets et LDD, LMD des SGBD. On peut cependant s'interroger sur la convergence hypothétique de ces deux standards.

## 5.5 Problèmes et perspectives

Dans la section 5.3.1, nous avons montré que la phase de développement d'une base de données débouche sur la production d'un schéma conceptuel, de schémas dérivés et d'applications. En phase d'exploitation, une base de données objet est perçue et, par suite, manipulée de diverses manières par plusieurs personnes dont les rôles peuvent être catégorisés :

- *l'administrateur de la base* maintient la cohérence des divers schémas (schéma conceptuel et schémas dérivés) ainsi que l'organisation physique sous-jacente,
- *les concepteurs d'applications* développent et maintiennent les diverses applications dédiées aux besoins des utilisateurs finaux,
- *les utilisateurs finaux* accèdent à la base par l'intermédiaire d'applications existantes, ou formulent directement, grâce au langage de requête, des recherches spécifiques.

En termes de fonctionnalités, chaque catégorie d'utilisateurs a des exigences diverses et spécifiques, mais tous espèrent bénéficier de services les moins contraignants possibles. Les auteurs du manifeste des bases de données objet (cf. § 5.3.1), suggéraient, au delà des *règles d'or*, des *règles optionnelles* (extensibilité du modèle, versions, etc.) qui donneraient aux systèmes de nouvelles possibilités en terme d'*adaptabilité*. Un système adaptatif (évolutif) est un système qui « absorbe » au cours du temps les divers événements de la réalité. Sous le vocable d'évolution, les travaux en bases de données objet traitent de divers problèmes rencontrés par le fait que le monde réel représenté et les entités de ce monde changent au cours du temps. Cette problématique, qui reste omniprésente pour les SGBD relationnels et déductifs, constitue un axe de recherche fécond dans l'approche objet pour les bases de données. Pour simplifier le propos, on distingue généralement :

- *Les problèmes d'évolution de schéma* : la phase de conception d'un schéma conceptuel passe par des affinements successifs ; le concepteur doit pouvoir manipuler et modifier le schéma.
- *Les problèmes d'évolution de données* : ceux-ci, plus classiques, correspondent à des *mises à jour* plus ou moins complexes sur les données stockées dans la base.

- *Les problèmes d'évolution globale schéma - données* : en phase d'exploitation, la notion d'évolution recouvre les changements concernant le niveau du schéma et les conséquences qui en découlent au niveau des données corrélées. Il faut choisir la ou les stratégies de propagation des mises à jour depuis le niveau schéma jusqu'aux instances.
- *Les problèmes de multi-expertise* : un même ensemble de données est accessible, mais il est raisonnable de présenter ces données sous des points de vue variés. La base de données est alors filtrée ce qui permet d'exprimer de multiples expertises d'une part (cf. *chapitre 10*) et plus prosaïquement de régler des problèmes de sécurité.

La dimension temporelle joue un rôle prépondérant dans le processus général. Elle est intimement liée à la sémantique des évolutions souhaitées (oubli du passé, maintien du passé, voire réversibilité) [Libourel, 1993]. Traditionnellement, la temporalité est prise en compte de deux manières différentes :

- *Le temps est implicite* : soit la base ne donne alors accès qu'à la dernière vision instantanée et cohérente des informations tant au niveau des instances qu'au niveau du schéma, soit on conserve la trace des évolutions, en admettant que les divers états observés sont chronologiques [Cellary et Jomier, 1990 ; Gançarski et Jomier, 1994].
- *Le temps est explicité* : il faut alors le modéliser et, de plus, il faut préciser quelle sorte de temps on veut prendre en compte. Les bases de données *temporelles* [Tansel *et al.*, 1993 ; Snodgrass et Ahn, 1986] distinguent les notions de temps *valide* (temps de la réalité du phénomène) et de temps *transactionnel* (temps auquel le phénomène est enregistré dans la base).

### 5.5.1 L'évolution de schéma en phase de conception

Les systèmes relationnels définissent la notion de *méta-base* (ou dictionnaire des données). La méta-base contient la description de tous les schémas relationnels du schéma conceptuel y compris sa propre description. L'administrateur de la base dispose d'opérations de modification (ajout, suppression, modification) qui, effectuées sur la méta-base, permettent d'effectuer les évolutions (certes restreintes) souhaitées du schéma conceptuel.

Dans l'approche objet, de très nombreux travaux traitent de l'*évolution de schéma* [Skarra et Zdonik, 1986 ; Penney et Stein, 1987 ; Nguyen et Rieu, 1989 ; Kim, 1990]. Les principaux SGBDO (GEMSTONE [Penney et Stein, 1987], ORION [Banerjee *et al.*, 1987], ENCORE [Skarra et Zdonik, 1986], etc.) proposent, sur les traces de leurs ancêtres relationnels, un ensemble d'opérations primitives souhaitables pour traduire la sémantique de l'évolution :

- Les modifications relatives à la *définition d'une classe*, qui comprennent des modifications *structurelles* (ajout, suppression, modification et renommage d'attributs) et des modifications *comportementales* (ajout, suppression, modification et renommage de méthodes).

- Les modifications relatives à la *définition du graphe d'héritage*, qui comprennent des opérations sur le graphe : opérations sur les nœuds (ajout, suppression, renommage d'une classe) et opérations sur les arcs (ajout, suppression de lien d'héritage).

Les opérations d'évolution sont contrôlées par un ensemble de règles et d'invariants relatifs à la sémantique du schéma et qui devront être préservés lors de toute évolution afin d'assurer la consistance du schéma (cf. § 5.3.2).

Les principaux invariants définis explicitement sont :

- L'invariant du schéma : le schéma est un graphe orienté sans circuit.
- L'invariant d'unicité des noms : toutes les classes ont un nom distinct, les attributs ou les méthodes définies ou héritées dans une classe ont des noms distincts (hypothèse du nom unique).
- L'invariant d'origine distincte : tous les attributs ou les méthodes d'une classe doivent avoir des origines distinctes (tout attribut ou toute méthode hérité doit dériver de façon déterministe d'une seule classe).
- L'invariant d'héritage complet : chaque classe hérite de tous les attributs et méthodes de ses super-classes, sauf s'ils sont redéfinis.

Toute évolution doit prendre un schéma en entrée et restituer un schéma en sortie, et doit respecter la totalité des invariants définis ci-dessus. Le système contrôle l'évolution en appliquant des règles heuristiques qui résolvent automatiquement toute atteinte aux invariants ou bien annulent l'évolution source d'inconhérence.

D'autres approches automatisables existent notamment dans le domaine des langages de programmation par objets, où la restructuration de hiérarchies obéit à des règles de qualité (factorisation maximale et réutilisabilité). L'évolution des hiérarchies est alors réalisée par des algorithmes qui relèvent de la *classification* [Bergstein et Lieberherr, 1991 ; Godin *et al.*, 1995 ; Moore et Clement, 1996 ; Dicky *et al.*, 1996 ; Moore, 1996] (cf. *chapitres 10 et 12*).

Dans certains de ces travaux [Lieberherr et Holland, 1989], au-delà des opérations de base répertoriées ci-dessus, certaines opérations plus complexes sont définies sur les classes (fusion, éclatement, agrégation, etc.) ou sur le graphe lui-même (insertion, suppression de sous-hiérarchie, etc.).

La souplesse des solutions proposées est liée à la variété des opérations de modification disponibles et à l'aide éventuelle fournie quant à la pertinence de ces opérations (c'est-à-dire à l'interaction avec le concepteur).

### 5.5.2 L'évolution des données

Les mises à jour (effectuées à schéma conceptuel constant) peuvent correspondre à de simples modifications de l'état des objets, mais peuvent aussi s'étendre à d'éventuelles *migrations* d'objets [Mendelzon *et al.*, 1994]. Le problème des migrations d'objets est particulièrement difficile à prendre en compte, pour des systèmes où la flexibilité des algorithmes de classification des objets offerts par les systèmes classificatoires sont absents (cf. *chapitres 10 et 12*). Faire migrer un objet (qui de plus est persistant) nécessite la mise à jour de tous les objets le référençant, ainsi que celles de tous les objets référencés.

Dans certaines applications spécifiques comme le génie logiciel ou la CAO, la trace de l'évolution d'un objet est une source précieuse de renseignements. Divers travaux proposent pour conserver cette trace de recourir à un mécanisme de *versions*. Les approches dites *macroscopiques* définissent le contexte ou l'environnement de travail (instances et classes) comme une *version de la base de données* [Cellary et Jomier, 1990; Gançarski et Jomier, 1994]. L'utilisateur peut ensuite décider de *dériver* ce contexte en une nouvelle version dans laquelle il pourra ajouter, supprimer, modifier les éléments à sa guise. Chaque version de contexte, de manière interne, est *estampillée* (numérotée et datée par le système) ; pour chacune on conserve les nouveaux objets apparus, les variations de valeurs des objets modifiés ou les références vers les objets stables (c'est-à-dire ceux qui existaient dans la version précédente et n'ont subi aucun changement).

### 5.5.3 Les évolutions globales schéma - données

Les problèmes soulevés ici sont plus complexes car l'évolution d'un schéma *instancié* (c'est-à-dire ayant des instances stockées dans la base) doit respecter la cohérence du schéma (cf. § 5.5.1) mais aussi la consistance des instances vis-à-vis de leur schéma. Les solutions proposées sont nombreuses et diverses. La plupart [Skarra et Zdonik, 1986; Penney et Stein, 1987; Kim et Chou, 1988] sont basées sur les principes définis au § 5.5.1:

- une taxonomie d'opérations de base réalisables au niveau du schéma,
- un ensemble d'invariants relatifs aux entités du schéma,
- un ensemble de règles qui permettrons le contrôle du respect des invariants lors de toute évolution.

Les évolutions effectuées au niveau du schéma doivent se répercuter sur les objets de la base. En phase de conception, la signification des opérations d'évolution du schéma est bien perçue par le concepteur; en phase d'exécution, par contre, il est souhaitable de la compléter, en précisant quelles conséquences aura l'opération choisie, au niveau des instances du schéma. Par exemple, la suppression d'une classe dans le schéma entraîne-t-elle la suppression de tous les objets instances en dépendant ou bien doit-on migrer ces objets vers des super-classes ou des sous-classes de la classe concernée ? Ces conséquences peuvent avoir un coût prohibitif car les modifications effectuées ont nécessairement un impact non négligeable sur les performances du système (pour réaliser la propagation sur les instances des modifications effectuées sur le schéma des phases de « verrouillage » ou d'arrêt de la base de données seront nécessaires). Un autre aspect important du problème, concerne le délai de la modification effective sur les instances : la répercussion de toute opération de mise à jour du schéma vers les instances entraîne un choix de stratégie. La propagation peut être :

- *automatique* (ORION [Kim *et al.*, 1990], GEMSTONE [Penney et Stein, 1987]) grâce à des transformations (par défaut) qui s'appuient sur les règles et invariants définis par le système. Plus précisément, elle peut être, soit *retardée*, la modification n'est effective que lorsque l'objet concerné est utilisé (ORION), soit *immédiate* (GEMSTONE). La propagation immédiate contribue au maintien de la cohérence au dépens des performances, car la base devient moins disponible pendant l'opération de modification des instances.
- *manuelle*, réalisée par l'utilisateur avec des fonctions de conversion appropriées, c'est le cas dans les systèmes ENCORE ou $O_2$ [Ferrandina *et al.*, 1995].

Quelle que soit la stratégie choisie, rares sont les applications qui peuvent indifféremment s'appliquer sur le schéma et les instances initiaux, puis sur le schéma et les instances modifiés. Toute évolution entraîne donc une mise à jour quasi obligatoire de tous les applicatifs.

Certaines solutions proposent alors d'avoir recours aux mécanismes de versions. Les approches *microscopiques* définissent les notions de versions de classe et de versions de schéma. Parmi ces approches nous pouvons citer celles réalisées dans le cadre du développement des systèmes ORION [Kim et Chou, 1988], ENCORE [Skarra et Zdonik, 1986], AVANCE [Bionnerstedt et Brigitts, 1988], OTGEN [Lerner et Habermann, 1990], CLOSQL [Monk et Sommerville, 1993]. Le mécanisme de versions permet de maintenir anciennes et nouvelles versions et de tendre vers une meilleure *transparence* de l'évolution globale : les applications accèdent aux objets au travers des diverses versions existantes.

D'autres solutions *mixtes* ou *par compromis* combinent les mécanismes de modification et de versions [Benatallah, 1996] au niveau schéma et suggèrent pour la propagation au niveau instance des combinaisons de stratégies consistant à *adapter* les instances.

### 5.5.4 Vues et relativisme sémantique

Le mécanisme de *vues relationnelles* qui assurait l'indépendance logique (cf. § 5.2.2) est introduit dans le contexte objet par différents auteurs comme un moyen de traduire différentes perceptions du même schéma conceptuel [Abiteboul et Bonner, 1991 ; Tanaka *et al.*, 1988 ; Tan et Katayama, 1989].

Les concepteurs d'applications connaissent le schéma conceptuel de base, mais il est préférable (pour des raisons de points de vue, de confidentialité et de sécurité) de ne fournir aux utilisateurs finaux que des applications s'appuyant sur des schémas externes présentant les données de différentes manières. Le schéma *virtuel* (externe ou dérivé) est défini, dans l'approche objet comme dans l'approche relationnelle, à partir de requêtes [13] formulées dans le langage d'interrogation et définissant des classes *virtuelles* [14]. Comme dans l'approche relationnelle, un schéma virtuel décrit uni-

13. Dans le contexte objet, ces requêtes exploitent les possibilités d'héritage et de surcharge du modèle.
14. Cette notion est totalement différente de celle de classe virtuelle de C++.

```
virtual schema Origine_Cursus from schema Gestion_Universitaire
import class Personne, Etudiant, Cours
virtual Class Etudiant_Parisiens as
   (select e from e in Les_Etudiants
   where e.adresse.ville="Paris")
End Etudiant_Parisiens
virtual Class Etudiant_Provinciaux as
   (select e from e in Les_Etudiants
   where e.adresse.ville<>"Paris")
End Etudiant_Provinciaux
```

FIG. 5.20: *Schéma virtuel Origine_Cursus.*

quement l'intension des classes virtuelles exprimée dans l'expression des requêtes ; l'extension de ces classes est obtenue après exécution des requêtes et peut correspondre à une sélection d'objets préexistants ou à la création de nouveaux objets. Par exemple, un schéma virtuel `Origine_Cursus`, construit sur le schéma conceptuel de la figure 5.4, pourrait restructurer le schéma de base en ne s'intéressant qu'à la situation locale des Etudiants.

La richesse d'un modèle de vues est étroitement lié à la puissance du langage de définition des vues [Rundensteiner, 1992 ; Souza dos Santos, 1995 ; Lacroix *et al.*, 1997]. Le mécanisme de vues a aussi été proposé, comme alternative dans le traitement des évolutions globales schéma-données évoquées dans le § 5.5.3. Il permet d'*émuler* les évolutions, en évitant les inconvénients causés par les modifications directes du schéma [Laasch *et al.*, 1991 ; Bertino, 1992 ; Tresch et Scholl, 1993 ; Bellahsene *et al.*, 1996 ; Ra et Rundensteiner, 1997]. De nombreuses questions restent encore *ouvertes* : les mises à jour au travers des vues sont-elles toutes envisageables sans matérialiser les vues ? Dispose-t-on de possibilités pour *inférer* le positionnement des classes virtuelles par rapport aux classes du schéma initial [Rundensteiner, 1994] (cf. *chapitre 12*) ?

## 5.6 Conclusion

À l'heure actuelle, les SGBDO ont atteint une large partie des objectifs qui leur avaient été fixés :

- ils assurent une meilleure représentation de la réalité en offrant des modèles de données complexes,
- ils permettent une plus grande facilité pour l'écriture et la maintenance d'applications grâce à la modularité et à la réutilisabilité héritées de la programmation par objets,
- ils s'avèrent efficaces pour de nombreux domaines d'applications (Biologie, Médecine, Multimedia, etc.),

La variété des directions dans lesquelles les travaux de recherche autour des SGBDO sont menés actuellement peut être synthétisée dans la figure 5.21. Chaque direction

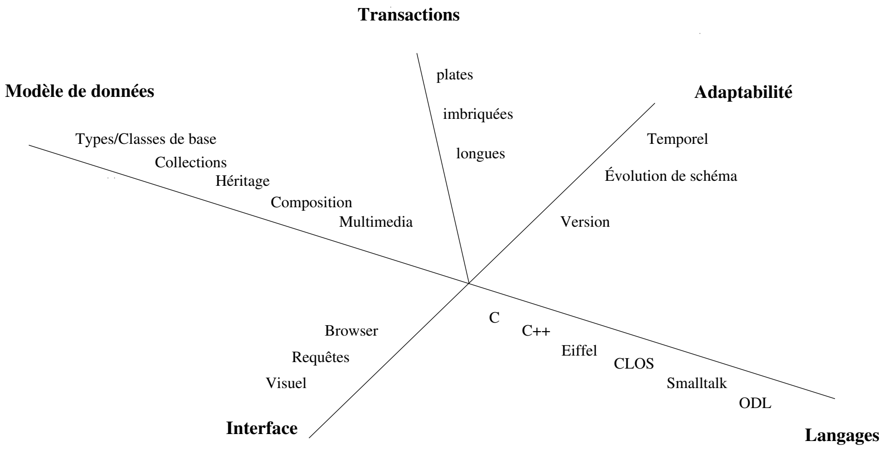



FIG. 5.21: *Les divers axes de recherche relatifs aux* SGBDO.

constitue un aspect différent, qui contribue aux réflexions théoriques et pratiques menées autour des SGBDO, et évolue en intégrant les divers concepts relatés en légende.

Cependant de nombreux points restent dans l'ombre et soulèvent encore un certain nombre de critiques ou de réticences :

- Le manque de théorie, de formalisation des modèles proposés. À quand une théorie de la normalisation pour les schémas objets ?
- Le concept de *relation* du modèle relationnel est enfoui dans le modèle objet (ou du moins ne constitue plus l'unité de base). La simplicité de la relation et l'efficacité de ses traitements dans le monde des SGBD semble faire défaut, ce qui donne, à l'heure actuelle, un regain d'intérêt (du moins dans la pratique) pour les couplages OO-Relationnel [Stonebraker, 1987 ; Gardarin, 1994], qui pourtant étaient fort décriés auparavant.
- Le manque de déclarativité. Les langages de requêtes ont gagné en complétude mais ont perdu une partie du côté déclaratif des calculs relationnels. Enfin la possibilité d'introduire des contraintes d'intégrité ou des règles de déduction au sein des SGBDO reste confidentielle.

Un des défis primordiaux reste le problème de la distribution, de l'interopérabilité, de la visibilité des grands réservoirs de données accessibles au travers des divers réseaux de communication.

# Deuxième partie

# Développements avancés de la notion d’objet

# 6

# Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances *

CE CHAPITRE ÉTUDIE COMMENT LES NOTIONS et les acquis des objets ont été utilisés dans le contexte du parallélisme et de la répartition. Nous distinguons l'approche applicative, l'approche intégrée, et l'approche réflexive. L'approche applicative utilise les concepts de la programmation par objets, en tant que tels, pour la structuration de systèmes informatiques parallèles et répartis sous la forme de bibliothèques. L'approche intégrée vise à la fusion plus ou moins aboutie de concepts tels que : objet et activité, transmission de message et transaction. L'approche réflexive a pour objectif d'intégrer des bibliothèques de protocoles avec un langage de programmation.

## 6.1 Introduction

Bien que la majorité des ordinateurs et des programmes restent éminemment séquentiels, la prise en compte du parallélisme et de la répartition s'impose peu à peu et de manière inéluctable. Des domaines d'application représentatifs sont par exemple : le travail coopératif assisté par ordinateur [Ellis *et al.*, 1991] et les jeux électroniques multi-utilisateurs. La popularité du réseau Internet et des applications associées (telles que le « World Wide Web » (WWW) [Berners-Lee *et al.*, 1992]) laisse prévoir un engouement vers le développement de divers services répartis. Les concepts de la programmation par objets offrent de bonnes bases pour aborder ces nouveaux enjeux. Les concepts sont en effet à la fois assez forts pour assurer structuration, modularité et encapsulation, et assez souples pour recouvrir également granularités variées, hétérogénéité éventuelle et enfin de nombreux protocoles spécialisés. Ce n'est ainsi pas un hasard si la quasi-totalité des architectures

* Article original publié dans : ©*Technique et Science Informatiques*, vol. 15, n$^o$ 6, Juin 1996. Republié avec l'accord gracieux d'AFCET – HERMES.

de systèmes d'exploitation répartis développées actuellement se fondent sur la notion d'objet.

Il est intéressant de noter que lors de la genèse des concepts objets à la fin des années 60, ces concepts n'étaient pas spécifiquement restreints à la programmation séquentielle. Cependant les contraintes technologiques, et sans doute également culturelles, de l'époque, ont alors concentré les développements de la technologie objet dans le monde de la programmation séquentielle. Un premier enjeu est donc tout d'abord la redécouverte des concepts plus généraux, prenant en compte des activités et interactions potentiellement simultanées. Un deuxième enjeu, plus important encore, est l'intégration plus ou moins complète de protocoles et mécanismes pour gérer les différents aspects liés à cette simultanéité, et plus encore à la répartition des objets et leurs activités.

Ce chapitre a pour ambition de récapituler en quoi le concept d'objet est une bonne fondation pour la programmation parallèle et répartie, et de distinguer les tendances actuelles pour intégrer les nouveaux enjeux qui en découlent. Il est important de noter que, bien que certains exemples seront utilisés pour illustrer les différentes tendances, l'objectif de ce chapitre n'est pas de présenter exhaustivement les langages et systèmes à objets parallèles et répartis. En particulier, un article récent et complémentaire se concentre lui sur les différents modèles de programmation concurrente par objets [Guerraoui, 1995a]. De plus, deux ouvrages récents, [Guerraoui *et al.*, 1994] et [Briot *et al.*, 1996], regroupent un certain nombre de contributions dans le domaine de la programmation parallèle et répartie par objets. Enfin, il faut noter aussi que des aspects complémentaires, tels que : formalismes et preuves de programmes, visualisation et mise au point, méthodologies d'analyse et de conception, techniques de mise en œuvre et d'optimisation, ne seront pas abordés dans ce chapitre, malgré leur importance.

## 6.2 Enjeux et besoins

Du fait de l'extension continuelle de la portée des applications informatiques, les problèmes à traiter sont de plus en plus variés et complexes. De pair avec cette tendance, les utilisateurs deviennent de plus en plus exigeants sur la qualité, la sûreté et enfin l'efficacité des services offerts.

### 6.2.1 Enjeux

Un premier enjeu est le *parallélisme*. Une des motivations est le gain d'efficacité visé en exploitant les architectures de machines à plusieurs processeurs. Ce n'est cependant pas la seule motivation. Pour de nombreux problèmes, la formulation se fait de manière beaucoup plus naturelle sous la forme d'activités simultanées et coordonnées, que sous la forme d'un algorithme séquentiel. Il faut donc tout d'abord faciliter une expression *logique* (*concurrence*) du parallélisme potentiel au niveau du programme, puis une utilisation optimale des ressources matérielles disponibles pour mettre en œuvre cette concurrence sous forme de *parallélisme* (*physique*) et donc de gain de performance.

Un deuxième enjeu est la *répartition*. Certaines applications sont intrinsèquement réparties (par exemple le travail coopératif). D'autres, pour des raisons de partage des ressources (par exemple une imprimante), de tolérance aux fautes, ou de répartition de charges, requièrent des architectures réparties. Comme pour la gestion du parallélisme, il est nécessaire, d'une part de faciliter l'expression des programmes sur de telles architectures, et d'autre part d'offrir les mécanismes sous-jacents adéquats de contrôle. Parmi ces mécanismes, on peut trouver par exemple le *contrôle de concurrence* entre transactions [Bernstein *et al.*, 1987], et la tolérance aux fautes [Simons et Spector, 1987].

Enfin il est de plus en plus désirable de pouvoir changer les modèles et mécanismes d'un système informatique pour pouvoir intégrer de nouveaux services, partenaires, et besoins, *au cours* de son fonctionnement. Ceci est lié au concept de système *ouvert*. Internet est un exemple représentatif et à large échelle de système ouvert.

### 6.2.2 Besoins

Nous pouvons maintenant résumer brièvement les principaux besoins qu'il faut pouvoir exprimer et satisfaire au mieux, dans le cadre des systèmes informatiques parallèles et répartis[2] :

- **(Dé)composition** : fondations conceptuelles et structurelles pour une bonne décomposition logique et physique de l'application en composants logiciels autonomes et susceptibles d'activités indépendantes ; et à l'inverse, fondations pour la composition de modules logiciels développés indépendamment.
- **Coordination** : protocoles évolués de synchronisation et de coordination des différentes activités pour assurer leur consistance et leur conjugaison.
- **Modularité** : indépendance des différents composants pour assurer leur interchangeabilité et leur protection.
- **Réutilisation** : généricité et réutilisation des composants logiciels.
- **Extensibilité et adaptabilité** : possibilité d'ajout dynamique de nouveaux services, composants et protocoles, en cours de fonctionnement.
- **Hétérogénéité** : diversité éventuelle de composants logiciels et matériels, de même que des protocoles et mécanismes qui peuvent être offerts.
- **Répartition** : mécanismes abstraits de gestion de la répartition et la duplication, statique et éventuellement dynamique (migration), des entités et services.
- **Sûreté** : protections et preuves de propriétés au niveau logiciel, et tolérance aux fautes au niveau matériel.
- **Efficacité** : utilisation optimale des ressources offertes par les architectures matérielles parallèles et réparties.

2. Par *systèmes informatiques*, nous considérons ici : matériel (ordinateurs), logiciels (systèmes d'exploitation, langages et environnements de programmation), et applications. Notez que la plupart des besoins énumérés ici correspondent aux besoins classiques de génie logiciel. Les trois derniers besoins sont eux plus spécifiques.

### 6.2.3 Classes de systèmes

Cet énoncé de besoins est relativement générique. Certains systèmes et applications pourront se concentrer sur certains d'entre eux au détriment d'autres. Nous pouvons ainsi regrouper trois grandes classes de systèmes informatiques parallèles ou/et répartis :

| | *principal objectif* | *nombre de processeurs* | *structure* | *répartition des objets* | *tolérance aux fautes* |
|---|---|---|---|---|---|
| Multi-processeur à mémoire partagée | efficacité | dizaine | homogène | non | rare |
| Ordinateur massivement parallèle | efficacité | centaine ou plus | homogène | oui, mais surtout statique | rare |
| Informatique répartie | coopération | indéterminé | hétérogène | pré-existante et dynamique | indispensable |

## 6.3 Objets comme fondation

La méthodologie de programmation par objets offre d'excellentes fondations pour aborder les besoins exprimés ci-dessus comme nous allons le voir dans cette section.

### 6.3.1 Concepts et avantages

La programmation par objets repose sur quelques concepts simples et unificateurs. Un programme se décompose en un ensemble de modules autonomes, appelés *objets*, qui interagissent au travers d'un protocole de communication unifié, la *transmission de message*. Les objets correspondent à différentes entités des domaine et problème considérés. Un objet se présente comme une *capsule* contenant à la fois des données et des opérations opérant sur celles-ci. Le principe de transmission de message introduit une séparation entre la signature des opérations qui pourront être invoquées de l'extérieur (appelée *interface*) et la représentation interne de l'objet. Ceci est appelé le principe d'*encapsulation* et il favorise la séparation entre spécification et mise en œuvre. Il permet également abstraction et *généricité*. Ainsi, par exemple, l'opérateur addition (+) des objets numériques est naturellement générique, étant interprété différemment (addition entière, flottante, rationnelle, etc.) suivant le type de l'objet numérique qui reçoit une telle invocation.

Ces principes fondateurs de la programmation par objets permettent à la fois décomposition, modularité et protection. De manière à favoriser l'abstraction et la factorisation d'objets semblables, la plupart des langages et systèmes à objets reposent sur la notion de *classe*, comme abstraction et factorisation d'objets. De plus [3], des mécanismes de spécialisation, et en premier lieu l'*héritage*, permettent de réutiliser des propriétés déjà définies par certaines classes d'objets pour les étendre et/ou les modifier partiellement. Ces mécanismes augmentent encore la généricité et la réutilisation des programmes et ont permis de valider les intérêts de la programmation par objets au niveau du génie logiciel [Meyer, 1988]. Enfin, la programmation par objets intègre de manière naturelle une gestion dynamique des ressources, puisque l'on peut créer de nouveaux objets au fur et à mesure des besoins.

Comme nous l'avons souligné dans l'introduction, les concepts fondateurs de la programmation par objets sont à la fois suffisamment forts pour structurer et rendre modulaire les programmes, et en même temps suffisamment souples pour permettre une grande généricité des programmes. Ceci provient de la distinction entre interface (services offerts) et intérieur (représentation/mise en œuvre) d'un objet. Notez également que la granularité même de l'objet n'est pas fixée. Ce dernier point est particulièrement intéressant pour réaliser des architectures logicielles hétérogènes.

### 6.3.2 Concurrence potentielle

Les concepts fondateurs des objets, tels qu'ils ont été exprimés à la fin des années 60, contenaient déjà implicitement les promesses de la programmation concurrente. D'ailleurs, le premier langage de programmation à parler d'objets, SIMULA-67 [Birtwistle *et al.*, 1973], offrait déjà les germes d'une activité autonome et simultanée associée à chaque objet. Une classe SIMULA-67 peut définir un corps de programme (une séquence d'instructions appelée « *body* » qui sera exécutée par chaque objet lors de sa création). Ce corps de programme est plus qu'un simple texte d'initialisation puisque l'objet peut alors explicitement se suspendre et relancer un autre objet sous la forme de coroutines, permettant ainsi une exécution quasi-simultanée. Les objets se montraient ainsi dès l'origine proches de la notion d'activité simultanée (*processus*). Objets et processus partagent d'ailleurs de nombreuses caractéristiques communes (variables, données persistantes, encapsulation, moyens de communication) comme le souligne Bertrand Meyer [Meyer, 1993b].

Cependant, pour des raisons technologiques (les ordinateurs de l'époque étaient rarement parallèles) et culturelles (la programmation était construite sur la notion de séquence d'instructions, et elle le reste d'ailleurs encore majoritairement), cette dimension d'activité concurrente a plutôt régressé parmi les successeurs de SIMULA. Ces potentialités n'ont commencé à être réellement développées que plus tard, au cours du milieu des années 80, les langages d'acteurs [Lieberman, 1987 ; Agha, 1986] faisant figure de nouveaux pionniers.

### 6.3.3 Répartition potentielle

Un objet apparaît de manière naturelle comme une unité potentielle de répartition. La transmission de message assure en effet non seulement l'indépendance entre les

3. Peter Wegner [Wegner, 1987] a d'ailleurs proposé une approche terminologique par couches successives : *object-based* pour les langages et systèmes fondés sur la notion d'*objet*, *class-based* si l'on ajoute le concept de *classe*, et enfin *object-oriented* en présence du mécanisme d'*héritage*.

services offerts par un objet et sa représentation interne, mais également l'indépendance vis-à-vis de son *emplacement* sur tel ou tel site (processeur, mémoire, ou machine). Si l'objet appelant et l'objet appelé se trouvent sur deux sites différents, la métaphore de l'envoi de message prend alors tout son sens, un réel « message » étant transmis à travers le réseau. Enfin l'autonomie et relative « complétude » des objets (comme capsule de données et d'opérations associées) facilite la prise en compte de migration ou de duplication éventuelles.

Il est intéressant de noter que l'architecture *client/serveur*, à la base de la plupart des systèmes répartis ou d'interfaces homme-machine actuels, possède déjà une partie des caractéristiques « objet » [Nicol *et al.*, 1993]. Cependant la dichotomie client/serveur n'est pas statique en programmation par objets, puisque tout objet peut être considéré comme un client ou un serveur suivant qu'il envoie ou reçoit un message. De fait, la grande majorité des projets d'architectures réparties actuelles est fondée sur le concept d'objet. Le standard ODP (« Open Distributed Processing ») [Nicol *et al.*, 1993], censé définir un modèle général de programmation répartie, est d'ailleurs conçu selon le modèle objet. Enfin remarquons qu'il existe une double tendance : d'une part les chercheurs issus de la communauté « langages de programmation par objets » étendent les environnements de programmation vers des architectures réparties, et d'autre part les chercheurs issus de la communauté « systèmes d'exploitation » adoptent le modèle objet pour structurer les architectures de systèmes.

### 6.3.4 Limitations

Malgré les potentialités de ses concepts fondateurs, la technologie objet, développée quasi-exclusivement dans un contexte séquentiel, ne se transpose pas toujours parfaitement telle quelle à la programmation parallèle et répartie. Par exemple, le mécanisme d'héritage induit un certain nombre de limitations à l'égard du parallélisme comme de la répartition comme nous le verrons au § 6.6.6. Plus encore, le concept d'objet, en tant que tel, n'apporte pas des réponses complètes aux besoins énoncés (au § 6.2.2).

Il se trouve qu'en parallèle au développement de la technologie objet, diverses communautés, et en particulier : programmation parallèle, systèmes d'exploitation, et bases de données, ont abordé ces enjeux de parallélisme et de répartition. Ainsi différents types d'abstractions et de mécanismes, tels que synchronisation, « capabilities », et transactions, ont été proposés, pour aborder différents besoins et enjeux tels que contrôle de la concurrence, gestion des ressources, et tolérance aux fautes. Un enjeu d'importance est donc l'adaptation de tels acquis dans la technologie et la méthodologie objet. Il est également important de pouvoir réutiliser au mieux la technologie objet déjà validée, voire jusqu'à des modules/programmes déjà développés.

Toute la question est ainsi en conséquence : « De quelle manière doit-on lier les concepts objets aux enjeux et acquis de la programmation parallèle et répartie ? ». Autrement dit, doit-on procéder par application, identification, extension, évolution, etc. La suite de ce chapitre a justement pour ambition d'éclairer cette question et les tentatives actuelles pour y répondre.

## 6.4 Objets pour le parallélisme et la répartition : différentes approches

Pour utiliser les concepts objets dans le contexte de la programmation parallèle et répartie, plusieurs voies sont possibles, comme en témoigne la diversité des systèmes et projets proposés. Nous les regroupons ici en trois approches générales.

### 6.4.1 Approches

La première approche est une approche *applicative* (§ 6.5). Il s'agit d'appliquer, *tels quels*, les concepts objets à la conception de programmes et systèmes parallèles et répartis. Les différents composants de ces systèmes (processus, fichiers, serveurs de nom, etc.) seront alors représentés par différentes classes spécifiques d'objets. Ceci assurera la généricité des architectures logicielles. La programmation reste ainsi essentiellement de la programmation par objets séquentielle traditionnelle. On procède donc avec une approche d'extension par *bibliothèques* plutôt que par extension des concepts et des langages de programmation les incarnant.

La deuxième approche est une approche *intégrée* (§ 6.6). Il s'agit alors d'étendre les concepts fondateurs des objets pour y intégrer les enjeux de la programmation parallèle et répartie. Les systèmes qui suivent cette approche intègrent (unifient) souvent objet avec activité (la notion d'*objet actif*, voir au § 6.6.3), et la transmission de message avec différents protocoles de synchronisation (synchronisation client / serveur, transactions, etc., voir au § 6.6.4). Les intégrations ne se font cependant pas toujours sans mal, et certains aspects créent des conflits, par exemple : héritage et synchronisation, duplication et communication (voir au § 6.6.6).

La troisième approche est une approche *réflexive* (§ 6.7). Il s'agit de séparer le programme proprement dit, des différents aspects de sa mise en œuvre (modèle de calcul, de communication, de répartition, etc.), eux-mêmes décrits dans un *méta*-programme. La programmation s'applique ainsi au contrôle de sa propre mise en œuvre, d'où le nom donné de « *réflexion* ». La réflexion permet également d'abstraire la gestion des ressources (équilibre de charges, dépendance du temps, etc.) et de la décrire avec toute la puissance d'un langage de programmation.

### 6.4.2 Complémentarité

Il est important de souligner dès à présent que ces trois approches ne sont en concurrence qu'en apparence. Pour être plus précis, leurs développements respectifs ont des objectifs complémentaires. L'approche *applicative* est destinée aux concepteurs de systèmes et vise à identifier les abstractions fondamentales des systèmes informatiques parallèles et répartis. L'approche *intégrée* est destinée aux concepteurs d'applications, et vise à la définition d'un langage de programmation de haut niveau comportant un nombre minimal de concepts. L'approche *réflexive* est destinée aux concepteurs d'application qui souhaitent spécialiser le système pour les besoins propres à leur type d'application, et de manière duale aux concepteurs de système qui souhaitent ainsi concevoir leur système selon une telle architecture « ouverte » et adaptable. L'approche réflexive vise donc à la définition d'une architecture permettant la spécialisation dynamique du système avec un impact minimal sur le programme de l'application.

## 6.5 Approche applicative

### 6.5.1 Besoins de modularité et de structure

L'idée sous-jacente à l'approche *applicative* est d'appliquer les concepts d'encapsulation et d'abstraction, voire de classe et d'héritage, à la conception et la mise en œuvre des systèmes parallèles et répartis. Il s'agit de programmer un système parallèle ou réparti avec une méthodologie à objets et dans un langage à objets. Le résultat est un ensemble de bibliothèques de classes représentant le système en question. Ces bibliothèques sont organisées en « *framework(s)* », et font rapidement apparaître des invariants d'architecture, appelés « *patterns* » ou « *design patterns* » [Gamma *et al.*, 1994] (*chapitre 4*).

La première motivation de l'approche applicative est d'accroître la modularité de ces systèmes. En effet, en décomposant un système en un ensemble de modules dotés d'interfaces bien définies, on peut espérer pouvoir modifier la mise en œuvre de certains composants avec un minimum d'impact sur les autres. D'une certaine manière, une telle modularité permettrait d'assurer la portabilité de ces systèmes.

La deuxième motivation est de mieux structurer les systèmes parallèles et répartis, en évitant des conceptions « à la Unix », dans lesquelles les niveaux d'abstractions sont difficilement discernables. Il est bien évidemment plus facile de comprendre un système lorsque ce dernier est constitué d'un ensemble de composants dont les interfaces (les services) sont bien définis. Ceci a d'ailleurs donné lieu à la conception actuelle des systèmes d'exploitation récents, tels que CHORUS [Rozier, 1992], MACH [Accetta *et al.*, 1986], et CHOICES [Campbell *et al.*, 1993], fondés sur un noyau minimal (« *micro-kernel* ») et dont les différents services sont assurés par différents serveurs spécialisés. Le bénéfice escompté est, d'une part de rendre les systèmes parallèles et répartis plus facilement utilisables et extensibles, et d'autre part d'améliorer les performances en n'utilisant dans chaque exécution que les modules qui sont strictement nécessaires.

Nous présentons maintenant trois exemples de l'approche applicative : tout d'abord les cas de SMALLTALK et EIFFEL, qui introduisent divers aspects de programmation concurrente et répartie sous la forme de bibliothèques de classes, puis le cas du système d'exploitation CHOICES qui illustre l'application de la méthodologie objet à la structuration et la généricité de systèmes d'exploitation répartis.

### 6.5.2 L'exemple de Smalltalk

L'une des principales originalités de l'environnement de programmation SMALLTALK-80 [Goldberg et Robson, 1983] est d'offrir, d'une part un langage à objets minimal, et d'autre part des bibliothèques de classes représentant divers mécanismes et outils de programmation, qui font ainsi la richesse de l'environnement. Il s'agit donc bien d'un « tout objet » uniformément appliqué à l'ensemble des concepts de programmation. Par exemple, les structures de contrôle (tests, boucles, itérations) sont représentées par des méthodes utilisant la seule généricité de l'envoi de message, et non pas par des tests booléens primitifs comme dans la plupart des autres langages de programmation. Leurs arguments sont des « blocs », qui sont des portions de programme, délimitées par des crochets (`[` et `]`), et dont l'évaluation est différée. Ils sont représentés comme instances de la classe `BlockContext`. Le fait que les structures de contrôle sont en SMALLTALK-80 des méthodes standard comme les autres, permet au programmeur d'en définir si besoin de nouvelles à son goût.

**Programmation concurrente**

La notion de bloc est également la base du multi-tâche en SMALLTALK-80. En réponse au message `fork`, un bloc crée un objet *processus*[4] qui exécute le bloc de programme en question, de manière concurrente à la séquence d'appel. Un processus est représenté par une instance de la classe `Process`. Les méthodes associées à cette classe permettent de gérer les processus (suspendre, relancer, changer sa priorité, etc.). Le séquenceur est lui-même représenté par un objet, instance de la classe `ProcessorScheduler`. Enfin, la primitive de synchronisation de base est le (classique) sémaphore, représenté par la classe `Semaphore`. Des abstractions de plus haut niveau existent également en standard : la classe `SharedQueue` pour communication et synchronisation entre processus, et la classe `Promise` pour représenter l'évaluation anticipée d'une expression par un processus concurrent.

Grâce à cette approche par bibliothèques, les mécanismes nécessaires à la programmation concurrente sont encapsulés dans des concepts bien définis. De par leur structuration en une hiérarchie de classes, ces concepts sont plus compréhensibles que s'ils étaient fournis en vrac comme des primitives d'un langage. Ils sont également très génériques et bien entendu extensibles. Ainsi un programmeur avisé peut étendre les classes existantes et définir de nouvelles abstractions pour la programmation concurrente, tels divers mécanismes de synchronisation (moniteurs, gardes, compteurs de synchronisation, etc.), comme par exemple dans les plates-formes SIMTALK [Bézivin, 1987] et ACTALK [Briot, 1994 ; Briot, 1996]. De plus, dans le cadre du projet ACTALK, le séquenceur standard a été étendu en un séquenceur générique, pour paramétrer et classifier différentes stratégies de séquencement [Lescaudron, 1992].

**Programmation répartie**

Bien qu'étant un environnement de programmation fondamentalement mono-utilisateur et mono-processeur, SMALLTALK-80 offre également un certain nombre de bibliothèques de base permettant la construction de mécanismes de répartition [Briot et Guerraoui, 1996]. Il existe des bibliothèques quasi-standard pour les communications distantes (*sockets* UNIX, *remote procedure call*). De plus, une bibliothèque standard de stockage de structures d'objets : le « Binary Object Streaming

4. Smalltalk utilise le terme « processus ». Cependant si l'on utilise la terminologie générale actuelle, on devrait plutôt employer le terme « tâches » (ou encore « processus de poids léger »), qui partagent toutes le même espace de noms à la différence des processus de « poids lourd », tels en Unix.

Service (BOSS) », offre un mécanisme de base d'encodage/représentation des objets pour construire divers mécanismes de programmation répartie, tels que : persistance, encodage des communications réparties, et transactions.

Ainsi, dans le cadre du projet GARF [Garbinato *et al.*, 1995], une bibliothèque de classes a été mise en œuvre pour la programmation répartie, comportant divers modèles de gestion d'objets (persistant, dupliqué, etc.) et de communication (« *multicast* », atomique, etc.). Le produit HP DISTRIBUTED SMALLTALK offre également un ensemble de services répartis selon le standard OMG (CORBA, voir au § 6.6.5), eux-mêmes mis en œuvre en SMALLTALK-80.

### 6.5.3 L'exemple d'Eiffel

EIFFEL est un langage à objets qui a été conçu pour appliquer la méthodologie et les techniques à objets au génie logiciel [Meyer, 1988] (*chapitre 2*). Bien que destiné à la programmation séquentielle, de nombreux travaux ont par la suite été menés pour étendre EIFFEL vers la prise en compte de la concurrence, du parallélisme et de la répartition (certaines de ces approches ont été regroupées dans le dossier spécial [Meyer, 1993a]).

L'approche défendue par le concepteur d'EIFFEL lui-même [Meyer, 1993b] est *applicative*, mais surtout *minimaliste*. Il s'agit d'élargir la portée des concepts et mécanismes déjà existants (et donc d'augmenter leur généricité) sans en introduire de nouveaux, du moins le strict minimum. Ainsi, la sémantique des assertions d'EIFFEL, sous forme de pré- et post-conditions à satisfaire, se trouve redéfinie dans un contexte concurrent comme une attente tant que les conditions ne sont pas réunies [5]. Cette approche est très séduisante par son économie de moyens, et de ce fait sa simplicité, mais elle ne peut à elle seule couvrir tout le champ des besoins.

Des constructions et mécanismes supplémentaires, pouvant être nécessaires, sont décrits et mis en œuvre à l'aide de bibliothèques, en appliquant la méthodologie objet pour les organiser. Ainsi par exemple, la classe `Concurrency` [Karaorman et Bruno, 1993] encapsule une activité associée à l'objet et la transmission de message sans attente et distante.

Il nous faut enfin signaler l'approche relativement originale de l'environnement ÉPÉE de programmation répartie [Jézéquel, 1993b]. ÉPÉE introduit le parallélisme (et la répartition) au niveau des données pour des structures de données complexes (par exemple des matrices), suivant en cela le modèle « Single Program Multiple Data (SPMD) » [6]. La mise en œuvre se fait par bibliothèques de structures abstraites réparties sur plusieurs processeurs, et ce sans aucun ajout au langage EIFFEL.

5. Selon les principes de la synchronisation comportementale, détaillée au § 6.6.4.

6. Le parallélisme de ÉPÉE est ainsi très différent de celui des langages à *objets actifs* [Yonezawa et Tokoro, 1987]) qui introduit le parallélisme au niveau de l'activation (encore appelé *parallélisme de contrôle*), selon le modèle « Multiple Instructions Multiple Data (MIMD) ». Il est vrai que les objets représentent la dualité (et l'unification) entre données d'une part, et procédures (activation potentielle) d'autre part. ÉPÉE révèle donc d'une certaine manière la dimension « données » des objets pour le parallélisme.

### 6.5.4 L'exemple de Choices

CHOICES [Campbell *et al.*, 1993] est un système d'exploitation développé à l'Université de l'Illinois depuis 1987. La principale caractéristique de ce système d'exploitation est qu'il a été conçu pour être *générique*. L'objectif de cette généricité est, non seulement de pouvoir être porté facilement sur diverses machines, mais également de pouvoir faire varier différentes caractéristiques telles que : formats de fichiers, réseaux de communication, et modèles de mémoire (partagée ou répartie). Une méthodologie à objets est proposée pour la conception et la programmation, à la fois d'applications réparties, et d'extensions du noyau CHOICES.

CHOICES est mis en œuvre en C++, et pour être plus précis, CHOICES a été développé à partir d'une bibliothèque de classes C++, définie pour la programmation répartie. Aussi bien le matériel que l'interface d'application et les ressources du système sont modélisés par des classes. Parmi les classes définies, on peut citer la classe `ObjectProxy` qui réalise les communications distantes entre objets, ainsi que les classes `MemoryObject` et `FileStream` qui gèrent la mémoire, et la classe `ObjectStar` qui étend en quelque sorte le concept de pointeur. Cette dernière classe permet de rendre transparentes les invocations distantes sans nécessiter d'étape de pré-compilation. De plus, la classe `ObjectStar` permet la récupération automatique de mémoire (un objet est détruit lorsqu'il n'y a plus de référence sur lui).

Même si un système comme CHOICES reste un prototype universitaire, l'un de ses grands mérites est d'avoir montré qu'un système d'exploitation réparti pouvait être développé en suivant une méthodologie à objets dans un langage à objets. Cependant, le champ d'applications qui reposent sur le système CHOICES est encore plutôt restreint et ne permet pas de juger de sa facilité d'utilisation. Les nombreux développements autour du système (par exemple, le récupérateur automatique de mémoire adaptable) permettent néanmoins d'apprécier dès maintenant, dans une certaine mesure, son extensibilité.

### 6.5.5 À la recherche d'abstractions standard

De manière plus générale, l'idée derrière l'approche applicative est la définition et la mise en œuvre d'abstractions standard permettant, d'une part de simplifier la programmation parallèle et répartie, et d'autre part de tirer parti de la flexibilité des systèmes d'exploitation sous-jacents.

L'exemple le plus significatif dans la programmation concurrente est l'abstraction de synchronisation nommée *sémaphore* qui, grâce à une interface bien définie (opérations `wait` et `signal`), et un comportement connu (métaphore des sémaphores ferroviaires), constitue désormais un standard de la programmation concurrente. Un tel type d'abstraction sert ensuite de fondation pour construire divers mécanismes de synchronisation plus élaborés. Les possibilités de classification et de spécialisation de la programmation par objets sont particulièrement appropriées pour rendre compte d'une telle bibliothèque/hiérarchie d'abstractions, par exemple dans les plates-formes SIMTALK et ACTALK déjà citées au § 6.5.2.

Dans un contexte réparti, Andrew Black [Black, 1991] a proposé une approche similaire en suggérant, comme premier exercice, de décomposer le concept de transaction en un ensemble d'abstractions. L'idée de cet exercice est de représenter des concepts tels que le « verrouillage », l'« atomicité » et la « persistance », par un ensemble d'objets que doit fournir un système pour supporter des transactions. La modularité d'une telle approche permettrait de définir divers modèles de transactions, adaptés à des applications particulières. Par exemple, une application de travail coopératif ne nécessite pas les mêmes contraintes de contrôle de concurrence entre transactions qu'une application de gestion bancaire[7]. Les concepts présentés dans les projets VENARI [Wing, 1994] et PHOENIX [Guerraoui et Schiper, 1995] vont dans cette direction, en permettant de définir différents modèles transactionnels à partir d'abstractions minimales.

### 6.5.6 Bilan de l'approche applicative

En résumé, l'approche applicative tente de réduire la complexité des systèmes parallèles et répartis, en les décomposant en une bibliothèque de classes. Chaque service du système est alors représenté par un objet. Cet objectif de modularité est très important car les systèmes parallèles et répartis sont des systèmes complexes, faisant intervenir des mécanismes de très bas niveau (par exemple la communication à travers le réseau pour un système réparti). De plus, ce sont des systèmes souvent développés par des équipes de programmeurs pour lesquelles la modularité est fondamentale. Enfin, la difficulté de la maintenance et de l'extension des systèmes « à la UNIX » est principalement due à un manque de modularité.

Bien que des tentatives soient entreprises dans ce sens, il est encore trop tôt pour considérer qu'il existe une bibliothèque de classes susceptible de devenir un standard pour la programmation parallèle ou répartie. Les difficultés d'un tel exercice (trouver les abstractions adéquates) résident, d'une part dans une bonne compréhension des mécanismes minimaux nécessaires pour ces types de programmation, et d'autre part dans un consensus sur ces mécanismes. Un tel consensus devrait mettre en jeu différentes communautés de chercheurs : langages de programmation, systèmes d'exploitation, et de certains types d'applications réparties (en particulier les bases de données pour des mécanismes transactionnels). Le fait qu'une abstraction comme le sémaphore soit devenue un standard pour la synchronisation permet néanmoins de considérer que ces difficultés sont surmontables à terme.

## 6.6 Approche intégrée

### 6.6.1 Besoins d'unification

La multitude de concepts manipulés constitue l'une des difficultés majeures de la programmation parallèle et répartie. En plus des concepts classiques de manipulation de données d'un langage traditionnel, la programmation parallèle et répartie introduit des concepts tels que : *processus*, *sémaphore*, *moniteur*, *transaction*, etc. L'approche applicative permet de bien structurer les mécanismes de parallélisme et

7. La seconde exige des garanties plus fortes (sérialisation stricte des transactions), à travers un mécanisme de verrouillage, alors que la première peut se passer d'un tel mécanisme.

de répartition, mais a tendance à laisser le programmeur en charge de deux tâches distinctes : d'une part la programmation en terme d'objets, et d'autre part la gestion du parallélisme et de la répartition (également exprimée à l'aide d'objets, mais *pas les mêmes !*).

De plus, la programmation à l'aide de bibliothèques peut devenir un peu lourde, comme la programmation du parallélisme et de la répartition s'ajoutent au style standard de programmation. Ainsi par exemple, la classe `Concurrency`, décrite au § 6.5.3, oblige à une certaine dose de manipulation explicite de messages (voir en particulier [Karaorman et Bruno, 1993, p. 109–11]), par opposition au modèle standard et implicite de transmission de message.

De manière à simplifier la programmation, il est alors possible d'*intégrer* de tels concepts et mécanismes directement dans un langage ainsi étendu (par exemple EIFFEL//, cf. *chapitre 7*), ou bien encore de créer un nouveau langage (par exemple ABCL/1 [Yonezawa, 1990]).

En résumé, au lieu de laisser la programmation d'une application et la programmation du parallélisme et de la répartition relativement orthogonales, l'approche *intégrée* vise à les fusionner en intégrant/unifiant les concepts autant que possible, pour offrir au programmeur un cadre conceptuel (objet) unique et simplifié.

### 6.6.2 Niveaux d'intégration

Il existe plusieurs niveaux d'identification et d'intégration entre les concepts objets et les concepts du parallélisme et de la répartition. Nous en considérons trois principaux qui sont indépendants. De ce fait, comme on va le voir par la suite, tel ou tel langage ou système pourra suivre un niveau d'intégration, mais pas un autre.

Une première intégration des concepts d'objet et d'activité (processus ou tâche) conduit au concept d'*objet actif*. En effet, un objet et un processus peuvent tous deux être considérés comme des entités contenant des données et des traitements (voir au § 6.3.2).

Une deuxième intégration conduit à associer la synchronisation à l'activation des objets, on parlera alors d'*objet synchronisé*. La transmission de message est alors considérée comme une synchronisation implicite entre émetteur et récepteur. En général, on associe en plus des mécanismes de contrôle de la concurrence des invocations au niveau d'un objet, par exemple en associant une *garde* au niveau de chaque méthode.

Remarquons qu'un objet actif implique déjà une forme simple d'objet synchronisé, puisque l'existence d'une activité privée à l'objet séquentialise de fait les invocations. Par contre, certains langages ou systèmes, comme GUIDE [Balter *et al.*, 1994] ou ARJUNA [Parrington et Shrivastava, 1988], associent la synchronisation aux objets tout en laissant objets et activités séparés.

Un troisième niveau d'intégration conduit à considérer l'objet comme unité de répartition, on parlera d'*objet réparti*. Les objets sont alors vus comme des entités pouvant être réparties et dupliquées sur différents sites. Ceci conduit également à étendre la métaphore de transmission de message à la communication à travers un réseau quelconque.

### 6.6.3 Objets actifs

L'idée sous-jacente au concept d'objet actif consiste à considérer l'objet comme doté d'une ressource de calcul, c'est-à-dire doué d'activité propre. Cette approche, naturelle, a eu une grande influence. L'ouvrage « Object-Oriented Concurrent Programming » [Yonezawa et Tokoro, 1987] regroupe un certain nombre de langages de programmation de cette famille, les langages d'acteurs [Lieberman, 1987 ; Agha, 1986] faisant en la matière figure de pionniers. Le concept d'objet actif a eu beaucoup de succès dans le domaine de l'intelligence artificielle, car il constitue une fondation naturelle pour construire des « agents » de plus haut niveau, pour des problèmes de gestion répartie de la connaissance (appelés « systèmes multi-agents ») [Gasser et Briot, 1992].

Au niveau du traitement des messages par un objet actif donné, le modèle par défaut est celui d'un objet traitant les requêtes de manière séquentielle (« acteur sérialisé »). La concurrence provient donc uniquement des activités indépendantes des divers objets (à ce titre on parle de concurrence *inter-objets*). Si on permet à un objet de traiter non pas une, mais plusieurs requêtes simultanément, on parle alors de concurrence *intra-objet*, et il faut en général un contrôle supplémentaire pour assurer la consistance de l'état de l'objet, comme nous allons le voir dans le paragraphe suivant.

### 6.6.4 Objets synchronisés

La présence d'activités simultanées impose la présence d'un certain degré de synchronisation, autrement dit de restriction de la concurrence entre activités, de manière à assurer un déroulement correct du programme. La synchronisation s'associe naturellement aux objets et à leur communication, comme nous allons le voir, à travers plusieurs (sous)-niveaux d'identification.

**Synchronisation au niveau de la transmission de message**

La transposition immédiate du principe de transmission de message d'un univers séquentiel à un univers concurrent entraîne une synchronisation implicite de l'émetteur du message (appelant) sur le récepteur (appelé). On parle de transmission *synchrone* : l'objet appelant attend la fin de l'exécution de la méthode invoquée et le retour de la réponse pour poursuivre son exécution.

Dans le cas des objets actifs, l'appelant et l'appelé possèdent des activités indépendantes. Il s'avère alors intéressant d'introduire un type de transmission *asynchrone*, où l'appelant poursuit son activité aussitôt le message envoyé, c'est-à-dire qu'il n'attend pas la fin de l'exécution de la méthode invoquée. Cette forme de transmission introduit ainsi une concurrence à travers la communication. Elle est bien adaptée à une architecture répartie car l'objet appelé/serveur peut être situé sur une machine distante et le temps de communication, ajouté au temps d'exécution de la méthode invoquée, peut être considérable. Enfin, la transmission avec réponse(s) *anticipée(s)* (par exemple en ABCL/1 [Yonezawa, 1990]) est une forme « mixte » de transmission asynchrone. Elle permet à l'appelant d'obtenir une promesse de réponse (appelée un « objet futur »), immédiatement sans attendre la fin du traitement effectif.

### Synchronisation des invocations au niveau de l'objet

L'association (naturelle) de la synchronisation à la transmission de messages, c'est-à-dire à l'invocation des requêtes, offre ainsi l'avantage de traiter de manière transparente une partie de la synchronisation. Ceci a en effet l'intérêt de rendre transparente aux clients de l'objet la synchronisation de leurs requêtes, celles-ci étant traitées au niveau de l'objet acceptant les requêtes.

Dans le cas général d'objet sans concurrence interne, les invocations sont, par défaut, traitées l'une après l'autre, suivant leur ordre d'arrivée. Cependant un contrôle supplémentaire plus fin ou/et au contraire plus global peut être nécessaire, comme nous allons le voir.

### Niveaux de synchronisation

Nous distinguerons trois niveaux de synchronisation supplémentaires éventuels. Ce sont trois niveaux successifs de contrôle correspondant respectivement à l'intérieur d'un objet, à son interface (les services qu'il offre), et enfin à la coordination entre plusieurs objets :

- **Synchronisation intra-objet** : Dans le cas de concurrence *intra-objet* (traitement simultané de plusieurs requêtes), il est nécessaire d'assurer un certain contrôle de cette concurrence pour préserver la consistance de l'état de l'objet. On exprime alors, en général, les restrictions en terme d'exclusions entre opérations. Ainsi, par exemple, plusieurs lecteurs peuvent accéder en lecture simultanément à un même livre. Par contre, l'accès en écriture par un écrivain exclut tous les autres (écrivains comme lecteurs) [8].
- **Synchronisation comportementale** : Il se peut qu'un objet ne puisse temporairement traiter certains types de requêtes qui font pourtant partie de son interface. Ainsi, par exemple, un tampon de taille bornée (« bounded buffer ») ne pourra accepter une requête d'insertion (`put:`) si/tant qu'il est plein. Plutôt que de signaler une erreur, il est en général plus judicieux de laisser en attente une telle requête en attendant que la condition d'accès soit satisfaite. Ceci permet une transparence des invocations de services entre objets.
- **Synchronisation inter-objets** : On peut enfin désirer assurer une cohérence, non plus seulement individuelle, mais globale entre de multiples objets. Prenons l'exemple d'un transfert de fonds entre deux comptes bancaires. En l'occurrence, on veut assurer l'indivisibilité, au sens transactionnel [Bernstein *et al.*, 1987], du transfert. Cela consiste à rendre invisible un état transitoire et inconsistant où le premier compte a été débité mais le second n'a pas encore été crédité du montant équivalent. La synchronisation comportementale n'est alors plus suffisante. Il faut introduire une synchronisation qui met en œuvre les deux objets représentant les comptes bancaires.

8. Dans le cas d'une exclusion mutuelle systématique, on retombe sur le cas d'un objet séquentiel au niveau de son traitement des requêtes (§ 6.6.3).

**Formalismes de synchronisation**

De nombreux formalismes de synchronisation ont été proposés pour assurer ces différents niveaux de contrôle, aucun formalisme ne couvrant encore l'ensemble des niveaux. La plupart sont issus de travaux antérieurs ou, du moins, non spécifiques à la méthodologie par objets. En particulier, divers protocoles de synchronisation sont issus du domaine des systèmes d'exploitation et le modèle des transactions est issu des besoins des bases de données. Ces protocoles représentent en conséquence un effort, plus ou moins poussé, d'intégration de ces formalismes de synchronisation dans le modèle des objets.

Cette intégration est en général aisée. Les formalismes *centralisés*, tels que les chemins de synchronisation (« path expressions »), qui spécifient de manière abstraite les possibles entrelacements ou séquences d'invocations, s'associent de manière naturelle au niveau de la classe (par exemple dans le langage PROCOL [van den Bos et Laffra, 1991]). Un autre exemple est le concept de « *body* » : une procédure centrale qui décrit explicitement les types de requêtes que l'objet va accepter au cours de son activité [9]. On le trouve en particulier dans les langages POOL [America, 1986] et EIFFEL// [Caromel, 1993] (où il est nommé « live routine »).

Les formalismes *décentralisés*, en particulier les *gardes* ou conditions booléennes d'activation, s'associent eux de manière naturelle au niveau de chaque méthode (par exemple dans le langage/système GUIDE [Balter *et al.*, 1994]). Les *compteurs de synchronisation* (compteurs mémorisant le statut des invocations pour chaque méthode, c'est-à-dire le nombre d'invocations reçues, débutées et terminées), associés aux gardes, permettent d'exprimer un contrôle fin de synchronisation intra-objet, par exemple dans l'exemple des lecteurs et écrivains.

Enfin, le formalisme des *états abstraits* est très approprié à la synchronisation comportementale (voir au § 6.6.4). Le principe est le suivant : un objet se conforme à un certain état abstrait auquel correspond un ensemble de méthodes autorisées. Dans le cas du tampon borné (évoqué au § 6.6.4), trois états abstraits sont suffisants : `vide`, `plein`, et `intermédiaire`. L'état abstrait `intermédiaire` peut d'ailleurs s'exprimer comme union de `vide` et `plein`, et est ainsi le seul à autoriser les deux méthodes `get` et `put:`. Après traitement d'une invocation, l'état abstrait est recalculé, et peut ainsi changer suivant l'état ou/et la disponibilité des services de l'objet.

Notons, dès maintenant, que bien que ces intégrations soient en général relativement aisées, le problème de la réutilisation des spécifications de synchronisation se pose néanmoins (il sera abordé au § 6.6.6).

### 6.6.5 Objets répartis

Un objet représente une unité indépendante d'exécution, contenant données, traitements, voire des ressources propres de mise en œuvre (activité). Il est donc naturel d'identifier l'objet comme unité de répartition, et de duplication éventuelle. Cela permet de considérer une application répartie comme un ensemble d'objets, éventuellement situés sur des processeurs distincts. La transmission de message re-

9. Ce concept est issu du concept de « body » de SIMULA-67 décrit au § 6.3.2.

couvre ainsi la notion d'invocation locale ou distante suivant les cas, et également l'inaccessibilité éventuelle d'un objet/service.

### Invocation distante

Le mécanisme fondamental d'un système à objets répartis est l'invocation distante. Il est en général admis que, pour faciliter la programmation d'applications réparties, le mécanisme d'invocation distante doit, par défaut, être transparent. Autrement dit, un objet client qui invoque un autre objet serveur ne doit pas distinguer le cas où ce dernier est situé sur un autre processeur du cas où les deux objets sont situés sur le même processeur.

### Accessibilité et tolérance aux fautes

Pour prendre en compte l'inaccessibilité des objets, le système ARGUS [Liskov et Scheifler, 1983] permet d'associer des exceptions à chaque invocation. Si un objet est situé sur un processeur qui est inaccessible, à cause d'une faute du réseau de communication ou du processeur lui-même, l'exception est déclenchée. Cela permet par exemple d'invoquer un autre objet à la place. De manière complémentaire, pour assurer qu'une invocation qui n'a pu être exécutée ne conduit pas à des incohérences, on peut associer la notion de transaction à l'invocation synchrone. Autrement dit, si l'invocation échoue (par exemple si l'objet *serveur* invoqué devient inaccessible), les effets de l'invocation sont annulés. Étendant cette approche, le système KAROS [Guerraoui *et al.*, 1992] permet d'associer les transactions également aux invocations asynchrones.

### Migration

Afin d'assurer une plus grande accessibilité des objets, certains systèmes offrent des mécanismes de migration. Dans le langage EMERALD [Jul, 1994], et la couche d'exécution générique COOL [Lea *et al.*, 1993], le programmeur d'une application répartie peut décider qu'à un certain moment un objet doit migrer d'un processeur à un autre processeur. Le programmeur peut aussi contrôler (sous forme d'« attachements » en EMERALD) quels autres objets liés doivent également migrer avec lui. L'objectif est en définitive de placer sur un même processeur les objets qui communiquent très souvent entre eux. Cela permet d'alléger les goulots d'étranglement et de minimiser les communications distantes. Certains systèmes réalisent la migration de manière automatique (sans l'intervention du programmeur), pour assurer un rééquilibrage de la charge du système.

### Duplication

La duplication est un autre mécanisme de gestion de la répartition des objets. Une première motivation, comme pour la migration, est d'offrir une plus grande accessibilité. Ainsi un objet, sujet de beaucoup d'invocations à partir de différents clients distants, gagnera à être dupliqué sur les différents processeurs clients (ce premier mécanisme est analogue à un « cache » local). Une deuxième motivation de la duplication est la tolérance aux fautes. Lorsqu'un objet existe en plusieurs copies sur des processeurs différents, il peut tolérer les fautes de certains de ces processeurs. Dans les deux cas, se pose le problème de la cohérence des différentes copies d'un même objet (c'est-à-dire la garantie qu'elles possèdent toutes les mêmes valeurs). Afin d'assurer une telle cohérence, dans le système ELECTRA [Maffeis, 1995] par exemple, la notion d'invocation distante a été étendue. Dans ce cas, une invocation de l'objet entraîne l'invocation de toutes les copies, et des invocations concurrentes sont exécutées dans le même ordre total par toutes les copies. Andrew Black a également présenté, dans [Black et Immel, 1993], un mécanisme général d'invocation de groupe qui s'applique très bien au cas où un objet est dupliqué.

#### Tendances et standardisation

Devant la multitude de systèmes à objets répartis, deux groupes de standardisation sont apparus: le groupe autour de ODP (*Open Distributed Processing*) et le groupe OMG (*Object Management Group*) (tous deux décrits dans [Nicol *et al.*, 1993]). Alors que le premier, principalement un groupe de réflexion, est constitué de membres d'instituts de recherches qui définissent un modèle abstrait d'objet réparti, le second, composé d'industriels, a défini un modèle de mise en œuvre de système à objets répartis intitulé CORBA (*Common Architecture for an Object Request Broker*). Le standard CORBA a pour objectif de faire communiquer des objets situés sur des plates-formes différentes, grâce à l'utilisation d'un même langage de définition d'interfaces nommé IDL (*Interface Definition Language*).

### 6.6.6 Limites de l'approche intégrée

L'approche intégrée procède par unification des mécanismes objet avec les mécanismes du parallélisme et de la répartition. Cependant, certains conflits et limitations peuvent survenir entre ces différents mécanismes. Particulièrement significatif est le cas du mécanisme d'héritage que nous allons décrire ici. Un premier conflit se manifeste entre l'utilisation traditionnelle de l'héritage (pour réutiliser variables et méthodes) et son application à la spécialisation de la synchronisation. Un deuxième conflit est relatif à la notion de duplication qui, si elle est appliquée telle quelle aux objets, peut provoquer des duplications non désirées de certaines invocations. Un troisième conflit concerne l'intégration de protocoles transactionnels incompatibles. Enfin, un quatrième conflit provient cette fois des hypothèses fortes (mémoire centralisée) des techniques traditionnelles de mise en œuvre des mécanismes de factorisation (classes et héritage), ce qui limite de fait leur transposition immédiate à un univers réparti.

#### Spécialisation de la synchronisation

Le mécanisme d'héritage est un des grands atouts de la programmation par objets pour la réutilisation et la spécialisation des programmes. Il est en conséquence naturel de vouloir utiliser l'héritage pour spécialiser les spécifications de synchronisation exprimées dans une classe d'objets. Cependant, l'expérience montre tout d'abord que la synchronisation est une dimension plus complexe à exprimer que la structure (variables d'instance), ou que les fonctionnalités (méthodes), d'une classe. Elle est ainsi beaucoup plus difficile à spécialiser/hériter, du fait de l'interdépendance des conditions de synchronisation. De plus les différentes utilisations de l'héritage (pour hériter des variables, des méthodes, et des synchronisations), peuvent entrer en conflit, comme le remarque [McHale, 1994]. Il se peut ainsi que dans certains cas, la définition d'une sous-classe, rajoutant une seule méthode, impose la redéfinition de l'ensemble des spécifications de synchronisation (voir le paragraphe immédiatement suivant). Cette limitation de la réutilisabilité des spécifications de synchronisation par l'héritage a été nommée par Satoshi Matsuoka « the *inheritance anomaly phenomenon* » (voir [Matsuoka et Yonezawa, 1993] et [McHale, 1994] pour deux études complètes comprenant des exemples détaillés).

Les spécifications selon les formalismes centralisés explicites (voir au § 6.6.4) s'avèrent très difficiles à réutiliser. En pratique il faut trop souvent les redéfinir entièrement. Les formalismes décentralisés, intrinsèquement plus modulaires, sont eux beaucoup plus adaptés à une spécialisation sélective. Cependant, cette décomposition extrême au niveau de chaque méthode est aussi une arme à double tranchant. L'essence du problème réside dans le fait que les spécifications de synchronisation, même décomposées au niveau de chaque méthode, restent plus ou moins interdépendantes. Ainsi, par exemple, dans le cas d'une synchronisation intra-objet avec des gardes, le rajout dans une sous-classe d'une nouvelle méthode d'écriture va imposer la spécialisation de toutes les autres gardes pour que chacune prenne en compte cette nouvelle exclusion mutuelle.

Les derniers développements des travaux dans le domaine montrent : l'intérêt de spécifier et spécialiser indépendamment les synchronisations comportementale et intra-objet [Thomas, 1992], la possibilité d'offrir le choix entre plusieurs formalismes de synchronisation [Matsuoka et Yonezawa, 1993], voire d'offrir au concepteur la possibilité de spécialiser et combiner divers formalismes [Briot, 1996], et enfin les promesses de la généricité (en instanciant des spécifications suffisamment abstraites et génériques), plutôt que l'héritage, comme outil de réutilisation [McHale, 1994].

**Duplication d'invocations**

Les protocoles de communication définis pour gérer la duplication des services dans un système réparti tolérant aux fautes (voir au § 6.6.5) considèrent un modèle client/serveur simple. L'application des mêmes protocoles aux objets pose le problème de la duplication des invocations. En effet, un objet agit généralement, à la fois comme un client, et comme un serveur. Autrement dit, un objet dupliqué en tant que serveur, peut cependant à son tour invoquer d'autres objets (cette fois en tant que client). Toutes les copies de l'objet se retrouveront en conséquence à invoquer ces autres objets plusieurs fois. Le résultat est la duplication inutile des invocations. Cette duplication peut conduire, au meilleur des cas à l'inefficacité du système, et au pire des cas à des incohérences. Une même opération (par exemple d'incrémentation) peut ainsi être exécutée plusieurs fois plutôt qu'une. Ce problème est discuté dans [Mazouni *et al.*, 1995] et des solutions alternatives (appelées « pre-filtering » et « post-filtering ») sont proposées. Le « pre-filtering » consiste à coordonner les traitements effectués par les copies d'un objet client, afin de ne générer qu'une invocation. Le « post-filtering » est l'opération duale, qui assure la coordination au niveau des copies du serveur, pour ne pas traiter des invocations redondantes.

**Compatibilité entre protocoles transactionnels**

Il est tentant d'intégrer le protocole de contrôle de la concurrence transactionnelle au niveau des objets. Ainsi on peut définir localement, pour un objet donné, le contrôle de concurrence adéquat qui permet un contrôle optimal. Par exemple, la prise en compte de la commutativité des opérations permet d'entrelacer (sans bloquer) des transactions pour un objet donné. Malheureusement, le gain apporté en matière de modularité et de spécialisation peut amener à des problèmes d'incompatibilité [Weihl, 1989]. Si les objets utilisent différents protocoles de sérialisation des transactions (c'est-à-dire, ordonnancent les transactions suivant des ordres différents), les exécutions globales des transactions peuvent être incohérentes (c'est-à-dire non sérialisables). Une approche proposée pour résoudre le problème est basée sur des conditions locales à garantir par les objets, afin d'assurer la compatibilité des différents protocoles [Guerraoui, 1995b].

**Mise en œuvre des factorisations (héritage et variables globales)**

Ce dernier exemple de limitation est plus général (c'est-à-dire moins spécifique à l'approche intégrée). Il provient du fait que les stratégies de mise en œuvre des mécanismes de factorisation (classes et héritage) ont souvent été fondées sur des hypothèses fortes d'informatique traditionnelle, c'est-à-dire séquentialité de l'exécution des programmes et mémoire centralisée. Elles peuvent ainsi atteindre leurs limites une fois transposées directement dans l'univers de la programmation parallèle et répartie.

L'utilisation des variables de classes, c'est-à-dire de variables partagées par tous les objets d'une même classe, pose un problème dans un système réparti. Il semble difficile, à moins d'introduire des mécanismes transactionnels complexes, de garantir que la mise à jour d'une variable de classe soit immédiatement reflétée sur toutes les instances d'une classe, lorsque celles-ci sont situées sur plusieurs processeurs. Ce problème est en fait lié aux variables partagées en général.

De manière plus générale encore, la mise en œuvre de l'héritage dans un système réparti, pose le problème de l'accès distant au code des super-classes, à moins que celles ci ne soient dupliquées sur tous les processeurs avec le coût conséquent. Une méthode de répartition des bibliothèques de classes en modules autonomes est proposée dans [Gransart, 1995]. Elle permet de gérer de manière abstraite la répartition, non pas seulement des instances, mais aussi des classes (c'est-à-dire du code) et donc de minimiser sa duplication.

Une approche extrême, pour remplacer le mécanisme d'héritage entre classes, est le concept de *délégation* (historiquement introduit dans le langage d'acteurs ACT 1 [Lieberman, 1987]). Intuitivement, un objet qui ne peut lui-même interpréter un message, le déléguera à un mandataire [10]. Ce dernier le traitera donc à sa place

10. On peut noter que, de manière à traiter correctement la récursion, ceci impose d'inclure le receveur initial dans le message ainsi délégué.

ou bien le délèguera lui-même à nouveau. Cette alternative à l'héritage est très séduisante [11], car elle ne repose que sur la transmission de messages, et donc est apte à être répartie. Cependant, elle nécessite un mécanisme non trivial de synchronisation pour assurer un ordonnancement correct des messages récursifs (avant les autres messages). De ce fait la délégation ne peut offrir une solution générale et complète comme alternative à l'héritage [Briot et Yonezawa, 1987].

### 6.6.7 Bilan de l'approche intégrée

En résumé, l'approche intégrée est extrêmement séduisante par la fusion qu'elle propose des concepts et mécanismes, d'une part de la programmation par objets, et d'autre part de la programmation parallèle et répartie. Elle offre ainsi au programmeur un nombre minimal de concepts et un cadre unique de vision de ces différents aspects. Cependant, comme nous l'avons vu (§ 6.6.6), elle souffre de limitations dans certains secteurs d'intégration.

Un autre danger potentiel est qu'une unification/intégration trop systématique peut aboutir à un modèle trop réducteur (*trop d'uniformité nuit à la variété !*) et posant des problèmes d'efficacité. Ainsi par exemple, dans les langages d'acteurs, tout objet est un objet actif. Or, les technologies actuelles ne permettent pas toujours de gérer efficacement la prolifération des processus qui peuvent résulter d'un tel modèle [Capobianchi *et al.*, 1992]. On peut également citer les systèmes ARGUS [Liskov et Scheifler, 1983] et KAROS [Guerraoui *et al.*, 1992] qui, en associant systématiquement des transactions aux invocations (transmissions de message), peuvent pénaliser les performances d'une application répartie dans laquelle cette association ne se justifie pas (par exemple une application de travail coopératif).

Une dernière limitation, et non la moindre, tient aux capacités parfois insuffisantes de réutiliser les programmes séquentiels déjà existants, hormis en les encapsulant dans des objets actifs. Une approche pragmatique consiste aussi à considérer une cohabitation raisonnable (différentes classes) entre les objets actifs et les autres, appelés alors *objets passifs*. Cette approche, alors moins homogène, impose des règles méthodologiques pour la distinction entre objets actifs et objets passifs [Caromel, 1993].

## 6.7 Approche réflexive

### 6.7.1 Vers une approche par combinaison

Comme nous l'avons vu précédemment, l'approche applicative (par bibliothèques) aide à la structuration des abstractions et mécanismes nécessaires à la programmation parallèle et répartie, grâce aux concepts d'encapsulation, de généricité, de classe, et d'héritage. L'approche intégrée, quant à elle, apporte une minimisation du nombre de concepts à disposition du programmeur et une meilleure trans-

11. Les avantages comparés de l'héritage et de la délégation, dans un contexte de programmation séquentielle, ont d'ailleurs fait l'objet d'un grand débat au cours de la deuxième moitié des années 80 [Stein *et al.*, 1989] (*chapitre 8*).

parence des mécanismes. Cependant, elle a tendance à restreindre la flexibilité et l'efficacité des mécanismes offerts. En effet, les langages et systèmes bâtis à partir de bibliothèques, se retrouvent en général plus extensibles que nombre de langages conçus selon l'approche intégrée. Ceci, parce que les bibliothèques peuvent construire et simuler, et donc assurent une plus grande flexibilité, alors que les nouveaux langages peuvent fixer trop tôt leurs modèles de calcul et de communication. L'idéal serait donc d'arriver à conjuguer les avantages des deux approches, autrement dit : la simplification de l'approche intégrée et la flexibilité de l'approche applicative.

Il faut signaler que l'approche applicative et l'approche intégrée sont en fait destinées à *différents* niveaux d'utilisation. L'approche intégrée est plus particulièrement destinée au programmeur d'applications et lui offre un cadre conceptuel unique et simplifié. L'approche applicative est plus particulièrement destinée au concepteur de systèmes, ou encore à l'utilisateur averti qui souhaite les spécialiser, et leur offre une méthodologie de structuration, sous forme de bibliothèques de composants et de protocoles.

En conséquence de quoi, et contrairement à ce qui aurait pu sembler au premier abord, l'approche *applicative* et l'approche *intégrée* ne sont *pas* en concurrence, mais sont bien au contraire *complémentaires*. La question qui se pose alors est la suivante : « Comment peut-on combiner au mieux ces deux approches ? », et pour être plus précis : « Comment les interfacer ? ». Il se trouve qu'une méthodologie générale d'adaptation du comportement de systèmes informatiques, appelée *réflexion*, offre un tel type d'articulation.

### 6.7.2 Réflexion

La *réflexion* est une méthodologie générale pour décrire, contrôler, et adapter le comportement d'un système informatique quelconque. L'idée de base est de doter le système d'une représentation d'un certain nombre de caractéristiques de son propre comportement, d'où le nom donné de « *réflexion* » (« *reflection* » en anglais). Ainsi diverses caractéristiques de représentation (statique) et d'exécution (dynamique) des programmes sont rendues concrètes, également sous la forme d'un (ou plusieurs) programme(s), appelés *méta-programmes*. Ils représentent donc le comportement par défaut (interprète, compilateur, moniteur d'exécution) du système ou langage. La spécialisation de méta-programmes permettra ainsi de particulariser l'exécution d'un programme utilisateur, en changeant les choix de représentation mémoire, le contexte de calcul, la nature des mécanismes et des protocoles, etc. Notez que le *même* langage est employé, pour l'écriture des programmes, comme pour leur contrôle. Par contre, la *séparation* entre programme utilisateur (appelé *niveau objet*) et description/contrôle (*niveau méta*) est, elle, clairement et complètement établie.

La réflexion permet de décorréler les bibliothèques de méta-programmes, spécifiant les caractéristiques d'exécution (gestion de la concurrence, de la répartition, des ressources), du programme de l'application. Ceci augmente en conséquence modularité, réutilisabilité et lisibilité des programmes. Enfin, la réflexion offre une méthodologie pour « ouvrir » et rendre adaptable, via une *méta-interface* [12], les choix de mise en œuvre et de gestion des ressources. Ces derniers sont en effet trop souvent, sinon pré-cablés et fixés, du moins non modifiables au niveau du langage de programmation lui-même, car délégués au système d'exploitation sous-jacent.

En résumé, la réflexion permet d'intégrer intimement des bibliothèques de protocoles avec un langage ou un système, offrant ainsi un cadre d'interface entre les approches et niveaux applicatifs et intégrés.

### 6.7.3 Réflexion et objets

La réflexion s'exprime particulièrement bien dans les modèles à objets qui assurent une bonne encapsulation des niveaux ainsi que la modularité des effets. Il est alors naturel d'organiser (décomposer) le contrôle du comportement d'un système informatique à objets, en un ensemble d'objets. Une telle organisation est appelée un « *Meta-Object Protocol (MOP)* » [Kiczales *et al.*, 1991]. Ses composants sont appelés *méta-objets* [Maes, 1987], puisque les différents méta-programmes sont représentés par des objets. Ils peuvent représenter certaines caractéristiques du contexte d'exécution d'un objet, telles que : représentation, mise en œuvre, exécution, communication, et emplacement. La spécialisation de méta-objets permet ainsi d'étendre, et de modifier, localement, le contexte d'exécution d'un ou de plusieurs objets du programme.

La réflexion aide également à exprimer et adapter la gestion des ressources, non seulement au niveau d'un objet individuel, mais également à un niveau plus large, tel que : séquenceur, processeur, espace de noms, groupe d'objets, etc. Ces ressources physiques ou logicielles sont alors représentées par des méta-objets au niveau du langage ou système. Ceci permet ainsi l'expression de politiques fines (en particulier pour l'équilibre de charges et le séquencement) avec toute la puissance algorithmique d'un langage de programmation, par opposition à des algorithmes globaux et câblés, habituellement optimisés pour un type d'application.

### 6.7.4 Degrés de réflexivité

L'architecture CODA [McAffer, 1995] est un exemple représentatif d'architecture réflexive générale (autrement dit un « MOP ») à *méta-composants* [13]. CODA considère par défaut sept méta-composants associés à chaque objet, et correspondant à : l'*envoi de message*, leur *réception*, le *stockage des messages reçus*, leur *sélection*, la *recherche de méthode*, l'*exécution*, et enfin l'*accès à l'état* de l'objet. Un objet doté des méta-composants par défaut se comporte comme un objet standard

12. Cette *méta-interface* permet au programmeur client d'adapter et d'optimiser le *comportement* d'un module logiciel, indépendamment de ses *fonctionnalités*, qui restent accessibles via l'interface *standard* (« *base-interface* »). Ceci a été nommé par Gregor Kiczales le concept d'« *open implementation* » [Kiczales, 1994], qui peut être traduit par « mise en œuvre ouverte ».

13. Par la suite, nous emploierons le terme ou encore *composant*, plutôt que le terme *méta-objet*, de manière à souligner les principes d'interchangeabilité entre composants d'une architecture réflexive telle que CODA.

(c'est-à-dire séquentiel et passif) [14]. L'utilisation de méta-composants particuliers, permet de particulariser sélectivement tel ou tel aspect du modèle de représentation ou d'exécution de l'objet. L'interface entre les méta-composants est clairement exprimée de façon à composer, de manière relativement libre, des méta-composants de diverses origines.

Notez que d'autres architectures réflexives peuvent être plus spécialisées, et proposent, quant à elles, un nombre plus réduit de composants (également plus abstraits). Ainsi, la plate-forme ACTALK [Briot, 1994] est spécialisée pour la programmation concurrente par objets selon l'approche intégrée. Elle classifie différents modèles (méta-composants) : (1) d'*activité* (acceptation implicite ou explicite des requêtes, concurrence intra-objet, etc.) et de *synchronisation* (états abstraits, gardes, compteurs de synchronisation, etc.) [Briot, 1996], (2) de *communication* (synchrone, asynchrone, avec réponse anticipée, etc.), et (3) d'*invocation* (avec estampillage temporel, avec priorités, etc.). ACTALK permet d'associer, relativement librement, ces différents modèles à des classes d'objets/programmes et ainsi de faire varier leurs paramètres de concurrence et de communication.

La plate-forme GARF [Garbinato *et al.*, 1994 ; Garbinato *et al.*, 1995] est, quant à elle, spécialisée pour la programmation répartie et tolérante aux fautes. Elle classifie différents modèles : (1) de gestion d'objet (persistant, dupliqué, etc.) et (2) de communication (un seul destinataire, « multicast », atomique, etc.). Ces deux dimensions semblent s'avérer en pratique suffisantes pour traiter une large gamme de problèmes d'applications réparties et tolérantes aux fautes.

De manière plus générale, selon les objectifs visés, ainsi que l'équilibre recherché entre flexibilité, généralité, simplicité et efficacité, un langage, ou système, sera plus ou moins *réflexif*. Tout dépend en effet de la quantité et la portée des mécanismes qui seront ainsi « remontés » au méta-niveau. Ainsi, certains mécanismes peuvent s'exprimer sous forme de *méthodes réflexives*, sans faire intervenir un méta-objet explicite et complet.

Le langage SMALLTALK-80 est un exemple très représentatif de cette dernière catégorie. En plus de la méta-représentation des *structures* du programme et de son exécution (les classes, méthodes, messages, contextes, processus, etc., sont des objets à part entière, voir au § 6.5.2), quelques mécanismes réflexifs très puissants permettent également, un certain contrôle de l'*exécution* (redéfinition de la transmission de messages en cas d'erreur, référence au contexte courant, échange de références, etc.). Ces facilités permettent la construction aisée, et l'excellente intégration [Briot et Guerraoui, 1996], de nombreuses plates-formes de programmation concurrente, parallèle et répartie, telles que SIMTALK, ACTALK, GARF et CODA.

### 6.7.5 Exemples d'applications

Nous allons maintenant rapidement illustrer, pour une architecture réflexive donnée : CODA [15], l'introduction transparente de divers modèles de parallélisme et

14. Pour être plus précis, comme un objet standard SMALLTALK, car CODA est actuellement mis en œuvre en SMALLTALK.

15. Voir [McAffer, 1995] pour une description plus détaillée de son architecture et de ses bibliothèques de composants.

de répartition. Notez que, dans le cas de CODA, ainsi que pour la plupart des autres architectures réflexives décrites dans ce chapitre, le modèle de programmation de base est *intégré*, tandis que la réflexion permet une spécialisation via différentes *bibliothèques* de méta-composants.

### Modèle d'exécution concurrente

De manière à introduire la concurrence au niveau d'un objet donné (en le rendant *actif*, selon les principes de l'approche intégrée), deux méta-composants sont spécialisés. Un composant spécialisé de *stockage des messages reçus*[16] est une file d'attente (« queue » de type FIFO) qui stockera les messages selon leur ordre d'arrivée. Un composant spécialisé d'*exécution* associe une activité indépendante (mise en œuvre par un processus SMALLTALK) à l'objet. Ce processus exécute une boucle infinie de sélection et de traitement du premier message retiré du composant de *stockage des messages reçus*.

### Modèle d'exécution répartie

De manière à introduire la répartition, un nouveau méta-composant est *rajouté*, pour *encoder* (« *marshal* » ou encore « serialize ») les messages destinés à des objets distants. De plus, deux nouveaux objets spécifiques sont ajoutés, qui représentent les notions de *référence distante* (vers un objet distant) et d'*espace mémoire et de nom*. L'objet référence distante possède un composant spécialisé de *réception de message*, qui a la responsabilité d'encoder le contenu (arguments) du message en une suite d'octets, et de l'envoyer à travers le réseau jusqu'à l'objet distant. Ce dernier possède un composant spécialisé de *réception de message*, qui reconstruit et réceptionne le message. Les choix d'encodage, tels que : quel argument doit être passé par référence, par valeur (c'est-à-dire copié), jusqu'à quel niveau de profondeur, etc., peuvent être spécialisés par un *descripteur d'encodage*, détenu par le composant d'*encodage*.

### Protocoles de migration et de duplication

La migration est introduite à travers un nouveau méta-composant qui décrit les protocoles (*comment*) et les politiques (*quand*, voir au 6.7.6) de migration. La duplication, mécanisme dual, est gérée par deux nouveaux méta-composants supplémentaires. Le premier contrôle l'accès à l'état de l'objet original. Le second contrôle l'accès à l'état d'une copie. À nouveau, les choix d'encodage, tels que : quel argument doit être passé par référence, par valeur, par « déplacement » (« *call-by-move* », c'est-à-dire migré, comme dans le langage EMERALD [Jul, 1994]), avec attachements, etc., peuvent être spécialisés par un *descripteur d'encodage*, détenu par le composant de *duplication* de l'objet original. Le descripteur permet également de spécialiser les caractéristiques suivantes : quelles parties de l'objet doivent être dupliquées (permettant ainsi une duplication sélective des seules parties critiques d'un objet donné) et diverses politiques de maintien de la consistance entre l'original et ses copies.

16. Le composant de stockage par défaut se contente de passer immédiatement chaque nouveau message reçu, directement au composant d'*exécution*.

### 6.7.6 (Quelques) autres exemples d'architectures réflexives

#### Installation dynamique et composition de protocoles

L'architecture réflexive MAUD [Agha *et al.*, 1993] se concentre sur la tolérance aux fautes. Elle offre un cadre pour une *installation dynamique* et la *composition* de méta-composants spécialisés pour des protocoles, tels que duplication et transactions. Trois méta-composants (envoi des messages, réception, et état) sont considérés dans MAUD. La possibilité d'associer les méta-composants (méta-objets), non seulement à des objets, mais aussi à des méta-composants (qui sont des objets de plein droit), ouvre la voie de la composition de protocoles. Le principe est le suivant : on associe un couple de méta-composants d'envoi, et de réception de messages, respectivement à un autre couple de méta-composants. On assure ainsi une composition des protocoles par couches successives. De plus, l'association dynamique de méta-composants permet une installation uniquement au cours des besoins et pendant l'exécution du programme.

#### Contrôle de la migration

L'avantage des objets, et plus encore des objets actifs, est qu'ils sont autonomes, donc plus aisés à faire migrer d'« une seule pièce ». Néanmoins, la décision éventuelle de migrer un objet est un point d'importance et qui reste souvent la responsabilité du programmeur (par exemple en EMERALD [Jul, 1994]). Il peut être ainsi intéressant de semi-automatiser cette décision, en suivant diverses considérations, telles que les charges des différents processeurs. La réflexion permet d'intégrer de telles informations statistiques (résidentes pour les ressources physiques et partagées, telles que les charges des processeurs, ou déductibles au niveau des méta-composants pour les informations locales à l'objet, tel que le pourcentage de communications distantes), et de s'en servir pour développer divers algorithmes de migration décrits au méta-niveau dans le langage. Des exemples de tels contrôles avec toute la puissance algorithmique du langage, et indépendants du programme de l'application, sont décrits dans [Okamura et Ishikawa, 1994].

#### Spécialisation de politiques système

Le système d'exploitation réparti APERTOS [Yokote, 1992] représente un exemple significatif de système d'exploitation complètement conçu selon une architecture réflexive (à objets). En plus de la modularité et de la généricité apportées par l'utilisation d'une approche applicative objet (comme pour le système d'exploitation CHOICES, déjà décrit au § 6.5.4), la réflexion permet la *spécialisation* (éventuellement *dynamique*) du système pour les besoins d'un type donné d'application. Ainsi en APERTOS, il est relativement aisé de spécialiser la politique de séquencement, par exemple pour introduire des contraintes de type temps-réel. Un gain supplémentaire est dans la taille du noyau du système, qui est particulièrement minimal, étant réduit à la mise en œuvre des opérations réflexives de base et les abstractions de ressources de base. Ce dernier point aide donc à la compréhension et au portage du système.

**Extension réflexive d'un système commercial existant**

Une méthodologie réflexive a récemment été utilisée pour incorporer des protocoles transactionnels étendus [17] dans un système transactionnel commercial *déjà existant* [Barga et Pu, 1995]. L'idée est d'étendre le moniteur transactionnel standard pour rendre visibles un certain nombre de caractéristiques, telles que : délégation de verrou, identification de dépendances entre transactions, définition de conflits, et de les représenter par des opérations réflexives. La mise en œuvre, minimale et modulaire, repose sur l'utilisation d'« appels montants » (« *upcalls* »). Il est alors possible de mettre en œuvre divers protocoles transactionnels étendus, tels que : « split/join », « cooperative groups » [Barga et Pu, 1995], à partir des opérations réflexives.

**Le « run-time » générique comme approche duale**

Notons également que, de manière plus générale, nous assistons à un rapprochement mutuel entre langages de programmation et systèmes d'exploitation. Les langages de programmation tendent à avoir une représentation de plus en plus étendue, et de haut niveau (réflexion), du modèle d'exécution sous-jacent. De manière duale, plusieurs systèmes d'exploitation répartis offrent maintenant des couches logicielles d'exécution générique (« generic run time »), par exemple COOL [Lea *et al.*, 1993]. Ces couches génériques sont conçues pour être utilisées par divers langages, à l'aide d'un ensemble d'appels montants (upcalls), déléguant les représentations ou fonctions spécifiques au langage de programmation.

Enfin, dans un même ordre d'idée d'interfaces entre niveaux, la réflexion a également été proposée pour lier les différents niveaux (d'objet à agent) d'un système multi-agent [Ferber et Carle, 1991].

### 6.7.7 Bilan de l'approche réflexive

La réflexion offre un cadre méthodologique pour la spécialisation de modèles d'exécution parallèle et répartie, par spécialisation et *intégration* de (méta)-*bibliothèques* intimement avec le langage ou le système, tout en les décorrélant des programmes d'applications.

De nombreuses architectures réflexives sont actuellement proposées et évaluées. Nous en avons évoqué ici plusieurs, à travers diverses gammes d'applications. Il est encore trop tôt pour dégager, et valider, une architecture réflexive générale et optimale pour la programmation parallèle et répartie (bien que CODA [McAffer, 1995] nous semble un pas intéressant dans cette direction). Il nous faut pouvoir compléter notre expérience dans différents domaines d'applications, pour être mieux à même d'identifier les équilibres à trouver entre la flexibilité requise, la complexité de l'architecture, et l'efficacité résultante.

Une première critique de la réflexion porte sur la relative complexité des architectures réflexives. Mais, et indépendamment du temps d'apprentissage nécessaire pour une nouvelle culture, cela est en partie lié aux gains qu'elles offrent en matière

17. C'est-à-dire, relâchant certaines des propriétés standard (« ACID » : atomicité, consistance, isolation, durabilité) des transactions.

de flexibilité. Un problème fondamental [18] tient à la capacité de composition arbitraire de méta-composants venant d'origines diverses, mais qui peuvent contrôler des aspects ou ressources qui se recoupent. Enfin, un problème concret tient à l'efficacité, du fait des indirections et interprétations supplémentaires que la réflexion peut entraîner. Deux approches opposées pour réduire ces surcoûts tiennent en : (1) la réduction de la portée de la réflexion au niveau de la compilation, par exemple pour l'architecture OPENC++ [Chiba, 1995], ou bien (2) l'utilisation de techniques de transformation de programmes, et en particulier l'évaluation partielle [Masuhara *et al.*, 1995], pour réduire au maximum l'interprétation.

## 6.8 Conclusion

La programmation par objets est, très certainement, un des meilleurs vecteurs actuels pour le développement de systèmes et applications informatiques parallèles et réparties. Nous avons identifié et étudié trois approches qui se révèlent complémentaires.

L'approche *applicative* (par *bibliothèques*) aide à la structuration des systèmes parallèles et répartis, à l'aide des concepts objets. L'approche *intégrée* offre au programmeur un cadre conceptuel simplifié, de par l'unification qu'elle réalise entre concepts objets et concepts de la programmation parallèle et répartie. Enfin, l'approche *réflexive* offre un cadre conceptuel pour intégrer intimement des bibliothèques de protocoles avec un langage ou un système. La réflexion permet de spécialiser et d'adapter le modèle d'exécution (parallèle, réparti, tolérant aux fautes, temps-réel, etc.) d'une application, avec un minimum de modifications pour le programme de l'application.

Les développements respectifs de ces trois approches ont des objectifs complémentaires. L'approche *applicative* est destinée aux concepteurs de systèmes et vise à identifier les abstractions fondamentales des systèmes informatiques parallèles et répartis. Elle est plus particulièrement destinée aux concepteurs de systèmes. Sa principale limitation est que la programmation d'une application et de l'architecture parallèle et répartie sous-jacente sont représentées par des concepts et objets distincts. L'approche *intégrée* est destinée aux concepteurs d'applications, et vise à la définition d'un langage de programmation de haut niveau comportant un nombre minimal de concepts. Sa principale limitation tient en la possible réduction de flexibilité et d'efficacité qu'elle entraîne. L'approche *réflexive* est destinée aux concepteurs d'application qui souhaitent spécialiser le système pour les besoins propres à leur type d'application, et de manière duale aux concepteurs de système qui souhaitent ainsi concevoir leur système selon une telle architecture « ouverte » et adaptable. Elle apporte ainsi une articulation entre (et donc *ne remplace pas*) les approches applicative et intégrée, ainsi que leurs niveaux respectifs d'intervention.

18. Ce problème général, de composition arbitraire de modules logiciels, dépasse d'ailleurs de loin le seul cadre de la réflexion. La réflexion offre cependant un cadre intéressant, pour aborder en particulier la notion de modules logiciels évolutifs, qui adaptent eux-mêmes leur interface à un nouvel environnement logiciel [Kishimoto *et al.*, 1995].

# 7

# Programmation parallèle et réactive à objets

DANS LE CADRE DE LA PROGRAMMATION PARALLÈLE, répartie, ou concurrente, nous présentons un modèle et un langage mettant en œuvre le paradigme *objet*. EIFFEL//, extension du langage EIFFEL (cf. *chapitres 2 et 3*), se situe dans la catégorie des *langages de classes*, et le parallélisme qui en résulte est de type MIMD.

L'objectif et l'originalité principale du modèle proposé est la *réutilisation* : apporter à la programmation parallèle le potentiel de réutilisation des langages à objets.

La section 7.1 présente les concepts du langage EIFFEL// ; nous les illustrons par un exemple à la section 7.2. La section 7.3 démontre comment il est possible de réaliser dans le cadre impératif proposé une programmation abstraite et déclarative du contrôle des processus. La réutilisation nous permet de proposer une démarche de programmation (section 7.4). Utilisant les possibilités d'extension par bibliothèques, la section 7.5 étend le modèle d'objet actif en introduisant un modèle d'objet réactif.

## 7.1 Modèle de programmation parallèle

Cette première section présente et discute le modèle de programmation parallèle asynchrone que nous proposons, son illustration et sa mise en œuvre par le langage EIFFEL//, une extension du langage EIFFEL [Meyer, 1992a] (voir *chapitre 2*).

### 7.1.1 Exécutions concurrentes

L'expression des exécutions concurrentes est une dimension essentielle des systèmes parallèles. Nous situant dans le cadre restreint du parallélisme asynchrone, la conception d'un modèle soulève une première question : comment exprimer les activités concurrentes ?

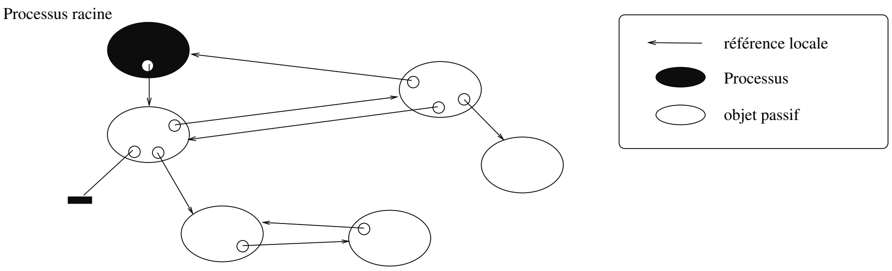



FIG. 7.1: *Système séquentiel à l'exécution.*

Une des caractéristiques de la programmation à objets est l'unification des concepts de *type* et *module* sous une seule et unique notion : la classe. Elle est à la fois un type abstrait de donnée et un module (voir *chapitre 6*).

Lorsqu'il s'agit d'ajouter le concept de processus, une unification fondamentale intervient : aucune construction spécifique n'est ajoutée au langage pour permettre l'expression de l'exécution concurrente ; la notion de processus est exprimée grâce à cette même notion de classe. Syntaxiquement, cette structure du langage va permettre d'exprimer les tâches d'un système. D'un point de vue sémantique, on obtient la conséquence suivante :

| Un processus est l'instance d'une classe ; il est typé. |
|---|

En résumé, une des contributions principales du paradigme objet à la programmation parallèle se traduit au niveau des déclarations par l'équation : *module = type = tâche*. Dynamiquement, au niveau des instances, l'équation devient : *objet = processus*. On parle alors aussi d'*objet actif*. Certains modèles de langages à objets parallèles ne présentent pas cette unification mais conservent une notion de processus orthogonale à la notion de module ; il nous semble que ces propositions n'exploitent pas un des intérêts principaux de la programmation à objets parallèle.

### 7.1.2 Structuration de l'activité

Nous venons de décider que les processus de notre modèle sont des objets. La question qui se pose maintenant est la suivante : tous les objets sont-ils des processus ?

Certains langages répondent par l'affirmative à cette question. C'est le cas du modèle acteur originel developpé en particulier par les langages PLASMA [Hewitt, 1977], ACT 1 [Lieberman, 1987] et ACT 3 [Agha, 1986], mais d'autres langages tels que ABCL/1 [Yonezawa *et al.*, 1987], ORIENT84/K [Ishikawa et Tokoro, 1987], CONCURRENT SMALLTALK [Yokote et Tokoro, 1987] et POOL [America, 1986] présentent également cette caractéristique.

Mais pour notre part, nous répondons à cette question par la négative. Sans pouvoir entrer ici dans les détails (voir [Caromel, 1991] pages 17 à 22), les raisons de ce choix sont les suivantes :

- Le premier argument est d'ordre pratique, les ressources devenant insuffisantes si tout objet, même le plus simple, doit être traité comme un processus.
- La principale raison est plutôt conceptuelle. Dans une application, les processus expriment les différentes activités parallèles. Si l'on décide que tous les objets sont des processus, la structuration et le contrôle du degré de parallélisme d'une application doivent être réalisés à un niveau d'abstraction trop faible, celui des routines par exemple.

En conséquence, la réponse à la question soulevée est la suivante :

| Les processus sont des objets, mais tous les objets ne sont pas des processus. |
|---|

Lors de la conception d'une application parallèle, décider que tel objet est un processus équivaut à préciser qu'il fonctionne de façon asynchrone par rapport au reste du système. Nous reviendrons sur les conséquences de ce choix, en particulier lors de l'étude des aspects méthodologiques (section 7.4).

### 7.1.3 Objets et processus

Un système parallèle se compose donc de deux types d'entités : des *objets* et des *processus*. Les objets restent des structures de données classiques : passifs, ils exécutent leurs routines uniquement lorsque celles-ci sont appelées par une autre entité du système. En revanche, la caractéristique d'un processus est de disposer d'une activité propre ; un système est constitué d'autant d'activités parallèles que de processus. De façon schématique, nous pouvons résumer la notion de processus par l'équation suivante : *processus = objet + activité propre*.

Dans un système séquentiel (figure 7.1), un programme est un ensemble d'objets munis d'un seul processus : l'objet racine du système. L'activité commence par l'exécution de la routine de création de cet objet (par exemple `Create` en EIFFEL). Le contrôle d'exécution se propage ensuite dans l'ensemble du système par le biais des appels de routines.

Avec l'introduction du parallélisme, un système parallèle comprend objets et processus (figure 7.2). Il n'y a plus un processus mais un ensemble de processus, chacun possédant une activité propre.

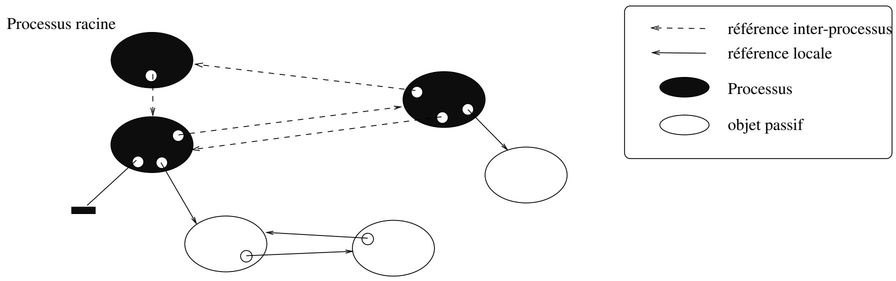



FIG. 7.2: *Système parallèle à l'exécution.*

### 7.1.4 La classe PROCESS

Rappelons, sous une autre forme, l'orientation qui a été prise : certaines classes s'instancient en processus. Un mécanisme doit permettre de définir les classes dont les instances sont des processus.

Nous utilisons l'héritage ; la notion de processus est représentée par une classe particulière : PROCESS. Un processus est :

| l'instance d'une classe qui hérite directement ou indirectement de la classe PROCESS. |
|---|

La figure 7.3 présente cette classe. Lors de la conception d'une classe, s'il apparaît que ses instances doivent être des processus, il suffit alors que cette classe hérite de PROCESS. Lorsque le langage permet l'héritage multiple (c'est le cas pour EIFFEL), ce mécanisme n'entre pas en conflit avec d'autres utilisations de l'héritage (spécialisation, généralisation, implémentation).

Après son initialisation, un objet processus exécute sa routine Live : elle représente le corps du processus. Lorsque cette routine prend fin, le processus se termine également. Notons que la routine Live définit un comportement par défaut : un service FIFO des points d'entrés du processus. Nous verrons ultérieurement comment définir un comportement spécifique (section 1.6).

Enfin, remarquons une dernière simplification due à l'unification objet/processus : la notion de terminaison des processus peut être réalisée selon le principe du ramasse-miettes. Un attribut particulier (I_am_alone) reflète le fait qu'un processus n'est plus référencé par aucun autre processus actif. Dans ce cas, le processus peut, si tel est le comportement recherché, se terminer. Ceci représente également la politique par défaut.

```
class PROCESS
    -- Toute instance d'un heritier de cette classe est un processus
feature
   Live is -- Corps par defaut des processus:
           -- service FIFO des points d'entrees, attente non active
      do
         from
         until I_am_alone
         loop
            <Service FIFO des requetes>
         end;
      end;
end -- PROCESS
```

FIG. 7.3: PROCESS *: la classe processus.*

### 7.1.5 Communication : la syntaxe

Un processus est un objet. En ce sens, il présente des routines exportées, les services que ce processus offre à ses clients. Ce mécanisme d'appel de routine, fondement de la programmation séquentielle, va également être utilisé pour la communication entre processus. Une syntaxe classique va permettre de noter l'appel des routines d'un processus : la *notation pointée*.

Plus précisément, soit `P` une classe héritière de `PROCESS` et `p` une entité declarée de type `P`. Si `f` est une routine déclarée et exportée de `P`, alors la notation

```
p.f ( ... );
```

dénote une communication entre le processus courant et le processus `p`.

L'unification est donc la suivante :

| Communication entre processus = Appel de routine |
|---|

Une banalisation des communications entre les processus constitue un des apports spécifiques de la programmation à objets. L'origine de l'unification peut être vue dans le modèle acteur développé par Carl Hewitt [Hewitt, 1977]. Depuis ces travaux, de nombreux langages présentent l'unification *appel de routine/communication entre processus*. Citons par exemple les langages ABCL/1, ORIENT84/K, CONCURRENT SMALLTALK, POOL et HYBRID [Nierstrasz, 1987].

Cependant, notre modèle apporte une dimension supplémentaire : l'utilisation de l'héritage (classe `PROCESS`) pour structurer le parallélisme. Il est possible de réaliser une *affectation polymorphe* entre objet et processus : une entité qui n'est pas un processus, va pouvoir dynamiquement référencer un processus. La liaison dynamique va donc se produire entre un objet et un processus : ce qui, statiquement, n'est qu'un simple appel de routine va, à l'exécution, donner lieu à une communication entre deux processus. Un tel mécanisme contribue très fortement à la souplesse du modèle proposé : il autorise la réutilisation de codes pré-existants lors de la construction des systèmes parallèles. Il permet également la transformation d'un système séquentiel en un système parallèle ; nous détaillerons cette dernière possibilité ultérieurement.

### 7.1.6 Sémantique de la communication

Le principe de la sémantique d'une communication entre processus consiste à décomposer celle-ci en 3 phases [1] :

1. une interruption de l'appelé par l'appelant,
2. un rendez-vous pour la transmission de la requête à l'appelé,
3. ultérieurement un service asynchrone de la requête.

Notons tout d'abord que cette communication est globalement *asynchrone* ; l'appelant n'est pas bloqué en attendant le service de la requête par l'appelé. Ce premier choix se justifie par un souci de *réutilisation* et de *structuration du parallélisme* : décider de l'asynchronisme au niveau de la classe permet de réaliser une parallélisation globale, et facilite ainsi la réutilisation.

1. Pour plus de détails sur ces choix, voir [Caromel, 1991], pages 30 à 36.

D'autre part, les deux premières phases nous permettent d'obtenir les avantages de deux mondes apparemment incompatibles, celui de la communication à base de *rendez-vous* et celui de l'envoi asynchrone de messages. Les langages qui utilisent le rendez-vous (CSP [Hoare, 1978 ; Hoare, 1987], ADA [Ichbiah *et al.*, 1979], OCCAM 2 [INMOS Limited, 1988] ...) présentent toujours une communication synchrone, alors que les langages à base d'envoi de messages asynchrones (ACT 3, ABCL/1, CONCURRENT SMALLTALK...) n'incorporent pas le concept de rendez-vous. Notons que la notion de *séquence* du langage LC3 [Banatre, 1990] permet également une certaine synthèse entre communications synchrone et asynchrone.

### Transmission synchrone

Ce paragraphe présente la mise en œuvre des deux premières étapes de la communication. Lorsqu'un processus envoie une requête à un autre processus, il doit interrompre ce dernier afin d'entrer en rendez-vous avec lui : rendez-vous pendant lequel la requête est transmise.

Plus précisement, un objet `o` qui exécute l'instruction `p.f ( ...)`, où `p` référence dynamiquement un processus, déclenche les opérations suivantes :

1. Le processus `p` est interrompu.
2. Un rendez-vous s'établit entre `o` et `p`.
3. Un objet de type `REQUEST` représentant l'appel de `f` est transmis de `o` vers `p` (figure 7.4).
4. Le rendez-vous se termine : `o` exécute l'instruction suivante, `p` reprend son activité précédente.

Le mécanisme d'exception utilisé ne présente aucune spécificité et nous ne le détaillerons pas ici. Notons simplement que lorsqu'une exception correspondant à une communication est déclenchée chez l'appelé, un récupérateur d'exception (*handler*) spécifique, `receive_request` est exécuté. Pendant l'exécution de ce récupérateur, l'objet de type `REQUEST` est transmis de l'appelant vers le contexte du processus appelé. Celui-ci, à la fin du récupérateur, peut reprendre son cours où l'exception l'avait interrompu.

La classe `REQUEST` permet quant à elle de modéliser un appel de routine ainsi que son service. L'attribut `feat` représente la routine à exécuter, `parameters` la liste des paramètres effectifs. La routine `serve` permet le service de la requête : la routine cible est exécutée avec les paramètres effectifs. Un opérateur particulier (**&**) appliqué à une routine retourne un objet du type `ROUTINE` ; celui-ci représente une modélisation de la routine qui peut ainsi être manipulée (passée en paramètre, etc.). Ce type de technique s'apparente à la *réflexivité*. Plus précisément, la modélisation des requêtes et des routines correspond à une *concrétisation de messages* (*message reification* [Ferber, 1989a ; Foote et Johnson, 1989]) : les appels de routines ainsi que les routines sont modélisés par des classes particulières, et peuvent ainsi être manipulés par le programmeur en tant qu'objets instances de ces classes.

```
class REQUEST  -- Modelisation d'une requete envoyee a un processus
export
   feat, parameters, time, serve
feature
   feat: ROUTINE;  -- Routine destinatrice de la requete
   parameters: LIST [ANY];  -- Parametres effectifs
   time: REAL;  -- Moment d'arrivee de la requete (temps local)
   serve is   -- Sert la requete:
              -- Execute la routine avec les parametres effectifs
     do ... end;
end -- REQUEST
```

FIG. 7.4: `REQUEST` *: la modélisation des requêtes.*

**Service asynchrone**

Considérant que la requête est dans le contexte du processus appelé, il s'agit maintenant de définir comment la *servir*, à savoir :

1. Sélection de la requête à servir.
2. Exécution de la routine correspondante avec les paramètres effectifs contenus dans la requête.

Les requêtes en attente de service sont mémorisées dans une structure de données classique, une liste chaînée par exemple. Afin de sélectionner une requête, il suffit donc de parcourir cette liste. Toutefois, la sémantique de la communication ajoute une contrainte : les accès à cette liste doivent être protégés.

En effet, l'envoi d'une requête est réalisé par une exception (cf. la phase synchrone de la communication). Au cours de celle-ci, la principale action est l'ajout d'un élément à la liste (une nouvelle requête en attente de service). Si une exception se produit lorsque le processus parcourt la liste pour sélectionner une requête à servir, alors il y a un risque d'incohérence. La solution à ce problème consiste à se protéger de l'exception utilisée pour la communication entre les processus chaque fois que l'on doit parcourir ou modifier la liste des requêtes.

Une routine permettant à un processus de servir une requête aura donc la structure générale suivante :

1. Protection contre l'arrivée éventuelle de nouvelles requêtes.
2. Sélection d'une requête à servir.
3. Sortie du mode protégé.
4. Service de la requête sélectionnée.

Une routine réalisant la sélection et le service de la plus ancienne requête envoyée à une routine donnée, aura donc la structure suivante :

```
serve_oldest_of (feat: ROUTINE) is
      -- Sert la plus ancienne requete sur feat
```

```
local
   w_request: REQUEST;
do
   <Se proteger contre l'arrivee de nouvelles requetes>
   <Selectionner "w_request": la plus ancienne requete sur "feat">
   <La retirer de la liste>
   <Sortir du mode protege>
    if not w_request.Void then
      w_request.serve;  -- service
   end;
end; -- serve_oldest_of
```

La phase de service (4) est réalisée grâce à la routine `serve` de la classe `REQUEST` (figure 7.4). Si la variable `w_request` contient la requête à servir, le service se traduit simplement par l'instruction `w_request.serve`.

Lors de la programmation d'un processus, ce type de routine est directement accessible ; on les utilisera afin de programmer le corps du processus. Par exemple, dans une classe `BUFFER`, le service de la plus ancienne requête reçue sur la routine `put` sera exprimé par : `serve_oldest_of (& put);` .

Notons que ce service est non-bloquant. Si aucune requête adressée à la routine `feat` n'a été trouvée, alors aucun service n'est réalisé et l'exécution continue. Cependant, des services bloquants sont tout aussi nécessaires. Ils sont également programmables : lorsqu'aucune requête à servir n'est disponible, il suffit d'attendre qu'il y en ait une. Une instruction particulière est utilisée : elle permet d'attendre l'arrivée d'une nouvelle requête.

Ces deux types de services (bloquant et non-bloquant) ne sont pas les seuls. Notons par exemple des services avec attente limitée, ou encore des services où l'on sélectionne la requête à servir en fonction de la valeur des paramètres effectifs. Il est également possible de programmer des routines qui ne sont pas des routines de service, mais qui sont néanmoins très utiles à la programmation du contrôle des processus : extraction d'informations, mise en attente, etc. Nous présentons ici quelques exemples de services rencontrés fréquemment :

**Services non-bloquants :**
`serve_oldest` : sert la requête la plus ancienne de toutes.
`serve_oldest_of (f)` : sert la plus ancienne requête portant sur `f`.
`serve_oldest_ofn (l)` : la plus ancienne sur une routine de la liste `l`.
...

**Services bloquants :**
`bl_serve_oldest` : sert la requête la plus ancienne de toutes.
`bl_serve_oldest_of (f)` : sert la plus ancienne requête portant sur `f`.
...

**Services bloquants avec attente limitée :**
`timed_serve_oldest (t)` : sert la requête la plus ancienne de toutes.
`timed_serve_oldest_of (f, t)` : sert la plus ancienne requête portant sur `f`.
...

**Extraction d'informations :**

`request_nb` : nombre de requêtes en attente.
`request_nb_of (f)` : nombre de requêtes en attente portant sur `f`.
...
**Mise en attente :**
`wait_a_request` : attente d'une nouvelle requête à servir.
`wait_a_request_of` : attente d'une requête à servir portant sur `f`.
...

Tous ces services sont donc disponibles pour la programmation des processus. Mais beaucoup d'autres encore sont envisageables. Un des points forts d'un tel mécanisme de bibliothèque est qu'il s'enrichit au fur et à mesure de son utilisation : si l'utilisateur ne trouve pas la spécification qu'il cherche, il a la possibilité d'écrire lui-même la routine dont il a besoin en utilisant le modèle de la routine `serve_oldest_of` présentée plus haut.

La figure 5 présente une classe *buffer non borné*. Elle hérite de la classe `PROCESS` ; ses instances seront des processus. Elle hérite également de la classe `LINKED_LIST`, structure de donnée non bornée, qui permet de réaliser le buffer. Les routines `put` et `get` sont héritées de `LINKED_LIST` et leur utilisation ne nécessite aucune redéfinition ; la classe `BUFFER` se contente de les exporter. La synchronisation par défaut FIFO ne convenant pas ici, la routine `Live` est redéfinie.

Le corps du processus est constitué d'une boucle. Les services sont réalisés par la routine non-bloquante `serve_oldest_of`. La politique est définie de telle sorte

```
class BUFFER [T]  -- Une classe processus, buffer non-borne
export
   put, get
inherit
   PROCESS
      redefine Live;
   LINKED_LIST [T];
feature
   Live is     -- Corps du processus:
      do
         from
         until I_am_alone
         loop
            serve_oldest_of (& put);
            if not empty then
               serve_oldest_of (& get);
            end;
            wait_a_request;
         end;
      end; -- Live
end -- BUFFER
```

FIG. 7.5: *Exemple d'une classe* `BUFFER` *non borné* .

que dans le cas où le buffer n'est ni plein, ni vide, des requêtes sur put et sur get sont en attente de service ; alors, on teste *alternativement* si des requêtes sur put et get sont présentes et à servir. Par contre, si l'on souhaite servir les requêtes selon *l'ordre chronologique d'arrivée*, alors la synchronisation pourra s'exprimer ainsi :

```
if not empty then
       serve_oldest_ofn (& put, & get);
else
       serve_oldest_of (& put);
end;
```

Cette variation constitue un exemple de la finesse offerte par une programmation explicite du contrôle.

Pour conclure cette section, notons que le langage ne comporte pas d'instruction non-déterministe du type `select` d'ADA, également présente dans d'autres langages (CSP : [... ], OCCAM 2 : ALT, CONCURRENT C [Gehani et Roome, 1989] : select, etc. ). Un des problèmes d'une telle instruction réside dans son caractère limitatif : ce n'est qu'une solution — ou *abstraction* — parmi d'autres pour programmer le contrôle des processus. La proposer comme mécanisme élémentaire nous prive d'autres solutions et nous impose ses limitations ; on trouvera une liste des problèmes rencontrés dans [Le Verrand, 1982 ; Gehani, 1984 ; Burns, 1985]. Par opposition, nous utilisons les structures de contrôle classiques (boucles, tests, routines) pour programmer le corps des processus, ce qui apporte toute la liberté et la puissance d'expression nécessaires. D'autre part, nous verrons à la section 2 comment *programmer* de telles abstractions, intégrant *non-déterminisme* et programmation *déclarative* du contrôle.

### 7.1.7 Synchronisation : attente par nécessité

Nous avons déterminé à la section précédente que la communication entre processus est toujours asynchrone : lorsqu'un objet envoie une requête à un processus, il poursuit son chemin sans attendre l'exécution du service demandé. Une telle sémantique, systématiquement asynchrone, a été très peu utilisée. Le seul exemple semble être le modèle d'acteurs ; mais uniquement le modèle originel pur [Hewitt, 1977 ; Agha et Hewitt, 1987], car les langages dérivés (ABCL/1, ORIENT84/K, CONCURRENT SMALLTALK, etc.) comportent également une communication synchrone. La raison en est simple : avec un tel modèle, la maitrise du parallélisme est difficile, voire impossible. De par la nature asynchrone des communications, celles-ci ne peuvent servir à coordonner les activités des différents processus. Un modèle systématiquement asynchrone manque de synchronisation !

En effet, les appels de fonctions sont, eux aussi, réalisés selon un mode asynchrone. Prenons l'exemple suivant où b référence un processus buffer (figure 7.5) :

```
v := b.get ( ... );
 ...
v.print;
```

Lorsque le contrôle atteint l'instruction `v.print`, il n'est pas sûr que la valeur v soit déjà disponible. Le processus b n'a peut-être pas encore réalisé le service demandé : exécuter `get` et renvoyer le résultat dans `v`. Dans la situation actuelle, il serait donc nécessaire d'ajouter entre ces deux instructions un élément de synchronisation permettant d'attendre que la valeur `v` soit disponible. Cette solution n'est pas acceptable. On perdrait alors tout l'avantage de l'asynchronisme (augmenter le parallélisme sans changer le code), puisqu'il faudrait modifier les programmes existants.

La solution à ce problème est aussi simple que puissante :

| |
|---|
| Un processus attend automatiquement lorsqu'il tente d'utiliser une valeur qui n'est pas encore disponible. |

Nous appelons ce principe l'*attente par nécessité* : un processus n'attend que s'il ne peut pas faire autrement. Lors de l'exécution d'une instruction

```
v := p.f ( ... );
```

(`p` référence un processus), le traitement se poursuit sans que la valeur de `v` soit déjà disponible ; `v` est alors appelé un *objet attendu*. C'est uniquement si l'on tente d'utiliser un objet attendu que l'on est mis en attente. Dès l'instant où la valeur est rendue disponible, renvoyée par `p`, l'exécution reprend. Si l'on ne tente pas d'utiliser la valeur `v` avant le retour de celle-ci, il n'y a pas d'attente.

Cette notion d'utilisation d'un objet est très importante. Elle signifie « utiliser l'objet lui-même », par opposition à l'utilisation de sa référence. Aussi, lorsqu'un objet attendu est affecté à une autre entité ou qu'il est passé en paramètre d'une routine, seule sa référence est utilisée : cela ne provoque pas d'attente. Un accès à l'objet lui-même se produit dans deux cas : (1) l'appel de l'une de ses routines, (2) l'accès à l'un de ses attributs. Syntaxiquement, cela se traduit par une notation pointée de cet objet (`v.r`).

Il est possible d'inclure des appels à des processus dans les expressions. Dans ce cas, l'attente par nécessité joue également son rôle de synchronisation. Par exemple :

```
r := obj.rout or p.f;
p.f.print;
```

Dans ces deux cas, le résultat de l'appel asynchrone est utilisé immédiatement afin de poursuivre l'évaluation de l'expression. En conséquence, une attente immédiate du résultat se produit.

La notion de *futur* (ABCL/1, [Halstead, 1985], etc.) recouvre aussi l'idée d'une variable résultat d'un calcul pas encore terminé. Elle est proche de celle d'attente par nécessité mais demande cependant une programmation explicite : un programmeur souhaitant employer une variable comme futur est obligé de le préciser. Syntaxiquement, cela se traduit toujours par une construction spécifique qu'il est nécessaire d'ajouter dans le texte du programme. En revanche, l'attente par nécessité ne requiert aucune notation particulière. Cette différence, au demeurant très faible et qui pourrait sembler uniquement syntaxique, a de très importantes répercussions sur un objectif important : la réutilisation dans le cadre de la programmation parallèle. Enfin, remarquons que l'attente par nécessité facilite également l'écriture directe de programmes parallèles ; le programmeur est déchargé de l'effort supplémentaire de codage imposé par l'utilisation de variables ou de notations spécifiques.

**Autre cas d'attente par nécessité**

Lors de l'appel de la routine d'un processus, la transmission des paramètres n'est plus effectuée par référence, mais *par copie* :

| Entre processus, les objets passifs sont transmis par copie. |
|---|

Ce choix (voir [Caromel, 1991] pages 50 à 54 pour sa justification) permet d'obtenir par défaut et automatiquement des systèmes sans objets partagés: chaque objet passif est accessible par un seul objet actif. La section « Démarche de conception » (section 7.4) détaille les implications de cette stratégie. Les processus, quant à eux, sont bien sûr toujours transmis par référence.

Cette règle peut provoquer une nouvelle situation d'attente automatique. Celle-ci se produit dans la situation suivante : un objet attendu est en paramètre d'un appel à un processus. Lors de l'exécution d'une instruction `p.g(obj)`, si

1. `obj` est un objet attendu ou
2. `obj` référence directement ou indirectement un objet attendu,

alors une attente par nécessité se produit : l'exécution est retardée jusqu'à ce que l'objet attendu ne le soit plus. En effet, contrairement à une transmission habituelle de paramètres où seule la référence des objets est utilisée, la transmission par copie nécessite un accès à la valeur même des objets.

**Attente explicite**

Il est parfois nécessaire de programmer une synchronisation qui ne découle pas de l'utilisation d'une donnée. Deux routines universelles, définies par exemple dans la classe `ANY` et donc applicables à tout objet, permettent de telles opérations : `Wait` et `Awaited`.

Dans le programme suivant :

```
v := p.f ( ... );
 ...
v.Wait;
```

l'instruction `v.Wait` est une attente explicite de la valeur `v`. Remarquons que l'instruction `p.f( ... ).Wait;` permet de réaliser l'équivalent d'une communication synchrone.

La routine universelle `Awaited` permet de tester l'état (attendu ou non) d'un objet. Un exemple d'utilisation est le suivant :

```
v := p.f ( ... );
from
until not v.Awaited
loop
 ... -- Actions à réaliser pendant l'attente de v
end;
< utilisation de v>
```

Après l'appel asynchrone `v := p.f ( ... )`, la boucle permet de programmer des opérations à réaliser en attendant l'objet `v` ; le temps d'attente est mis à profit afin de réaliser d'autres actions.

### 7.1.8 Résumé

La présentation du modèle de programmation parallèle est maintenant complète ; pour plus de détails, voir [Caromel, 1991]. Cette section en résume les principales caractéristiques :

- Un processus est l'instance d'une classe qui hérite directement ou indirectement de la classe `PROCESS`.
- Le polymorphisme joue entre objets et processus.
- La sémantique de la communication comporte trois phases :
  Interruption — Rendez-vous — Asynchronisme.
- Une bibliothèque de services types permet de programmer le corps des processus.
- L'attente par nécessité synchronise les processus entre eux.

## 7.2 Arbre binaire de recherche

À titre d'exemple, cette section présente comment programmer de manière parallèle l'accès à une structure de données récursive, en l'occurrence un *arbre binaire de recherche*.

### 7.2.1 Version séquentielle

L'organisation de ce type de structure consiste à associer à chaque information conservée une clé utilisée par des opérations d'*insertion* et de *recherche*, l'ensemble de ces clés pouvant être trié selon un ordre total.

Les informations sont organisées sous une forme arborescente. Chaque nœud de l'arbre contient : une information, sa clé associée, une référence sur un fils gauche, une référence sur un fils droit.

De plus, l'invariant suivant existe : toutes les clés du sous-arbre *gauche* (respectivement *droit*) sont *inférieures* (resp. *supérieures*) à la clé courante. La classe `BTREE` (figure 7.6) répond à cette spécification. Elle est basée sur deux paramètres génériques :

- `KEY`, le type de la clé : il est contraint à la conformité avec la classe `COMPARABLE` afin d'assurer qu'il dispose d'une relation d'ordre total (nous emploierons les opérations inférieur à, `lt`, et égal à, `equal`) ;
- `INFO`, le type des informations conservées dans chaque nœud.

Deux routines exportées `insert` et `search` permettent d'utiliser la table. Les utilisateurs référencent le nœud racine de l'arbre et exécutent des requêtes du type :

```
bt.insert(k,v);
...
value := bt.search(k);
```

```
class BTREE [KEY->COMPARABLE, INFO]
 export
   insert, search, left {BTREE}, right {BTREE}
 feature
  local_key: KEY;  local_info: INFO;
  left, right: like current;
  insert(k: KEY; i: INFO) is
   do
     if local_key.Void then
        local_key := k;   local_info := i;
        left.Create; right.Create;
     elsif k.equal(local_key) then
        local_info := i;
     elsif k.lt(local_key) then
        left.insert(k, i);             -- (1)
     else
        right.insert(k, i);            -- (2)
     end;
   end;  -- insert
  search(k: KEY): INFO is
   do
     if local_key.Void then
        result := <Pas trouve>;
     elsif k.equal(local_key) then
        result := local_info;
     elsif k.lt(local_key) then
        result := left.search(k);        -- (3)
     else
        result := right.search(k);       -- (4)
     end;
   end;  -- search
end;  -- class BTREE
```

FIG. 7.6: BTREE *: arbre binaire de recherche séquentiel.*

Une telle requête est :

- soit satisfaite sur le nœud courant,
- soit récursivement répercutée sur l'un des deux fils, par exemple le gauche, par une instruction du type : `left.insert(k,v);`

Une table vide est constituée d'un nœud unique avec une clé à `void`. À chaque feuille correspond un nœud sans aucune information, comportant lui aussi une clé à `void`.

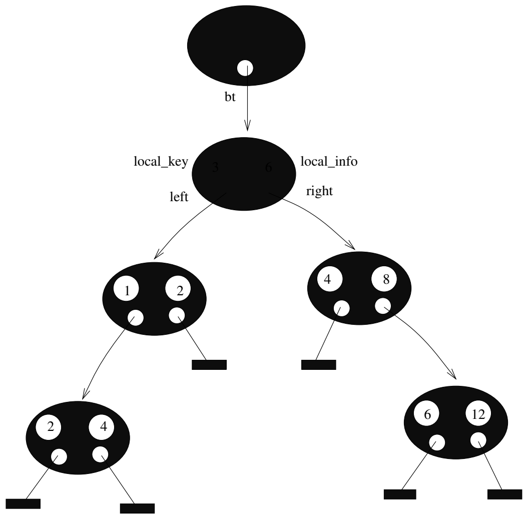



FIG. 7.7: *Arbre binaire parallèle à l'exécution.*

### 7.2.2 Parallélisation

Le principe de parallélisation est simple (figure 7.7) : chaque nœud de l'arbre devient un processus. Ainsi, plusieurs requêtes vont pouvoir progresser en parallèle à l'intérieur de l'arbre.

#### Programmation parallèle

La classe `P_BTREE` qui correspond à un nœud actif est définie uniquement par héritage des classes `BTREE` et `PROCESS` (figure 7.8). Aucune autre programmation n'est nécessaire, comme nous allons le montrer.

```
class P_BTREE [KEY->COMPARABLE, INFO]
 export
   repeat BTREE
 inherit
   BTREE[KEY,INFO];
   PROCESS
 feature
end;  -- class P_BTREE
```

FIG. 7.8: `P_BTREE` *: arbre binaire de recherche parallèle.*

En effet, chaque nœud étant un processus, les appels de routines sur `insert` et `search` deviennent automatiquement et systématiquement *asynchrones*. Lorsqu'un utilisateur qui référence le nœud racine de l'arbre exécute les instructions :

```
bt.insert(k1, v1);
bt.insert(k2, v2);
...
bt.insert(kn, vn);
```

il ne reste pas bloqué pendant l'exécution complète de chaque requête. L'utilisateur peut poursuivre l'exécution dès que la requête est transmise au processus racine de l'arbre et avant que toutes les opérations soient effectivement terminées. Même si elles ne peuvent être traitées immédiatement, les requêtes sont mémorisées dans le processus racine de l'arbre.

Dans le cas d'une recherche, le fonctionnement demeure également asynchrone :

```
value1 := bt.search(k3);
value2 := bt.search(k4);
...
value1.print;    -- attente par nécessité possible
```

et dans ce cas, la synchronisation sera réalisée automatiquement et seulement si le besoin s'en fait sentir grâce au mécanisme d'attente par nécessité.

L'arbre de recherche se comporte de manière asynchrone vis-à-vis de ses clients, mais possède également un comportement interne parallèle. Dans le cas d'une insertion, lorsqu'un nœud exécute l'instruction

```
left.insert(k1, v1);
```

il transmet la requête à son fils gauche. Celui-ci étant lui-même un processus, l'appel est aussi asynchrone. Ainsi le nœud se retrouve-t-il à nouveau libre de traiter une nouvelle requête, pouvant par exemple déclencher un appel sur son fils droit. À un instant donné, plusieurs requêtes peuvent donc être traitées en parallèle dans l'arbre ; leur nombre (et donc le parallélisme maximum obtenu) est d'ailleurs au plus égal au nombre de nœuds de l'arbre.

Dans le cas d'une recherche, le traitement en parallèle des requêtes repose sur un mécanisme que nous n'avons pas encore présenté : les *continuations automatiques*. Après qu'un nœud ait envoyé une requête `search` à l'un de ses nœuds fils, il doit renvoyer un résultat de sa recherche ; or son fils n'a pas instantanément achevé sa propre recherche récursive.

Le mécanisme mis en œuvre afin d'éviter toute attente inutile est le suivant :

| |
|---|
| Une continuation automatique se déclenche lorsque le résultat que doit renvoyer un processus est encore attendu : le processus qui sert la requête n'est cependant pas bloqué et peut servir de nouvelles requêtes. L'arrivée ultérieure du résultat de la requête renverra automatiquement le résultat attendu au niveau du client. |

Avec ce mécanisme, il est donc également possible d'effectuer autant de recherches simultanées qu'il existe de nœuds dans l'arbre et ceci sans modification du code de la routine `search`.

La définition de la classe `P_BTREE` ne comporte pas de synchronisation (routine `Live`), c'est-à-dire que les requêtes y sont simplement traitées selon la synchronisation FIFO héritée par défaut de la classe `PROCESS`. Cette politique de service est très importante, car elle permet d'assurer qu'aucune requête ne sera dépassée par une autre, émise plus tard : l'arbre de recherche parallèle conserve ainsi la même sémantique que l'arbre séquentiel.

Bien que cet exemple soit simple, il montre que les mécanismes introduits se révèlent très puissants pour exprimer des comportements parallèles. Il montre aussi comment la bonne intégration de ces mécanismes aux concepts objet rend l'expression du comportement plus simple et plus transparente. Les schémas de parallélisation présentés dans cet exemple se retrouvent dans de nombreux problèmes et peuvent souvent être obtenus d'une façon tout aussi automatique.

## 7.3 Abstractions pour la programmation du contrôle

L'utilisation des structures de contrôle classiques nous permet d'exprimer de façon très fine le comportement des processus (figure 7.5). Ceci est très utile lorsque des optimisations sont nécessaires ou encore dans le cadre de systèmes à caractère temps-réel. Cependant, on peut quelquefois vouloir s'abstraire de telles considérations et travailler dans un cadre de plus haut niveau. Cette section démontre comment le cadre présenté autorise ce type de programmation parallèle. Nous présentons la technique permettant la programmation du contrôle des processus selon un style *implicite*, voire *déclaratif*.

### 7.3.1 Définition et exemples

Nous définirons une abstraction comme *un cadre particulier permettant de programmer le contrôle des processus*. Une abstraction offre au programmeur qui l'utilise une certaine vision de la programmation du contrôle.

Quelques langages proposent de telles abstractions. Citons en particulier les *compteurs de synchronisation* [Robert et Verjus, 1977], le concept de *médiateur* [Grass et Campbell, 1986], les *conditions d'activation* du système GUIDE [Decouchant *et al.*, 1989], les *ensembles d'activation* du prototype ROSETTE [Tomlinson et Singh, 1989], la notion de *comportement abstrait* du langage ACT++ [Kafura et Lee, 1989a ; Kafura et Lee, 1989b], les *fonctions de planification* (*scheduling functions*) et les *pré/post ambules* (*pre-post ambles*) [McHale, 1994]. Notons également la spécification de synchronisation par *expression de chemins* [Campbell et Haberman, 1974].

Dans le cadre du système Guide par exemple, un buffer borné s'exprime de la façon suivante :

```
CLASS buffer IS
  ...
CONTROL
   put: (completed (put) - completed (get)<size) and current (put) = 0;
   get: (completed (put) < completed (get))      and current (get) = 0;
END buffer
```

La partie `CONTROL` permet d'exprimer les conditions d'activation des routines `put` et `get`. Les compteurs utilisés, `completed` et `current`, ont pour valeurs respectives le nombre d'exécutions « terminées » et « en cours » pour une routine. Dans le langage ACT++, le même buffer s'écrit :

```
class buffer: Actor {
behavior:
   empty_buffer = {put ()};
   full_buffer = {get ()};
   partial_buffer = {put (), get ()};
public:
  ...
};
```

Ici, c'est la section BEHAVIOR qui définit la synchronisation. Pour chacun des trois états définis (*empty_buffer*, *full_buffer* et *partial_buffer*), les routines pouvant y être activées sont précisées.

De par la nature *déclarative* des conditions de synchronisation, de telles programmations sont en effet plus abstraites qu'une définition impérative de l'activité d'un processus. Étant plus proches d'une spécification formelle, elles facilitent en particulier la construction de preuves [Dijkstra, 1975 ; Hoare, 1985 ; Hoare, 1987]. Le problème de telles abstractions provient du fait que chacune d'elles est proposée comme cadre élémentaire et surtout unique permettant la programmation du contrôle des processus. Outre le fait, comme nous l'avons vu précédemment, qu'il est parfois nécessaire d'avoir un contrôle plus fin sur l'exécution, il existe toujours des cas pratiques qui ne rentrent pas, ou mal, dans le cadre du formalisme proposé. Le programmeur est alors limité ou contraint à une expression maladroite. Le simple fait qu'il existe autant d'abstractions différentes démontre qu'aucune n'est universelle.

### 7.3.2 Programmer une abstraction

Notre cadre de programmation parallèle nous permet de programmer de telles abstractions. Par exemple, définissons une abstraction qui va permettre d'associer une *condition de blocage* à une routine : lorsque cette condition est vraie, on ne sert pas les requêtes portant sur cette routine. Pour ce faire, on définit un processus particulier (ABSTRACT_PROCESS, figure 7.9). Quand on souhaitera programmer des applications parallèles selon cette abstraction, on héritera de cette classe au lieu de PROCESS.

La programmation de la classe ABSTRACT_PROCESS est relativement simple :

- une h-table (addressage associatif) mémorise les couples (routine à synchroniser, condition) ;
- une routine (associate) permet de définir de tels couples ;
- une routine (synchronization), définie par les utilisateurs de l'abstraction, permet de spécifier la synchronisation (par le biais de la routine associate) ;
- la routine Live sert une requête si, après évaluation, sa condition de blocage est à faux. Dans le cas contraire, la requête est remise dans la file d'attente ; on tentera ultérieurement de la servir à nouveau.

```
class ABSTRACT_PROCESS-- Exemple de processus dote d'une abstraction
                      -- du controle (gardes de methodes)
inherit
   PROCESS
      redefine Live
feature
   l_sync: HTABLE [ROUTINE, STRING];
      -- Une h-table memorise les synchronisations
   Live is  -- Synchronisation du processus: on sert une requete
            -- si sa condition bloquante n'est pas vraie.
    local req: REQUEST; r: ROUTINE;
      do
         synchronization;
         from until I_am_alone
         loop
            from <debut de la liste des requetes>
            until <fin de la liste>
            loop  req := <requete courante>;
                  r := l_sync.item (req.name);
                    -- routine de bloquage
                  if not r.execute then -- Test
                    <servir requete courante>;
                    <enlever la requete courante>;
                  end;
            end; -- loop
         end; -- loop
      end; -- Live
   associate (rout: ROUTINE, sync: ROUTINE) is
     -- Associe une condition a une routine
      do
         l_sync.put (rout.name, sync);
      end;
   synchronization is  -- Synchronisation par defaut:
                       -- aucune condition, service FIFO.
      do end;
end -- ABSTRACT_PROCESS
```

FIG. 7.9: ABSTRACT_PROCESS *: un exemple d'abstraction* .

Ce mécanisme permet, par exemple, de définir un buffer borné (classe ABST_BUFFER, figure 7.10). Cette classe hérite de l'abstraction définie figure 7.9 (ABSTRACT_PROCESS). Elle hérite également de la classe FIXED_LIST d'où proviennent les routines exportées (put, get), et celles utilisées pour la synchronisation (full, empty). La synchronisation est réalisée par l'association des routines put et get à leur routine de synchronisation, respectivement full et empty.

Une abstraction aussi simple est, bien sûr, d'utilisation limitée. Mais il est possible de sophistiquer le mécanisme à volonté. On remarquera qu'une instruction de

```
class ABST_BUFFER [T]  -- Un buffer defini en declaratif
export
   put, get
inherit
   ABSTRACT_PROCESS
      redefine synchronization;
   FIXED_LIST [T]
feature
   synchronization is  -- La synchronisation du buffer
      do
         associate (& put, & full);
         associate (& get, & empty);
      end;
end -- ABST_BUFFER
```

FIG. 7.10: `ABST_BUFFER` *: une définition déclarative d'un buffer .*

communication sélective avec non-déterminisme, comme le `select` d'ADA, n'apparaît que comme une abstraction parmi beaucoup d'autres. Elle peut, elle aussi, être programmée.

### 7.3.3 Les abstractions et la réutilisation

Le modèle développé dans le cadre de ce travail favorise la réutilisation de routines. Mais en plus de ce type de réutilisation présenté en section 7.2, il est également intéressant de réutiliser le contrôle lui-même (le corps des processus). Cependant, une programmation explicite du contrôle (figure 7.5) ne favorise pas cette réutilisation ; ce problème a été identifié et discuté dans différents travaux [Briot et Yonezawa, 1987 ; America, 1987 ; Decouchant *et al.*, 1989 ; Papathomas, 1989 ; Tomlinson et Singh, 1989 ; Kafura et Lee, 1989a ; Kafura et Lee, 1989b ; Frolund, 1992 ; Matsuoka et Yonezawa, 1993 ; Meseguer, 1993 ; Bergmans, 1994]. En effet, la nature impérative du contrôle explicite oblige souvent à réécrire complètement le corps d'un processus lorsque la spécification change, même légèrement. L'abstraction présentée à la section précédente autorise une telle réutilisation du contrôle, au moins dans un grand nombre de cas. Nous illustrons ce mécanisme en définissant une variante du buffer programmé selon cette abstraction.

On souhaite définir un buffer qui, en plus de `put` et de `get`, exporte deux nouvelles routines :

- `get_last` : retourne le dernier élément ajouté dans le buffer ;
- `nb_element` : fonction qui retourne le nombre d'éléments du buffer.

La classe `ABST_BUFFER2` (figure 7.11) programme cette spécification. Afin de définir cette classe processus, on hérite du buffer `ABST_BUFFER` (figure 7.10). L'ancienne routine de synchronisation est renommée en `old_synchronization`. Ainsi, elle pourra être réemployée lors de la redéfinition de la routine de synchronisation ; on réalise de cette manière une définition incrémentale du contrôle. La nouvelle routine

```
class ABST_BUFFER2 [T]  -- Nouveau buffer, definition incrementale
                        -- de la synchronisation
export
   put, get, get_last, nb_element
inherit
   ABST_BUFFER
      rename synchronization as old_synchronisation;
      redefine synchronization
feature
   synchronization is   -- La nouvelle synchronisation du buffer
      do
         old_synchronisation;
         associate (& get_last, & empty);
      end;
end -- ABST_BUFFER2
```

FIG. 7.11: ABST_BUFFER2 *: réutilisation de la synchronisation.*

exportée `get_last` se voit attacher la condition de blocage `empty`. En revanche, aucune contrainte de synchronisation n'est spécifiée pour la routine `nb_element` : elle peut être appelée dans toutes les configurations du buffer — sa condition de blocage a toujours la valeur *faux*.

### 7.3.4 Résumé – perspectives

En conclusion de cette section, notons que le mécanisme d'abstraction présenté va permettre le développement de bibliothèques d'abstractions pour la synchronisation des processus ; la figure 7.12 illustre ces possibilités. Le programmeur d'applications parallèles pourra sélectionner celle qui convient le mieux au problème à programmer — comme on peut déjà le faire pour les structures de données. Dans le cas où l'abstraction dont il a besoin ne s'y trouverait pas, il pourra en définir une nouvelle. Si cela s'avère impossible, il lui restera la ressource de revenir à une programmation explicite du contrôle en héritant directement de la classe `PROCESS`.

## 7.4 Démarche de conception

Cette section propose une démarche de conception : nous définissons une méthode pragmatique constituée de quatre phases successives (figure 7.13). D'ambition limitée, son but est de guider le programmeur lors de la structuration et de la programmation d'une application parallèle.

La méthode est principalement fondée sur la réutilisation de programmes. La structuration d'un système en différentes activités parallèles se fonde sur la structuration objet préexistant dans l'application séquentielle ; on décide de transformer un certain nombre d'objets en processus. Contrairement à d'autres méthodes

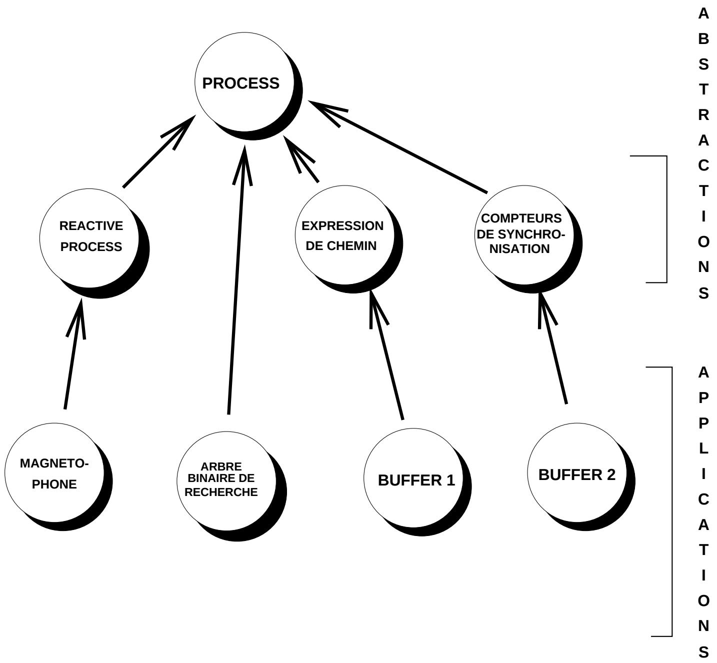



FIG. 7.12: *Exemple de bibliothèque d'abstractions.*

[Ward, 1986 ; Gommaa, 1989 ; Heitz et Labreuille, 1988] qui décident *a priori* des différents processus, le principe est ici de retarder cette identification.

La méthode propose tout d'abord de structurer et de programmer le système de façon totalement séquentielle, sans se soucier du parallélisme. La deuxième phase structure l'application en processus à partir du système séquentiel obtenu précédemment. La programmation des processus est réalisée pendant la troisième phase. Enfin, une dernière phase permet de modifier le système afin de l'adapter aux spécifications.

### Phase 1 : Structuration et programmation séquentielle

Cette phase structure l'application en un ensemble d'objets. La programmation est ensuite réalisée ; on obtient donc un ensemble de classes. On réalise ici une programmation séquentielle classique : le domaine de l'application est modélisé par un ensemble de classes tout à fait classiques.

Différentes méthodes (voir *chapitre 4*) proposent des guides pour réaliser cette conception [Booch, 1986 ; Booch, 1987 ; Booch, 1990 ; Booch, 1994a ; Booch, 1994b ; Booch et Rumbaugh, 1995 ; Booch *et al.*, 1996 ; Halbert et O'Brien, 1987 ; Meyer, 1988 ; Meyer, 1990 ; Gindre et Sada, 1989]. Nous ne détaillons pas cette phase qui ne correspond pas à notre propos. L'important est que l'on dispose, au terme de cette étape, d'un système séquentiel complet et exécutable. Il est donc possible de tester les algorithmes et leur implémentation. En séparant ces tests de la programmation parallèle, la complexité de la mise au point est considérablement réduite.

1. Structuration et programmation séquentielle
2. Identification des processus :
   (1) Activités initiales
   (2) Objets partagés
3. Programmation des processus :
   (1) Définition des classes actives
   (2) Définition des interfaces
   (3) Programmation du corps des processus
   (4) Utilisation des processus
4. Adaptations et modifications :
   (1) Raffinement de la synchronisation
   (2) Redéfinition de routines
   (3) Raffinement de la topologie
   (4) Ajout de nouveaux processus

FIG. 7.13: *Résumé de la méthode.*

**Phase 2 : Identification des processus**

À partir du système séquentiel programmé en phase 1, on identifie les processus qui constitueront donc un sous-ensemble des objets déjà modélisés. Ceci s'avère réalisable dans la mesure où la structuration en objets est d'un grain plus fin que la structuration en processus. Cette identification correspond à la structuration du système initial en un ensemble de sous-systèmes, chacun étant le siège d'une des activités parallèles du futur système. Elle est réalisée en plusieurs étapes :

*(1) Activités initiales*

Nous identifions ici les sources d'activité initiale, c'est-à-dire : tous les objets du système où une activité doit ou peut apparaître.

Les critères varient selon le domaine d'application. Dans des systèmes temps-réel, on recherchera par exemple les objets de contrôle qui assurent continuellement leur activité, les objets périodiques qui assurent leur fonction d'une façon régulière, les objets réagissant à des entrées/sorties asynchrones, etc. Dans la parallélisation d'algorithmes, on recherchera les objets dont les fonctions peuvent être réalisées en parallèle.

*(2) Objets partagés*

L'étape 1 fait apparaître les objets partagés, réferencés au minimum par deux processus. Le modèle de programmation (section 7.4) interdit de tels objets. Nous devons considérer le cas de chacun d'eux. Une première solution :

1. ne rien faire. Les paramètres des routines des processus sont automatiquement transmis par copie, ce qui ne crée jamais d'objet partagé. Cependant, cette solution ne s'avère pas toujours réalisable, pour deux raisons :

```
class P_A   -- Une classe processus
export  ... -- Nouvelle interface
inherit
   A;   -- Classe d'origine
   PROCESS  redefine Live;
feature
   Live is
        -- Activite propre et synchronisation
      do
         ...
      end;
   ...   -- Nouvelles routines
end -- P_A
```

FIG. 7.14: *Programmation des processus.*

- Le passage par copie peut entraîner une modification de la sémantique globale du système. Il est donc toujours nécessaire de vérifier ce point.
- La copie de l'objet peut survenir de facon répétée. En fonction de la taille de l'objet, cela pourra être très coûteux en temps et en espace.

Pour chaque objet dont la copie ne constitue pas une solution satisfaisante, la seconde possibilité consiste à :

2. le transformer en processus. Devenant un processus, l'objet est à nouveau transmis par référence ; les problèmes énoncés ci-dessus disparaissent.

Cette étape, qui peut donc nous amener à identifier de nouveaux processus, est appliquée jusqu'à ce qu'elle n'introduise plus de nouveaux objets partagés.

**Phase 3 : Programmation des processus**

Cette phase s'attache à la programmation des processus identifiés à l'étape précédente. Pour les objets demeurant passifs, aucune programmation n'est nécessaire. La programmation des processus correspond à une transformation des classes séquentielles en classes parallèles. Plusieurs étapes distinctes se détachent :

*(1) Définition des classes actives*

Pour chaque classe `A` identifiée comme une source d'activité, définir une nouvelle classe (`P_A`) qui hérite de `A` et de la classe `PROCESS` ou, dans le cas où l'on choisit une programmation moins explicite, d'une abstraction particulière. Dans ce dernier cas, le choix de l'abstraction dépend des propriétés qu'elle met en valeur et de leur adéquation pour la description de l'application. La figure 7.14 illustre cette définition, ainsi que les étapes (2) et (3) présentées ci-dessous.

*(2) Définition des interfaces*

Les routines exportées des processus ne sont pas forcément les mêmes que celles de leur classe d'origine. Les classes définies dans la phase 1 se veulent générales et réutilisables ; elles comporteront normalement toutes les routines caractéristiques des entités qu'elles modélisent. En général, l'interface des processus sera plus réduite. Ce n'est que très rarement qu'il faut définir de nouvelles routines.

*(3) Programmation du corps des processus*

L'activité du processus reste à définir. La routine `Live` doit être programmée. Celle héritée de la classe `PROCESS` définit une synchronisation FIFO. Dans le cas où l'activité du processus consiste à servir les requêtes dans l'ordre où elles arrivent, sans réaliser aucune autre action, cette synchronisation convient. Dans les autres cas, il faudra redéfinir cette routine pour programmer explicitement le corps du processus. En général, les deux éléments suivants doivent être exprimés lors de la programmation du processus :

- le service des routines exportées, ainsi que la synchronisation de ces services ;
- l'activité propre du processus : les actions qu'il réalise de façon indépendante.

Le cas extrême est celui où l'un de ces deux éléments est inexistant. Cependant, dans la grande majorité des cas, l'activité d'un processus est une combinaison de ces deux types d'actions.

*(4) Utilisation des processus*

Les processus ayant été définis, il faut maintenant les utiliser. Tout d'abord, il est nécessaire de créer des instances de ces processus. Pour ce faire, des entités des classes processus sont déclarées et une création est réalisée sur ces entités. Il faut ensuite que notre système utilise ces processus à la place des anciens objets. Le polymorphisme est utilisé : nous affectons les processus aux variables du type des classes d'origine (dans notre exemple, un processus de type `P_A` est affecté à une variable de type `A`). Ainsi, la liaison dynamique va jouer son rôle : chaque fois qu'une variable de type `A` apparaît, le processus est maintenant utilisé. La figure 7.15 présente ces deux types d'opérations.

Cette programmation peut être nécessaire dans différents endroits du système. Une localisation systématique peut être obtenue en recherchant toutes les instructions où l'on crée des instances des classes transformées en processus. Cependant, ces actions sont souvent localisées dans la classe racine d'une application. C'est en effet à cet endroit que la topologie principale du système est mise en place.

La phase de programmation des processus est maintenant terminée. À ce stade, une première version du système parallèle est entièrement programmée ; son comportement parallèle peut donc être testé.

**Phase 4 : Adaptations et modifications**

Si le système fonctionne, cela n'implique pas qu'il réalise ses spécifications en matière de parallélisme et de distribution. Cette dernière phase propose quelques

```
a: A; -- Variables identifiees comme
b: B; -- devant etre des processus
...
set_processes is  -- Mise en place des processus
   local -- Declarations des processus
            p_a: P_A;  p_b: P_B;
   do
      p_a.Create; -- Creation des processus
      p_b.Create;
      a := p_a;   -- Affectations polymorphes
      b := p_b;
      ... appels des routines sequentieles avec
      ... les objets actifs p_a et p_b
   end;
```

FIG. 7.15: *Utilisation des processus.*

transformations permettant de modifier son fonctionnement, son comportement parallèle et son efficacité.

*(1) Raffinement de la synchronisation*

L'étape 3.3 nous a permis de définir la synchronisation des processus. Celle-ci représente un élément très critique d'un système parallèle — tout particulièrement pour ceux présentant un caractère temps-réel. Il sera souvent nécessaire de la modifier afin d'obtenir le comportement souhaité. Parmi ces transformations, nous pouvons noter :

- changement dans la précédence des services : les requêtes portant sur une routine donnée deviennent plus ou moins prioritaires ;

- changement dans la répartition *service/ activité propre* : par exemple, des requêtes pourront être laissées en attente au profit de la fonction de contrôle du processus.

Ces modifications pourront être réalisées par la définition d'une nouvelle classe : (i) héritant du processus défini précédemment, (ii) redéfinissant la routine `Live`.

*(2) Redéfinition de routines*

Si le modèle de programmation parallèle rend possible la réutilisation de routines séquentielles dans un cadre parallèle, il sera cependant quelquefois nécessaire d'en redéfinir certaines. Ce seront celles qui reflètent un comportement parallèle du système. Un exemple d'une telle transformation est présenté dans [Caromel, 1993].

*(3) Raffinement de la topologie*

Dans un système parallèle, l'information transite d'un processus à l'autre par le biais des communications. Si un processus `p1` a besoin d'une information `i` détenue par un processus `p2`, il existe deux cas de figure :

1. `p1` envoie une requête à `p2` pour lui demander `i`,
2. `p1` reçoit de façon asynchrone l'information `i`, grâce à une routine exportée sur laquelle `p2` envoie une requête.

Cela correspond à deux topologies différentes : dans le cas 1, `p1` référence `p2` ; dans le cas 2, `p2` référence `p1`.

Une modification très fréquente des systèmes parallèles consiste à passer d'une situation à l'autre. Dans le cadre d'une optimisation, on ira fréquemment d'une situation 1 vers une situation 2. En effet, l'idée consiste à minimiser les communications inter-processus (IPC) en ne transférant `i` que si sa valeur a changé. Le processus qui détient `i` envoie la nouvelle valeur par une requête.

De telles transformations impliquent les modifications suivantes :

- Etablissement de la nouvelle topologie.
- Modification de la programmation de `p1` et `p2`. L'un ne sert plus une routine, mais envoie une requête ; l'autre n'envoie plus une requête, mais sert une routine.
- Définition d'une routine permettant le transfert de `i` (si elle n'existe pas déjà).

*(4) Ajout de nouveaux processus*

Une dernière transformation consiste à ajouter de nouveaux processus au système. La raison pourra être le gain d'efficacité obtenu par le placement de ce nouveau processus sur une autre machine, ou encore la recherche d'une meilleure structuration des différentes activités.

Dès que l'on identifie un nouveau processus, il est nécessaire de reprendre la phase 2 à partir de l'étape 2.2 (objets partagés). Dans un second temps, la phase de programmation (3) est appliquée.

La présentation de la méthode est terminée ; la figure 7.13 résume ses différentes phases et étapes. Cette méthode, d'ambition limitée, n'est qu'une première étape vers une construction plus systématique des applications parallèles.

## 7.5 Réactivité

Nous illustrons dans cette section les possibilités d'évolution du modèle d'objet actif par l'introduction d'une bibliothèque EIFFEL// spécialisée définissant ici un modèle d'*objets réactifs*. Comme pour le modèle de base, les idées développées ici sont indépendantes du langage EIFFEL et peuvent être appliquées à d'autres langages à objets.

La programmation réactive vise à la mise en œuvre de systèmes justement dits « réactifs » dont le comportement ne peut pas être vu comme une fonction de ses entrées à un moment donné. En effet, contrairement à ce dernier type de programme, appelé « transformationnel », un programme réactif définit les réactions à des stimuli du système modélisé. Ces derniers surviennent à des instants non prévisibles et tout au long de l'exécution du programme. Un système réactif doit enfin répondre à ces stimuli au rythme que lui impose l'environnement. La programmation dirigée par les événements, par exemple dans le cadre de la programmation d'interfaces graphiques, constitue un bon domaine d'application pour la programmation réactive.

Nous allons introduire dans le modèle à objets actifs la possibilité d'interrompre provisoirement ou définitivement l'exécution d'un traitement en cours (une méthode) afin de permettre à l'objet de réagir sans délai et d'évoluer au rythme des stimuli reçus. Cette interruption sera produite par un client du processus ; ce dernier peut émettre trois commandes : `suspend` pour interrompre l'exécution de l'objet, `resume` pour le faire redémarrer ou `abort` pour abandonner définitivement le traitement précédemment entrepris. Le processus, object actif, devient donc *réactif* dans le cas de ces sollicitations. Dans le cadre d'un tel processus, l'utilisation peut schématiquement être la suivante :

```
my_process.foo(...);
...
my_process.suspend;
...
my_process.resume;
...
my_process.abort;
...
```

Cette proposition ainsi que d'autres cadres réactifs ont été plus largement présentés dans [Caromel *et al.*, 1993 ; Caromel et Roudier, 1996 ; Roudier, 1996].

### 7.5.1 Programmation

La programmation d'un processus réactif (classe `REACTIVE_PROCESS`, figure 7.17) illustre une nouvelle possibilité de modification du comportement standard d'un processus dans le cadre d'une abstraction. Cette modification n'y est pas introduite au niveau du `Live` : c'est la sémantique de la communication d'EIFFEL// et plus particulièrement sa phase d'interruption (voir section 7.1.6) qui est exploitée et permet d'introduire la notion de réactivité. Les actions effectuées pendant la phase de rendez-vous de la communication sont décrites dans la routine `receive_request` : cette méthode correspond normalement à la réception d'une requête et à son stockage dans l'objet. Cette routine fait partie du méta-niveau du processus : elle est partie intégrante du langage et n'est pas figée mais peut au contraire être modifiée par l'utilisateur du langage.

La sémantique de la communication est ainsi modifiée (figure 7.16) : la phase de rendez-vous qui permet normalement d'ajouter des requêtes à la file d'attente est

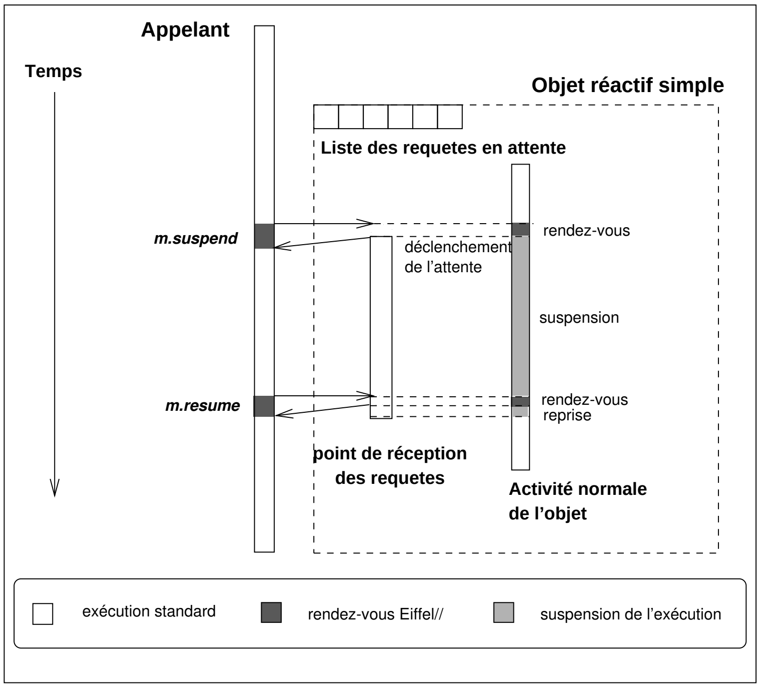



FIG. 7.16: *Principe de la suspension réactive .*

reprogrammée pour permettre, par exemple à l'arrivée d'une requête `suspend`, de se mettre en attente jusqu'à l'arrivée d'une requête `resume` ou `abort`.

Comme précédemment, la classe `REACTIVE_PROCESS` est définie comme un processus particulier. Quand on souhaitera programmer une application parallèle suivant ce modèle de réactivité, on héritera de cette classe ou d'un de ses descendants au lieu de `PROCESS`.

Il est ainsi possible de transformer un objet quelconque en objet réactif. Ceci permet par exemple d'interrompre l'exécution d'une méthode d'un objet que l'on ne veut plus poursuivre. Une telle exécution, une fois démarrée, est d'habitude menée à son terme quoi qu'il puisse se passer dans l'environnement du processus ; pour des raisons diverses (performances, sûreté), certaines modifications du système peuvent nécessiter l'arrêt du traitement en cours.

### 7.5.2 Exemple : un magnétophone réactif

Prenons l'exemple d'un magnétophone simple[2]. Ce magnétophone comporte plusieurs boutons (`start` pour écouter la bande, `stop` pour arrêter toute opération, `backward` pour revenir en arrière, `forward` pour aller en avant), un compteur, et un

2. Cet exemple a été utilisé comme cas d'étude commun par le groupe de travail sur les objets réactifs de Nice-Sophia Antipolis.

```
class REACTIVE_PROCESS  -- processus reactif simple
 export
   suspend, resume, abort
 inherit
   PROCESS redefine receive_request,
           rename receive_request as old_receive_request
 feature
   suspend is do end;  -- nouvelles commandes reactives
     ...
   receive_request is  -- traitement d'une requete qui arrive
     do
       if new_request=suspend then
          from  new_request.serve;
                wait_request;
          until new_request.feat=&abort or ...=&resume
          loop
                if <requete non reactive> then
                   request.add_request(new_request);
          end;
       end; -- if
          ... traitement abort et suspend ...
     end;  -- receive_request
end;  -- class REACTIVE_PROCESS
```

FIG. 7.17: REACTIVE_PROCESS *: processus réactif simple.*

bouton `reset` pour remettre le compteur à zéro. Il permet d'écouter une cassette (la bande est simulée par un fichier audio) sur le haut-parleur d'une station de travail.

Nous montrons comment une programmation réactive de ce système permet de le décrire de manière incrémentale et structurée.

### Conception séquentielle

Notre première approche de la programmation de ce magnétophone est *séquentielle*. Ceci ne signifie pas que la conception en soit contrainte et restreinte par opposition à une conception « parallèle », mais simplement que nous programmons l'application suivant le paradigme normal du langage de base (*langage à objets impératif*), sans nous préoccuper de tout problème de parallélisme, concurrence ou réactivité. Au contraire, nous nous concentrons sur l'encapsulation de fonctionnalités transformationnelles du magnétophone. Celles-ci correspondent d'ailleurs plutôt à une représentation modulaire des spécifications les plus physiques du système, sans prise en compte d'une interaction forte avec l'utilisateur.

Nous structurons cette version comme suit (figure 7.18): l'interface avec l'utilisateur s'appuie sur les classes SPEAKER et TAPE_DISPLAY pour les sorties, et la classe TAPE_COMMAND pour les entrées. Cette dernière classe traduit les événements en provenance de l'interface graphique en appels de routines pour le magnétophone.

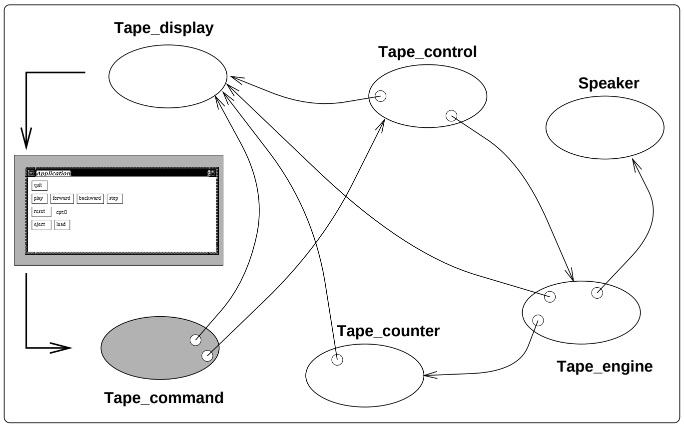



FIG. 7.18: *Magnétophone séquentiel à l'exécution.*

Le cœur du magnétophone est défini dans la classe TAPE_ENGINE ; celle-ci décrit et modélise les contraintes physiques, comme le comportement de la commande `play`. La programmation de cette routine `play` est la suivante : elle lit la bande et la reproduit sur le haut-parleur *jusqu'à sa fin*, ce qui provoque l'arrêt de la lecture :

```
class TAPE_ENGINE
    ...
  play is
   -- lecture (jusqu'a la fin de la bande)
  do
    from
    until end_of_tape
    loop
      one_play; -- lecture d'un fragment de bande
    end;
    tape_display.set_stop;
  end; -- play
```

Notons que nous n'essayons pas d'interrompre le fonctionnement de la routine en attendant des événements de nature réactive : notre approche est ici purement séquentielle et non réactive.

Finalement, la classe TAPE_CONTROL représente le contrôle logique du magnétophone : il décrit les commandes disponibles avec leurs contraintes spécifiques d'utilisation, l'ordonnancement logique des opérations, etc. On ne peut pas, par exemple, appuyer sur le bouton `play` et déclencher la lecture d'une cassette avant d'en avoir effectivement inséré une dans le magnétophone. Dans cet exemple, les routines de la classe TAPE_CONTROL correspondent aux boutons (`start`, `stop`, etc.) qui peuvent être actionnés par le biais de l'interface graphique.

Cette version est un système complet et opérationnel qui peut être testé ; la seule restriction est que, n'étant pas réactif (seulement interactif, il interagit avec les actions sur les boutons), il n'est pas conforme à la spécification complète du magnétophone : les opérations (lecture, avance rapide, etc.) ne sont pas sensibles à une éventuelle demande d'arrêt ou à toute autre commande réactive. Cependant les spécifications fonctionnelles de chacune de ces opérations sont réalisées. Il serait tout à fait possible d'écrire une version séquentielle suivant la spécification exacte, mais au prix d'une grande complication du flot de contrôle (entrelacement d'actions fonctionnelles et de tests d'événements) et de la perte de l'encapsulation de ces opérations ; c'est ce que permet d'éviter la technique de programmation réactive que nous proposons.

### Version réactive

La version réactive (figure 7.19) est obtenue tout d'abord en transformant la classe séquentielle `TAPE_ENGINE` en un processus réactif (`R_TAPE_ENGINE`) qui hérite simplement de `TAPE_ENGINE` et de `REACTIVE_PROCESS` :

```
class R_TAPE_ENGINE
  export repeat TAPE_ENGINE, repeat REACTIVE_PROCESS
  inherit TAPE_ENGINE, REACTIVE_PROCESS
  feature
    -- Aucune redefinition n'est necessaire
end;  -- class R_TAPE_ENGINE
```

Cette classe fonctionne maintenant suivant les spécifications d'un magnétophone réactif. Par exemple, la lecture de la bande peut être stoppée à tout moment grâce à la réactivité introduite par le simple fait d'hériter de la classe `REACTIVE_PROCESS`.

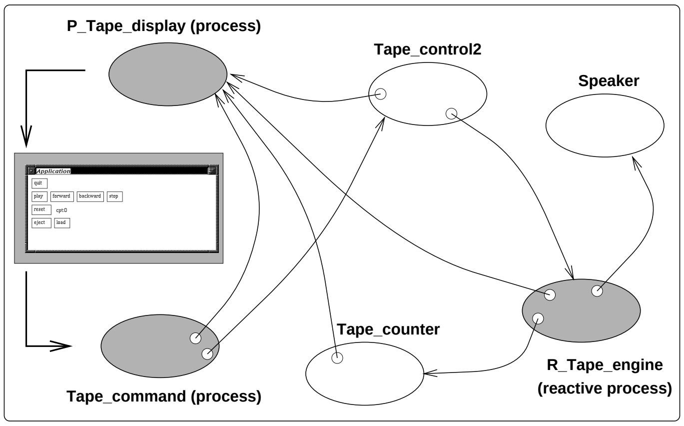



FIG. 7.19: *Magnétophone réactif à l'exécution.*

Sur la figure 7.18 représentant la version séquentielle, on peut voir que l'afficheur (objet `Tape_display`) est accessible par les objets `Tape_command` et `Tape_engine`. Ayant identifié l'objet `Tape_engine` comme un processus de type `R_TAPE_ENGINE`, l'objet `Tape_display` est donc potentiellement accessible par deux processus : il est devenu partagé. Selon la méthode présentée à la section 4, nous devons donc le transformer lui aussi en objet actif. Sa programmation est immédiate :

```
class P_TAPE_DISPLAY
     export repeat TAPE_DISPLAY
     inherit
       TAPE_DISPLAY; PROCESS
     feature
        -- Aucune redefinition
    end;  -- class P_TAPE_DISPLAY
```

Il nous faut maintenant dériver un nouveau contrôleur à partir de TAPE_CONTROL : la classe TAPE_CONTROL2. Les contrôles séquentiels que nous avons définis précédemment constituent un sous-ensemble de la spécification globale du fonctionnement du magnétophone. En plus de ces contrôles qui sont réutilisés tels quels, le nouveau contrôleur va définir le comportement réactif du moteur. La nouvelle classe R_TAPE_ENGINE qui le décrit hérite du moteur séquentiel et de la classe REACTIVE_PROCESS, ce qui permet à ses méthodes de devenir interruptibles. Par exemple, la méthode stop de la classe TAPE_CONTROL était utilisée dans la version séquentielle, mais à l'exclusion de toute possibilité réactive. Elle y assurait la mise à jour de l'affichage lors de l'appui sur le bouton stop, mais n'avait pas de contrôle sur l'exécution des autres méthodes. On peut maintenant la redéfinir comme suit pour obtenir un arrêt du magnétophone (pendant la lecture d'une bande, par exemple) :

```
class TAPE_CONTROL2
   export repeat TAPE_CONTROL
   inherit TAPE_CONTROL rename stop as old_stop,
                               redefine stop
  feature
    stop is
     do
       tape_engine.abort;
       old_stop;
     end;  -- stop
       ...
```

Quelques adaptations sont nécessaires pour assurer un fonctionnement cohérent du système : les routines de bas niveau doivent notamment être rendues ininterruptibles. L'application remplit maintenant le cahier des charges fixé quant à son comportement et pour un nombre de transformations très limité. La figure 7.19 présente le système à l'exécution.

Cet exemple peut être étendu. Supposons qu'un bouton pause doive être pris en compte, arrêtant le défilement de la bande (par exemple pendant l'exécution d'une méthode play) : l'utilisation de la méthode suspend permettra de réaliser ce comportement.

```
prior_activity: like activity;
   ...
pause is
 do
   if activity/=suspended
     then tape_engine.suspend;
          prior_activity:=activity;
          activity:=suspended;
```

```
      else tape_engine.resume;
           activity:=prior_activity;
    end; -- if
  end;  -- pause
```

L'application simple que nous venons de présenter illustre l'intérêt de la programmation réactive à objets : elle permet d'employer les mécanismes objets classiques pour définir le comportement d'un système de manière incrémentale, à plusieurs niveaux de description. Ce mode de description ouvre des perspectives de réutilisation et de redéfinition de composants séquentiels ou parallèles existants.

D'autres modèles d'objets actifs peuvent être définis en étendent le modèle proposé. Ce type d'extension tire parti du polymorphisme et des points offrant un accès réflexif aux mécanimes du langage comme le point de réception des requêtes (`receive_request`).

## 7.6 Conclusion

Nous avons présenté dans cet article les mécanismes de base du langage EIFFEL// ainsi que des constructions de plus haut niveau pour l'expression de synchronisations ou d'interactions, notamment réactives, plus proches des problèmes et des préoccupations du programmeur. Ce langage a été expérimenté sur des stations Sun connectées par un réseau Ethernet ; l'implantation utilise les processus Unix pour générer les processus du niveau langage.

Nous travaillons actuellement sur le langage C++// [Caromel *et al.*, 1996]. Il s'agit d'une extension du langage C++ que nous construisons suivant les mêmes idées : son modèle de programmation étend et généralise celui d'EIFFEL//. C++// introduit des possibilités supplémentaires de placement d'autant plus importantes que la nouvelle implantation va employer des processus légers (threads). Nous développons en plus des capacités réflexives dans C++//. Nous y avons notamment défini quatre points réflexifs symétriques liés à la communication au lieu d'un seul (`receive_request`). Du point de vue de son implantation, également, C++// s'appuie maintenant sur le modèle réflexif Europa C++ ; ce dernier vise à offrir un cadre commun pour l'implantation de modèles variés de programmation parallèle et distribuée. La définition et l'implantation de C++// s'inscrit dans les objectifs du projet SLOOP [Furmento et Baude, 1995 ; Baude *et al.*, 1996b ; Baude *et al.*, 1996a] : l'implantation s'appuie notamment sur une couche de communication objet développée dans le projet, la bibliothèque SCHOONER [Furmento et Baude, 1995], qui offre une base portable sur de nombreuses machines parallèles ; des modèles de communication spécialisés devraient pouvoir être intégrés et être accessibles aux programmeurs. La première cible que nous visons avec ces outils, la simulation distribuée, devrait d'ailleurs leur servir de banc d'essai.

Nous poursuivons également un effort de définition de la sémantique formelle d'EIFFEL// [Attali *et al.*, 1993 ; Attali *et al.*, 1995 ; Attali *et al.*, 1996], ceci dans le but de formaliser, de prouver et d'automatiser les transformations de programmes réalisées jusqu'à présent de façon manuelle.

# 8

# Les langages à prototypes

LES LANGAGES À PROTOTYPES proposent une forme de programmation par objets s'appuyant sur la représentation d'objets concrets plutôt que sur celle de concepts, ou de façon plus pragmatique, une forme de programmation par objets sans classes. Ce chapitre décrit ces langages, les raisons qui ont conduit à leur émergence, les possibilités nouvelles qu'ils apportent, mais aussi les problèmes qu'ils posent. Nous répondons en premier lieu aux questions suivantes. Qu'est-ce qu'un prototype? Qu'est-ce qu'un objet concret? Pourquoi chercher à se passer des classes? Quelle est l'architecture primitive d'un langage à prototypes et quelle est la genèse de ces langages? Nous caractérisons plus précisément les deux mécanismes de création d'objets : le clonage, et la description différentielle. Cette caractérisation nous permet de différencier l'héritage dans les hiérarchies d'objets (la délégation) de l'héritage via des hiérarchies de classes (qui est l'héritage classique des langages à objets).

Elle nous permet également de présenter les spécificités ainsi que les problèmes que pose la programmation par prototypes. Au travers de cette présentation critique, nous posons finalement la question de l'intérêt et de l'avenir de cette forme de programmation par objets.

## 8.1 Introduction

Les langages à objets les plus utilisés aujourd'hui (en particulier dans le monde industriel) sont issus de SIMULA (1967) et de SMALLTALK (1972). Ce sont des langages au sein desquels la classe, modélisation intensionnelle d'un concept, est l'unité fondamentale autour de laquelle s'organise la représentation de connaissances et se structurent les programmes (*chapitres 1, 3 et 2*).

Parmi les autres familles de langages de programmation par objets effectivement utilisées ou largement étudiées dans des travaux de recherche figure la famille des langages dits à prototypes dont nous traitons dans ce chapitre. Un prototype est un *représentant typique* d'une famille ou d'une catégorie d'objets [Cohen et Murphy, 1984]. Les langages à prototypes, apparus au milieu des années 80, proposent une approche de la programmation par objets reposant sur la notion de prototype plutôt que sur celle de classe. Ils ont été inspirés par les premiers langages de *frames* utilisés en représentation de connaissances (*chapitre 10*) et par certains langages d'acteurs, issus des travaux de C. Hewitt, utilisés en programmation distribuée (*chapitre 6*). Ils ont été étudiés et développés, en réaction à un certain nombre de limitations propres au modèle à classes, avec les objectifs suivants:

– permettre une description simple des objets ne nécessitant pas la description préalable de modèles abstrait,

– offrir un modèle de programmation plus simple que celui des classes, qui jouent de trop nombreux rôles,

– enfin, offrir de nouvelles possibilités de représentation de connaissances.

Les langages à prototypes sont issus de ces objectifs ainsi que du constat d'un certain nombre de limitations du modèle à classes [1] [Borning, 1986], avec l'espoir d'obtenir une plus grande puissance d'expression dans des langages par ailleurs plus simples. Depuis le milieu des années 1980, de nombreux langages ont vu le jour : SELF [Ungar et Smith, 1987 ; Agesen *et al.*, 1993 ; Agesen *et al.*, 1995 ; Chambers *et al.*, 1991 ; Smith et Ungar, 1995], KEVO [Taivalsaari, 1991 ; Taivalsaari, 1993], NAS [Codani, 1988], EXEMPLARS [LaLonde *et al.*, 1986 ; Lalonde, 1989], AGORA [Steyaert, 1994], GARNET [Myers *et al.*, 1990 ; Myers *et al.*, 1992], MOOSTRAP [Mulet et Cointe, 1993 ; Mulet, 1995], CECIL [Chambers, 1993], OMEGA [Blaschek, 1994], NEWTON-SCRIPT [Smith, 1994] . D'autres langages, tels OBJECT-LISP [Apple, 1989] ou YAFOOL [Ducournau, 1991] ne se réclamant pas de l'approche par prototypes offrent néanmoins des mécanismes proches.

La caractérisation très générale et informelle des langages à prototypes est relativement aisée : ce sont des langages dans lesquels on trouve en principe une seule sorte d'objets dotés d'attributs [2] et de méthodes, trois primitives de création d'objets : création *ex nihilo*, *clonage* et *extension* (ou *description différentielle*), un mécanisme de calcul, l'envoi de message, intégrant un mécanisme de *délégation*. Ceci étant posé, leur caractérisation, leur utilisation et leur compréhension précise posent en fait un certain nombre de problèmes.

- Il existe diverses interprétations de ce qu'est un prototype, objet concret ou représentant moyen d'un concept, qui peuvent conduire à des langages assez différents [Malenfant, 1995].
- La sémantique des mécanismes de base (clonage, copie différentielle, délégation) n'est pas unifiée et autorise différentes interprétations [Dony *et al.*, 1992 ; Malenfant, 1995 ; Bardou et Dony, 1996 ; Bardou *et al.*, 1996 ; Malenfant, 1996].
- La description différentielle rend les objets interdépendants, ce qui pose de nouveaux problèmes et autorise diverses interprétations quant au statut des objets [Dony *et al.*, 1992 ; Malenfant, 1996 ; Bardou et Dony, 1996 ; Bardou *et al.*, 1996 ; Bardou, 1998].

1. Nous serons fréquemment amenés à comparer le monde des prototypes et celui des classes, nous supposons ce dernier connu du lecteur, qui pourra consulter les *chapitres 1, 3 et 2* à ce sujet.

2. Nous utilisons ce terme pour désigner une caractéristique non comportementale d'un objet ou d'un *frame*, nous aurions pu utiliser les équivalents que sont « champ » ou « slot ».

- En même temps que les classes, a été supprimée par exemple, la possibilité d'exprimer que deux concepts partagent certaines caractéristiques. Cette seconde possibilité est si importante en terme d'organisation des programmes que de nombreux langages à prototypes ont cherché à la réintroduire, ce qui a été fait de façon plus ou moins appropriée. Cette réinsertion de formes d'abstraction dans le modèle a remis en cause certains postulats initiaux et a brouillé les frontières entre langages à prototypes et langages à classes [Malenfant, 1996].

Nous nous proposons de décrire ces langages, de juger des possibilités qu'ils offrent, d'étudier dans quelle mesure ils satisfont les objectifs que leurs concepteurs s'étaient fixés et à quel prix. Nous nous demandons si ces langages sont viables (peut-on se passer de la représentation des concepts), dans l'affirmative lesquels utiliser et, dans la négative, si certaines des idées qu'ils ont introduites peuvent être appliquées dans d'autres contextes? Le paragraphe 8.2 rappelle ce qu'est la notion de prototype en science cognitive. Le paragraphe 8.3 présente les premières utilisations de cette notion en représentation de connaissances par objets ainsi qu'en programmation distribuée. Nous y décrivons les primitives de clonage et de description différentielle. Le paragraphe 8.4 expose les motivations qui ont conduit les chercheurs à concevoir des langages à objets sans classes ; il montre l'intérêt potentiel de la programmation par prototypes. Le paragraphe 8.5 décrit les premières propositions de langages sans classes. Le paragraphe 8.6 fait le point sur les concepts et les mécanismes de base de la programmation par prototypes. Le paragraphe 8.7 propose une caractérisation plus fine de ces concepts et de ces mécanismes. Cette caractérisation permet de différencier les hiérarchies d'objets des hiérarchies de classes et de mieux comprendre les différentes évolutions des langages à prototypes. Le paragraphe 8.8 décrit les problèmes liés à l'identité des objets et le paragraphe 8.9 les problèmes liés à l'organisation des programmes. En conclusion nous présentons un bilan de l'expérience ainsi les axes de recherche que notre analyse fait apparaître.

## 8.2 Notion de prototype

On trouve en sciences cognitives l'idée de représenter un concept, ou une famille d'entités, par un représentant distingué ainsi que l'idée de copie différentielle. Dans ce contexte, différents modèles de la notion de concept ont été proposés [Smith et Medin, 1981 ; Cohen et Murphy, 1984 ; Kleiber, 1991].

Un de ces modèles est fondé sur la théorie des ensembles : à chaque concept correspond une collection d'entités (extension), et chaque concept admet une définition qui caractérise son « essence » et définit les conditions nécessaires et suffisantes à l'appartenance d'une instance à ce concept (intension), cf. *chapitre 12*. La relation qui lie une instance à un concept et celle qui lie un concept plus spécifique à un concept plus général s'y apparentent respectivement aux relations ensemblistes d'appartenance et d'inclusion. Ce modèle de concepts conduit à une mise en œuvre basée sur les classes.

Un autre modèle (développé en linguistique) permet de ne pas valuer systématiquement toutes les caractéristiques d'une instance. Il y a toujours des conditions nécessaires et suffisantes pour l'appartenance à un concept, mais on s'accorde la possibilité de ne pas savoir : on sait qu'une instance appartient à un concept, qu'elle n'y appartient pas, ou bien on n'en sait rien (*chapitre 10*). La « théorie des prototypes » est une extension de cette approche dans laquelle la relation d'appartenance est une certaine relation de ressemblance plus ambiguë. Dans cette théorie, les concepts ne sont décrits ni en intension ni en extension mais indirectement au travers de prototypes du concept, c'est-à-dire d'exemples. Cette théorie découle du principe selon lequel l'humain se représente mentalement un concept, identifie une famille d'objets et mène des raisonnements sur ses membres en faisant référence, au moins dans un premier temps, à un objet précis, typique de la famille. Ma « 2CV » est, par exemple, un prototype du concept de « voiture », comme « netscape » l'est pour le concept de « navigateur internet ». On trouve aussi dans la théorie des prototypes la notion de description différentielle qui désigne la possibilité de décrire 'un nouveau représentant du concept via l'expression de ses différences par rapport à un représentant existant.

Afin de mieux expliquer comment ces notions ont été utilisées, il nous apparaît nécessaire de distinguer deux sortes de prototypes que nous rencontrerons dans nos langages : le représentant concret et le représentant moyen d'un concept. Il est préalablement nécessaire d'établir une distinction terminologique entre les objets du monde dont nous souhaitons réaliser une description informatique (que nous appellerons le « domaine », cf. *chapitre 10*) et les objets de nos langages. Nous utiliserons le terme « entité » pour désigner les premiers.

– **Représentant concret et instance prototypique.** Le représentant concret, dont ma « 2CV » est un exemple pour le concept « voiture », correspond à une entité concrète. Nous reprenons le terme d' « instance prototypique » pour désigner un représentant concret utilisé comme référence pour décrire ou créer d'autres objets. L'instance prototypique d'un concept est ainsi souvent le premier objet d'une famille.

– **Représentant moyen.** Un représentant moyen représente une entité qui peut être abstraite ou incomplète. Le représentant moyen ne représente aucune entité concrète. Il peut ne posséder que les attributs les plus courants avec les valeurs les plus courantes pour la catégorie d'entités qu'il représente. La « ménagère de moins de 50 ans » est un exemple célèbre de représentant moyen du concept « téléspectateur » ; un objet possédant quatre roues et un moteur est un représentant moyen du concept « voiture ». Ses attributs peuvent contenir des valeurs moyennes, par exemple, la femme française typique a 1,8 enfants.

## 8.3 Utilisations informatiques de la notion de prototype antérieures aux langages à prototypes

### 8.3.1 Les prototypes en représentation des connaissances

Les langages à prototypes existaient avant que l'appellation n'apparaisse. On trouve ainsi les notions de prototype et de copie différentielle dans la théorie des

```
Frame
  nom: "baleine"
  catégorie: mammifère
  milieu: marin
  ennemi: homme
  poids: 10000
  couleur: bleu
```

FIG. 8.1: *Exemple de* Frame

```
Frame
  nom: "Moby-Dick"
  est-un: baleine
  couleur: blanche
  ennemi: Cpt-Haccab
```

FIG. 8.2: *Description différentielle*

*frames* de Minsky [Minsky, 1975] et dans certains systèmes inspirés de cette théorie comme les langages de *frames* tels KRL [Bobrow et Winograd, 1977] ou FRL [Roberts et Goldstein, 1977] (voir *chapitre 10*).

> « *Les* frames *sont un formalisme de représentation créé pour prendre en compte des connaissances qui se décrivent mal ... [dans d'autres formalismes] ... comme la typicalité, les valeurs par défauts, les exceptions, les informations incomplètes ou redondantes. La structure d'un* frame *... doit pouvoir évoluer à tout moment, par modification, adjonction ou modification de propriétés.* » [Masini *et al.*, 1989]

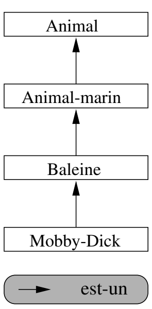
Nous allons donner ici une vision simplifiée à l'extrême de ce que sont les *frames*, sans illustrer leur richesse et leur diversité ; notre but est de montrer en quoi ils utilisent la théorie des prototypes et comment ils ont influencés certains des langages à prototypes utilisés aujourd'hui. Le lecteur se reportera aux articles précédemment cités, au chapitre proposé dans [Masini *et al.*, 1989] et au *chapitre 10* pour plus de précisions.

• **Structure d'un frame.** Un *frame* est un ensemble d'attributs; chaque attribut permet de représenter une des caractéristiques du *frame* et se présente sous la forme d'un couple « nom d'attribut – ensemble de facettes ». La facette la plus courante étant la valeur de l'attribut, nous ne considérerons que celle-ci dans nos exemples. La figure 8.1 propose un exemple de définition d'un *frame* dotée de 4 attributs représentant de façon minimale une baleine.

• **Description différentielle.** La description (ou création) différentielle permet de décrire un nouveau *frame* en exprimant ses différences par rapport à un *frame* existant[3]. Elle met en relation le nouveau *frame* avec celui sur lequel sa description différentielle s'appuie et qui est appelé son prototype ou son *parent*. Cette relation est matérialisée par un lien généralement appelé *est-un*. Nous avons représenté ce lien dans nos exemples par l'intermédiaire d'un attribut supplémentaire[4] également nommé *est-un*. La figure 8.2 montre la définition d'un *frame* représentant `Moby-Dick`

3. « *The object being used as a basis for comparison (which we call the prototype) provides a perspective from which to view the object being described. (...) It is quite possible (and we believe natural) for an object to be represented in a knowledge system only through a set of such comparisons.* » [Bobrow et Winograd, 1977]

4. Ce lien est en premier lieu utilisé par le système et n'est pas nécessairement accessible au programmeur via un attribut. Nous avons donc, pour des raisons de simplicité dans notre exposé, introduit, une forme de réflexivité qui pose le problème de la modification éventuelle de l'attribut *est-un*.




```
Frame
  nom: "animal"
  milieu: terrestre
  nourriture: nourriture
  nbPattes: 4
  poids: 10

Frame
  nom: "animal-marin"
  est-un: animal
  milieu: marin
  nourriture: n-marine
  nbPattes: 0
  nbNageoires: 4

Frame
  nom: "baleine"
  est-un: animal-marin
  nourriture: plancton
  couleur: bleu
  poids: 10000
  ennemi: homme

Frame
  nom: "Moby-Dick"
  est-un: baleine
  couleur: blanc
  ennemi: Cpt-Haccab
```

FIG. 8.3: *Exemple de hiérarchie de* frames.

qui est comme la baleine précédente à ceci près qu'elle est blanche et que son ennemi est défini plus précisément.

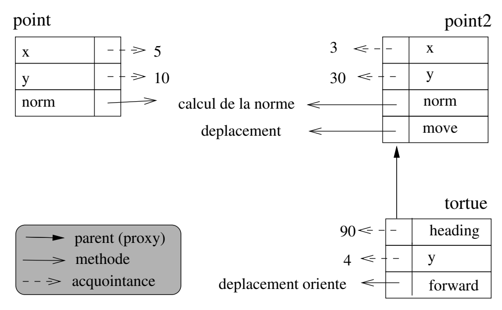
• **Héritage et Hiérarchies de *frames*.** La relation *est-un* est une relation d'ordre définissant des hiérarchie de *frames* [Brachman, 1983]. Un *frame* hérite de son parent un ensemble d'attributs et on trouve dans les systèmes de *frames* des hiérarchies d'héritage, comme celle de la figure 8.3, très similaires aux hiérarchies de classes [5], à ceci près que les nœuds de cette hiérarchie représentent des exemples plutôt que des description de concepts. Au sommet de la hiérarchie se trouvent généralement des représentants moyens de concepts (par exemple `Animal`) et dans le bas de la hiérarchie des représentants concrets (par exemple `Moby-Dick`). On trouve des hiérarchies similaires dans les programmes réalisés avec les langages à prototypes.

### 8.3.2 Langages d'acteurs

L'idée de représenter des entités du monde par des objets sans classes a également été appliquée dans le langage ACT 1 [6] [Lieberman, 1981], bien qu'il ne soit fait mention, dans les articles relatifs à ce langage, ni de la notion de prototype ni de l'utilisation qui a pu en être faite dans les langages de *frames* qui lui sont antérieurs [7], tel que KRL. On trouve cependant dans ACT 1 des idées et des mécanismes assez similaires à ceux évoqués précédemment ainsi qu'une part des caractéristiques essentielles des langages à prototypes actuels.

• **Structure d'un acteur.** Les objets dans ACT 1 sont appelés « acteurs », ils possèdent des attributs (appelés *accointances*) référencés par un nom et possédant une valeur. ACT 1 est un langage de programmation, les acteurs sont donc également dotés de comportements (nous utilisons les termes classiques de « méthode » pour désigner un comportement et celui de « propriété » pour désigner indiff'eremment

5. Notons que la définition d'une sous-classe est également une description différentielle.
6. Voir les *chapitres 6 et 7* pour tout ce qui concerne la programmation parallèle
7. Ceci peut être dû en partie au fait qu'il n'est pas évident d'isoler clairement l'utilisation faite des prototypes dans KRL. D'autre part, l'objectif de ACT 1, mettre en œuvre un outil de programmation parallèle à base d'objets, est notablement éloigné de celui de KRL.




```
(object 'extend 'point
  '((x 5) (y 10))
  '((norm (lambda ()
...))))

(point 'create 'point2
  '((x 3) (y 30))
  '((move (lambda (newX
newY)
      (setq x newX y
newY))))

(point2 'extend 'tortue
  '((heading 0) (y 4))
  '((forward (lambda (...)
...))))
```

FIG. 8.4: *Clonage et extension en* ACT 1.

un attribut ou une méthode). Les méthodes peuvent être invoquées en envoyant des messages aux acteurs [8].

Un acteur, objet sans classe ne pouvant être créé par instanciation, est créé par copie ou extension d'un acteur existant. La figure 8.4 montre un acteur [9] appelé `point` possédant deux attributs `x` et `y` et une méthode `norm` calculant la distance de ce point a l'origine. Trois primitives `create`, `extend` et `c-extend` permettent de créer de nouveaux acteurs [Briot, 1984]. Ce sont ces trois primitives et la mise en œuvre de la copie différentielle qui nous intéressent ici. Nous en discutons au travers des exemples proposés dans [Briot, 1984].

• **Clonage.** Bien que des primitives de copie d'objets aient existé dans les langages à classes (par exemple en SMALLTALK) antérieurement à ACT 1, ce langage a introduit la copie superficielle [10] (primitive `create`), ou *clonage*, comme moyen primitif de création d'objets. La primitive `create` permet ainsi de créer `point2` par copie de `point` (Fig. 8.4), de spécifier de nouvelles valeurs de propriétés, par exemple `x` et `y`, et d'en définir de nouvelles, par exemple la méthode `move`.

• **Extensions.** La création par description différentielle en ACT 1 est conceptuellement similaire à celle des *frames*. Elle s'effectue en envoyant à un acteur existant le message `extend`, qui crée un nouvel acteur, que nous appellerons donc « extension » du premier, lui même appelé en ACT 1 le *mandataire* (*proxy* en anglais) du nouvel acteur ; c'est l'équivalent du prototype ou du parent des *frames*. La figure 8.4 montre la définition de l'acteur nommé `tortue` représentant une tortue *Logo* [11] comme une extension de l'acteur précédent `point2` qui devient son *manda-*

8. Cette vision est simplificatrice mais nous suffit ici ; en fait, les comportements d'un acteur sont regroupés au sein d'un *script* et l'invocation peut faire intervenir un mécanisme de filtrage.

9. Créé par extension d'un acteur pré-défini.

10. A l'inverse de la copie profonde, la copie superficielle ne copie pas les objets composant l'objet copié.

11. C'est à dire représentant un robot se déplaçant dans le plan tout en traçant un trait sur son passage.

*taire*. Une tortue est comme un point mais possède en plus un `cap` et une méthode `forward` lui permettant d'avancer dans la direction définie par son `cap`.

• **Héritage et première forme de délégation**. Le lien reliant une extension à son mandataire est tout à fait similaire au lien *est-un* des *frames*. L'extension peut hériter des propriétés de son parent. L'héritage est mis en œuvre lorsqu'un acteur ne sait pas répondre à un message parce qu'il ne possède pas la propriété demandée, auquel cas le système demande à son mandataire de répondre à sa place. Le mandataire est ainsi habilité à répondre à un message en lieu et place de ses extensions. Ce passage du contrôle au mandataire est appelé « délégation »[12]. Les articles décrivant ACT 1 laissent dans l'ombre un point important de la problématique des langages à prototypes (cf. paragraphe 8.5.1) en ne précisant pas le contexte d'exécution une méthode après qu'il y ait eu une délégation. On trouve quoi qu'il en soit dans ACT 1 une première forme de ce qui deviendra la délégation dans les langages à prototype.

• **Copie-extension.** Une des caractéristiques des hiérarchies d'objet est la dépendance qu'établit le lien de délégation entre un parent et une de ses extensions :

> « *Les accointances (les attributs) et le* script *(l'ensemble des méthodes) de mon mandataire (mon parent) sont aussi les miens.* » [Briot, 1984]

Les propriétés du parent sont partagées par ses extensions. Pour permettre la création différentielle d'un nouvel acteur indépendant, une troisième primitive de ACT 1, nommée `c-extend`, compose un clonage[13] et une extension du clone.

## 8.4 Motivations et intérêts de la programmation par prototypes

On trouve dans les systèmes que nous venons de décrire l'essence de ce qui a été appelé programmation par prototypes. Les études relatives à l'introduction d'objets sans classes dans les langages de programmation par objets ont été réalisées au milieu des années 80. Elles visaient à proposer des alternatives au style usuel de programmation par classes utilisé avec SIMULA, SMALLTALK, C++ ou les FLAVORS. Ces études portaient en premier lieu sur la complexité du monde des classes [Borning, 1986] et les limitations que celles-ci imposent en terme de représentation.

Expérimenter un modèle de programmation par objets s'appuyant sur la théorie des prototypes est apparu comme une possibilité de réduire cette complexité (langages plus simples) et de relâcher les contraintes portant sur les objets. Ce paragraphe illustre les problèmes que posent les classes et montre en quoi les prototypes sont une solution potentielle à ces problèmes.

12. « *Whenever an actor receives a message he cannot answer immediately on the basis of his own local knowledge (...), he delegates the message to another actor, called his proxy* ». [Lieberman, 1981]

13. Notons que ce clonage peut être complexe : si l'on souhaite par exemple créer une tortue colorée (traçant des traits colorés) comme une « c-extension » de l'acteur tortue, il faut d'abord cloner `point2` puis cloner `tortue`, lier `clone-tortue` à `clone-point2` par un lien *est-un* et enfin créer le nouvel objet par extension de `clone-tortue`. Cet exemple introduit le problème du clonage d'un objet placé dans une hiérarchie d'objets (cf. paragraphe 8.8.3).

### 8.4.1 Description simplifiée des objets

Le processus de raisonnement humain fait souvent passer l'exemple avant l'abstraction [Lieberman, 1986], l'accumulation d'exemples menant à terme à une généralisation. Or le modèle à classes oblige le programmeur à formaliser un concept, une abstraction, avant de permettre la manipulation de représentants (d'exemples, d'instances) de ce concept. Les langages à prototypes proposent un modèle de programmation plus proche de la démarche cognitive, s'appuyant sur les exemples, attribuée à l'humain face à un problème complexe. Un langage à prototypes permet la description et la manipulation d'objets sans l'obligation préalable d'avoir à décrire leur modèle abstrait.

### 8.4.2 Modèle de programmation plus simple

Le modèle à classes est complexe [Borning, 1986 ; LaLonde *et al.*, 1986 ; Lalonde, 1989 ; Stein *et al.*, 1989], parce que les classes y jouent différents rôles qu'il est parfois difficile de dissocier et qui peuvent rendre leur conception, leur mise en œuvre et leur maintenance difficile. Notons parmi ces rôles : descripteur de la structure des instances, bibliothèque de comportements pour les instances, support de l'encapsulation et support à l'implantation de types abstraits, à l'organisation des programmes, à la modularité, au partage entre descriptions de concepts, à la réutilisation. De plus, le même lien entre classes est utilisé pour modéliser différentes relations, entre concepts ou entre types abstraits (suivant la vision que l'on a d'une classe à un instant donné), subtilement différentes les unes des autres [Lalonde, 1989] : héritage de spécifications, héritage d'implantations, sous-typage ou ressemblance [14]. Enfin, dans un système intégrant des méta-classes (comme SMALLTALK ou CLOS), la classe se voit de plus dotée du rôle d'instance, pouvant recevoir des messages et mener, si l'on peut dire, sa propre vie. La « sur-utilisation » du même support (la classe) tend à rendre complexe la programmation par objets, et plus encore la réutilisation de programmes existants.

De ce constat est issue l'idée de rechercher d'autres formes d'organisation pour les programmes et d'autres manières de représenter les objets. L'idée initiale de la programmation par prototypes est de réaliser des programmes en ne manipulant qu'une seule sorte d'objets privés du rôle de descripteur. Cette idée suppose que parmi les différents rôles joués par les classes, certains sont soit non indispensables soit modélisables autrement.

### 8.4.3 Expressivité

Les prototypes ont été utilisés dans de nombreux langages de représentation de connaissances, par exemple dans KRL, pour la souplesse de représentation qu'ils autorisent. L'absence de classes permet de relâcher certaines des contraintes qui pèsent sur leurs instances. Les prototypes autorisent notamment la définition de

14. Par exemple en SMALLTALK, la classe des ensembles (*Set*) est une sous-classe de la classe des collections non ordonnées quelconques (`Bag`) : « *A set is like a bag except that duplicates are not allowed* » [Lalonde, 1989].

caractéristiques distinctes pour différents objets d'une même famille conceptuelle, l'expression du partage de valeurs d'attributs entre objets, l'évolution aisée de leur structure ainsi que l'expression de connaissances par défaut, incomplètes ou exceptionnelles. La description de certaines connaissances pose, dans un langage à classes, des problèmes que la programmation par prototypes permet de résoudre. En voici une liste non exhaustive.

• **Instances différenciées ou exceptionnelles.**

Considérons en premier lieu le cas des instances exceptionnelles [Etherington et Reiter, 1983 ; Borgida, 1986 ; Dony, 1989] ayant des caractéristiques propres que les autres objets de la même famille n'ont pas. La manière standard de représenter une instance exceptionnelle, par exemple `Jumbo` l'éléphant qui sait voler, ou la liste vide dont les méthodes *car* et *cdr* s'implantent différemment de celles des listes non vides, est de créer une nouvelle classe pour les représenter. On trouve, en SMALLTALK par exemple, des classes n'ayant qu'une seule instance : `True`, `False` ou `UndefinedObject`. Même si des alternatives existent pour définir des propriétés au niveau d'un objet, elles sont restées marginales [15].

La représentation d'objets exceptionnels pose évidemment moins de problèmes dans un langage basé sur la description d'individus. On peut par exemple y définir l'éléphant sachant voler par clonage d'un autre éléphant et ajout de la propriété. La représentation de booléens évoquée ci-dessus est tout à fait naturelle dans un langage à prototypes. La possibilité de ne pas créer une classe pour chaque instance exceptionnelle est particulièrement intéressante lorsque celles-ci sont nombreuses (on peut penser par exemple à la modélisation des règles de la grammaire de la langue française [Habert et Fleury, 1993]).

• **Représentation de points de vues d'une même entité.**

Le modèle à classes ne permet pas à des instances de partager des attributs ou des valeurs d'attributs. Posons le problème de la représentation d'une personne et de divers points de vues sur cette personne, par exemple le point de vue « employé » ou le point de vue « sportif ». Il n'existe pas de solution standard dans un langage à classes pour résoudre ce problème. Le schéma classique de spécialisation de la classe `Personne` définissant un attribut `âge`, par deux sous-classes `Employé` et `Sportif` (Fig. 8.5), ne répond pas à la question car deux instances respectives de ces deux sous-classes représentent des entités différentes indépendantes en terme d'état ; si l'âge de l'une change, l'autre n'est pas affecté. D'autres solutions standard à ce problème (utilisant des constructions présentes dans la majorité des langages à classes) sont insatisfaisantes ; décrivons-en quelques-unes.

- On peut imaginer utiliser la composition : redéfinir les classes `Sportif` et `Employé` non plus comme des sous-classes de `Personne`, mais comme possédant

15. L'expression « poser problème » que nous avons employée ne dénote en effet pas nécessairement une impossibilité de représentation : il est souvent possible d'adapter, dans une implantation donnée, le modèle à classes pour lui faire faire ce que l'on souhaite. Certaines de ces adaptations ne dénaturent pas le modèle mais affectent l'efficacité de la recherche de méthode comme par exemple le qualifieur « `eql` » de CLOS ; d'autres nécessitent des constructions spéciales et le rendent plus complexe comme les *patterns* d'objets du langage BETA [Kristensen *et al.*, 1987] ; d'autres enfin sont intrinsèquement contradictoires avec le modèle (de vraies instances différenciées [Stein *et al.*, 1989] par exemple) et font que le résultat de l'adaptation engendre d'autres problèmes sémantiques.

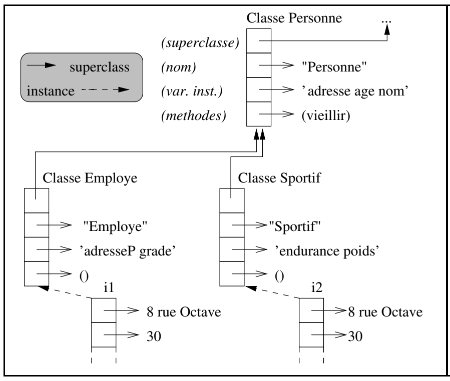



FIG. 8.5: *Duplication de la valeur de la propriété* `âge`.

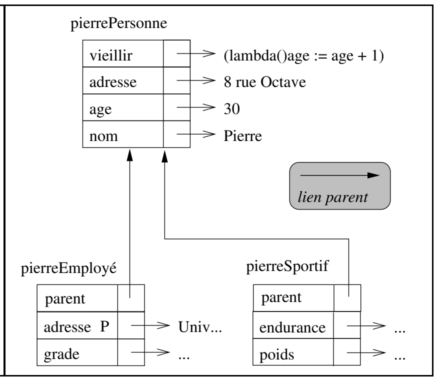



FIG. 8.6: *Partage de propriété.*

une variable d'instance de type `Personne`. Le problème est alors reporté sur l'accès aux attributs et aux méthodes de la personne : comment demander à un sportif son âge ? D'une part, il y a un problème d'encapsulation si l'attribut `âge` est privé, d'autre part, il est nécessaire de redéfinir toutes les méthodes de `Personne` sur `Employé` et `Sportif` afin d'y mettre une ré-expédition de message [16].

- Une seconde solution consiste à créer une nouvelle classe `EmployéSportif`, sous-classe de `Employé` et de `Sportif`. Le défaut ici est que la hiérarchie résultante peut devenir rapidement inexploitable si l'on souhaite créer de multiple extensions de la classe `Personne` et les combiner.

- Les variables de classes à la SMALLTALK ou leurs équivalents permettent à des instances de partager des valeurs mais la portée de ces variables est trop large, définir une telle variable au niveau de la classe `Personne` confère le même âge à toutes les instances de `Personne`, `Sportif` et `Employé`.

C'est parce qu'il n'y a pas de bonne solution standard à ce problème que des mécanismes spécifiques de représentations de facettes d'une même entité (on parle de points de vues) ont été développés ; qu'il s'agisse de la « multi-instanciation » de ROME [Carré, 1989], ou du mécanisme de multi-hiérarchies (une par point de vue) avec passerelles de communication développé dans TROEPS [Mariño *et al.*, 1990] ou encore de la notion du maintien par par des démons de contraintes entre objets co-référents [Ferber et Volle, 1988]. Dans un monde de prototypes, l'héritage entre objets permet de répondre simplement à la question posée (Fig. 8.6). Deux objets représentant la partie « employé » et la partie « sportif » de la personne peuvent être définis comme des extensions de l'objet représentant la « personne primitive » et détenant l'attribut `âge`; cet attribut et donc sa valeur étant alors partagés par les trois objets.

16. Le corps de la méthode `âge` de `Sportif` est un envoi du message `âge` à une instance de `Personne`.

• **Objets incomplets.**

La possibilité pour un objet d'hériter les valeurs des attributs d'un autre objet est utilisée intensivement dans les langages de *frames* pour représenter des objets incomplets *i.e.* des objets dont certaines valeurs d'attributs ne sont pas connues mais dont des valeurs par défaut peuvent être trouvées dans les objets dont ils héritent. Par exemple si on cherche ce que mange `Moby-Dick`, on trouvera une valeur dans le *frame* `baleine` dont `Mobby-dick` hérite (cf. Fig. 8.3). Les valeurs héritées peuvent changer au gré de l'évolution des parents d'un objet.

Cette possibilité de manipulation d'objets incomplets ne peut être comparée à la possibilité de spécifier, dans la définition d'une classe, des valeurs initiales pour les différents attributs de ses futures instances. Ces valeurs sont utilisées à l'instanciation, il n'existe ensuite plus aucune relation entre un objet et sa classe pour ce qui concerne les valeurs des attributs. Ne pas avoir d'indirections dans l'accès aux attributs est le gage d'une compilation efficace; il ne cache pas une impossibilité de représentation liée au modèle à classes [17].

## 8.5 Premières propositions de langages à prototypes

### 8.5.1 Langages fondés sur les instances prototypiques et la délégation

Henry Lieberman a repris dans [Lieberman, 1986 ; Lieberman, 1990] certaines idées de ACT 1 pour les appliquer à la programmation par objets. Il a proposé un modèle de programmation basé sur les instances prototypiques et l'a mis en œuvre ultérieurement dans le langage OBJECT-LISP [Apple, 1989]. La figure 8.7 propose une version OBJECT-LISP de l'exemple « point-tortue » présenté au paragraphe 8.3.2. Le premier exemple concret d'un concept (`point`) sert de modèle pour définir les suivants. Les nouveaux objets sont créés par extension avec un équivalent de la primitive `extend` de ACT 1. Une extension possède un lien *est-un* vers son prototype. Les objets possèdent des attributs et des méthodes; ils communiquent en s'envoyant des messages (primitive `ask`). Il n'existe pas de mécanisme d'encapsulation : les variables sont accessibles par envoi de message (`(ask tortue x)`) ou directement dans le corps des méthodes. Le mécanisme de délégation est mis en œuvre aussi bien lors de l'accès à la valeur d'une variable que pour l'activation d'une méthode : si la propriété n'est pas trouvée chez le receveur alors la demande est déléguée à son parent.

Le point véritablement nouveau par rapport à ACT 1 est l'explicitation du mécanisme de liaison dynamique. Le mécanisme est conceptuellement parfaitement similaire à celui des langages à classes [18] (*chapitres 1, 3 et 2*). Dans notre exemple, l'envoi du message `norm` à `tortue` rend ainsi la valeur `5`, la méthode `norm` est

17. Il est évidemment possible d'implanter une telle relation, ce qui a été fait par exemple dans certains langages de représentation [Rechenmann, 1988]

18. Il permet au code d'une méthode d'être interprété dans le contexte des attributs et des méthodes du receveur initial du message, et ce, quel que soit l'endroit où la méthode a été trouvée. Nous supposons ce mécanisme, ainsi que ses applications à l'écriture de méthodes polymorphes, connu du lecteur.

```
(setq point (kindof))                  ;Création d'un objet ex nihilo.
(ask point (have 'x 3))                ;Création d'un attribut pour point.
(ask point (have 'y 10))
(defobfun (norm point) ()              ;Une méthode norm pour l'objet point.
  (sqrt (+ (* x x) (* y y))))          ;Les variables sont celles du receveur.
(defobfun (move point) (newx newy)     ;Une méthode avec paramètres,
  (ask self (have 'x (+ x newx))       ;pour additionner deux points.
  (ask self (have 'y (+ y newy)))      ;Modification des valeurs des attributs
(defobfun (plus apoint) (p)            ;Une méthode d'addition de deux points.
  (let ((newx (+ x (ask p x)))
        (newy (+ y (ask p y)))
        (newp (kindof apoint)))        ;création d'une extension de l'objet
    (ask newp (have 'x newx))          ;passé en argument.
    (ask newp (have 'y newy))
    newp))

(setq point2 (kindof point))           ;point2 est une extension de point,
(ask point2 (have 'y 4)                ;avec un nouvel attribut y.
(setq tortue (kindof point2))          ;Une extension de point2,
(ask tortue (have 'cap 90))            ;avec un nouvel attribut cap,
(defobfun (forward tortue) (dist)      ;et une méthode forward.
  (ask self
       (move (* dist (cos cap))
             (* dist (sin cap))))
```

OBJECT-LISP *est une extension de Lisp vers les objets autorisant l'envoi de messages ainsi que l'appel fonctionnel classique. L'envoi de message est réalisé par la fonction* `ask` *; son premier argument est le receveur et le second un appel fonctionnel ou le nom de fonction fait figure de sélecteur du message (il doit exister une méthode correspondant à ce nom) ; les arguments de l'appel fonctionnel sont les arguments du message. Les primitives suivantes sont utilisées dans l'exemple :*

*– création d'objets « ex nihilo » : fonction* `kindof` *sans arguments,*

*– création d'extensions : fonctions* `kindof` *(avec un argument qui est l'objet étendu),*

*– définition de méthodes : fonction* `defobfun`*,*

*– définition d'attributs : méthode* `have`.

FIG. 8.7: *Un exemple de langage à prototypes* – OBJECT-LISP.

trouvée dans le parent du receveur (`point2`) mais l'accès aux variables `x` et `y` est interprété dans le contexte du receveur initial (`tortue`), ce qui donne, via deux nouvelles délégations, 3 pour `x` et 4 pour `y`.

Un certain nombre de langages sont issus de ce modèle et ont, à la base, les mêmes caractéristiques : citons par exemple SELF, GARNET, NEWTON-SCRIPT ou MOOSTRAP. SELF est certainement le plus connu ; il a donné lieu au plus grand nombre de publications, a bénéficié d'un gros effort de développement et d'une large diffusion.

### 8.5.2 Langages fondés sur les instances prototypiques et le clonage

A la même époque, [Borning, 1986] a proposé une description informelle d'un monde d'objets sans classes organisé autour du clonage. Un prototype y représente un exemple standard d'instance et les nouveaux objets sont produits par copies et modifications de prototypes. Une fois la copie effectuée, aucune relation n'est maintenue entre le prototype copié et son clone. Conscient toutefois de la pauvreté du modèle ainsi obtenu, Borning proposait de l'étendre en instaurant une certaine forme d'héritage à base de contraintes [Borning, 1981]. Ce modèle n'a pas été développé par son auteur mais a inspiré les langages à prototypes basés sur le clonage comme KEVO [Taivalsaari, 1993], OMEGA [Blaschek, 1994] ou OBLIQ [Cardelli, 1995].

### 8.5.3 Langages intégrant hiérarchies de classes et hiérarchies d'objets.

Le problème a également été abordé plus directement sous l'angle de l'organisation des programmes : pour contourner les limitations que nous avons évoquées, a été imaginé [LaLonde *et al.*, 1986 ; Lalonde, 1989], à la même époque et parallèlement à celui de Lieberman, un modèle de programmation intégrant des classes et des instances (appelés cette fois *exemplars*, ce que l'on peut traduire par « exemplaires ») dotées d'une certaine autonomie. La principale raison d'être de cette proposition était d'expérimenter un découplage entre une hiérarchie de sous-typage, composée de classes et une hiérarchie de réutilisation d'implantations, composée d'instances. Les classes détiennent l'interface des types abstraits qu'elles implantent. Les instances détiennent les méthodes et sont organisées en une hiérarchie de délégation. L'héritage des méthodes se fait entre instances, celui des spécifications entre classes. Une classe peut donc avoir deux instances possédant des méthodes implantées différemment. Par exemple la classe `Liste` a une instance `listeVide` et une autre `listeNonVide` possédant des versions différentes de la méthode `append`, mais les deux instances ont la même interface définie par la classe. Par ailleurs, une instance peut hériter de n'importe quelle autre instance certaines méthodes privées nécessaires à l'implantation des méthodes composant son interface. Par exemple la classe `Dictionnaire` n'est logiquement pas définie comme une sous-classe de la classe `Ensemble`, mais une instance de `dictionnaire` peut hériter d'une instance de la classe `Ensemble` lorsque qu'un dictionnaire est implanté comme un ensemble de couples « clés – valeurs ». La hiérarchie de délégation entre instances n'est pas nécessairement isomorphe à celle d'héritage entre classes.

Cette proposition pose un certain nombre de problèmes qu'il serait trop long de décrire ici ; les auteurs l'ont d'ailleurs abandonnée, n'ayant pas réussi à faire la synthèse entre le rôle des classes et l'héritage entre instances. Cette étude reste cependant intéressante. D'une part, l'héritage entre instances est tout à fait similaire à celui proposé par Lieberman et a donc inspiré au même titre les langages à prototypes ultérieurs. D'autre part, elle proposait une première tentative d'utilisation de la délégation dans un monde de classes, idée qui revient aujourd'hui à l'ordre du jour.

### 8.5.4 Langages de frames pour la programmation

Certains langages, dits hybrides [Masini *et al.*, 1989], autorisent la définition de méthodes dans un monde dédié à la représentation et fondé sur les *frames* comme

YAFOOL [Ducournau, 1991]. On peut donc les assimiler aux langages à prototypes. L'originalité de ces langages par rapport à ceux qui se voulaient uniquement basés sur les instances prototypiques (comme SELF ou OBJECT-LISP) est qu'ils permettent de définir des hiérarchies intégrant des représentants moyens (comme `Animal`) comme celle de la figure 8.3. Nous verrons que les concepteurs de la plupart des langages à prototypes ont dû réintroduire cette possibilité.

## 8.6 Récapitulatif des concepts et mécanismes primitifs

Voici une synthèse des propositions précédentes qui permet d'isoler les concepts fondamentaux des langages à prototypes.

• **Objets sans classes**. La caractéristique commune à tous les objets des langages à prototypes est de ne pas être liés à une classe, de ne pas avoir de descripteur. L'objectif de ne manipuler que des objets concrets, affiché par SELF par exemple, n'est pas généralisé, nous avons vu qu'il existait des langages permettant de manipuler des représentants moyens. Certains auteurs parlent également d'objets autonomes mais cette caractéristique n'est pas générale en programmation par prototypes; les objets ne sont pas non plus véritablement autonomes dès lors qu'ils sont créés comme des extensions d'autres objets.

• **État et comportements**. Les objets sont définis par un ensemble de propriétés. Une propriété est à la base un couple « nom[19]– valeur[20] ». Les propriétés sont soit des attributs, auquel cas la valeur est un objet quelconque, soit des méthodes, la valeur est alors une fonction.

• **Envoi de messages**. Les objets communiquent par envoi de messages ; ils sont capables de répondre aux messages en appliquant un de leurs comportements (ou éventuellement en rendant la valeur d'un de leurs attributs). Ils peuvent être vus comme des *frames* sans facettes dotés de procédures et répondant à des messages comme les acteurs ACT 1.

• **Trois formes primitives de création d'objets**. Les objets peuvent être créés soit *ex nihilo*, soit en copiant un objet existant (clonage), et dans certains langages en étendant un objet existant (extension ou création différentielle).

• **Héritage**. Dans le cas de la création différentielle, un nouvel objet est créé comme une extension d'un objet existant, qui devient son parent. La relation « est-extension-de » lie le nouvel objet et son parent. Il s'agit d'une relation d'ordre qui définit des hiérarchies d'objets. Dans une hiérarchie, une extension hérite les propriétés de son parent qui n'ont pas été redéfinies à son niveau. Si une propriété héritée est un attribut alors l'extension possède cet attribut, si c'est une méthode alors elle lui est applicable. La relation est matérialisée par un lien appelé lien « parent » ou « lien de délégation ». Ce lien est parfois accessible au programmeur;

19. Une propriété est unique dans le système mais plusieurs propriétés peuvent avoir le même nom (c'est ce que nous appelons « surcharge »).

20. Une propriété peut également posséder un type, un domaine, une signature, des facettes, etc.

en SELF par exemple, chaque objet possède au moins un attribut nommé `parent` contenant l'adresse de son parent.

• **Délégation**. La délégation est le nom donné au mécanisme qui met en œuvre l'héritage, c'est-à-dire qui cherche et active une propriété héritée. La délégation est généralement implicite [21] [Stein *et al.*, 1989 ; Dony *et al.*, 1992]. Dans ce cas, le terme « déléguer » est une image ; dans la pratique, le système, lorsqu'il ne trouve pas de propriété dans le receveur du message, la recherche dans ses parents successifs et s'il la trouve, l'active [22]

• **Liaison dynamique**. Les langages à prototypes sont des langages où les schémas usuels de réutilisation des langages à objets s'appliquent. Ainsi, l'activation d'une propriété s'effectue toujours dans le contexte du receveur initial du message. Dans toute méthode, la variable `self` (ou un équivalent) désigne l'objet qui a effectivement reçu le message et non celui à qui on l'a délégué (*i.e.* celui où la méthode à été trouvée). L'accès à un attribut ou l'envoi d'un message à `self` nécessitent donc un mécanisme de liaison dynamique.

Nous avons à ce point de l'exposé une idée générale de ce que sont les langages à prototypes et les possibilités nouvelles qu'ils offrent. Les langages existants proposent néanmoins un ensemble de variations subtiles autour des concepts que nous avons présenté. Ces variations résultent d'une part d'interprétations différentes données aux concepts précédents (par exemple à celui d'extension) et d'autre part à la nécessité, pour les concepteurs, de résoudre des problèmes non envisagés initialement (comme celui de la gestion de propriétés communes à des ensembles d'objets).

## 8.7 Caractérisation des mécanismes et interprétation des concepts

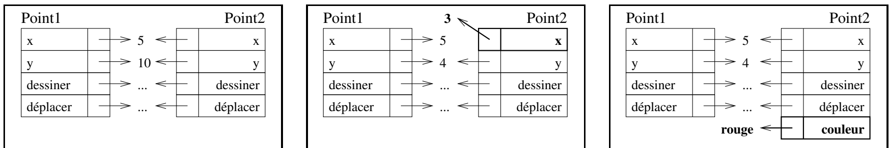
Les points, sources de confusions, qui réclament plus particulièrement des précisions sont : la caractérisation de la différence entre clonage et création différentielle, la caractérisation de la différence entre l'héritage dans les langages à classes et dans les langages à prototypes, et enfin la compréhension des diverses utilisations possibles du concept d' « extension ». Nos caractérisations sont fondées sur l'héritage et le partage, ce qui est partagé ou hérité est relatif aux propriétés. On distingue trois formes de partage [23].

- Il y a partage de noms lorsque deux objets ont chacun une propriété de même nom.

21. Dans l'autre alternative, la délégation explicite, l'objet dispose d'un module de réception des messages et choisit lui-même la propriété à activer ou délègue lui-même le message à un autre objet. La délégation explicite est citée dans certains articles [Stein *et al.*, 1989] et est utilisée dans ACT 1 mais nous ne connaissons pas de langages à prototypes qui l'utilisent.

22. Le lecteur trouvera dans [Malenfant, 1995] une description formelle de la sémantique du mécanisme de délégation pour un langage à la Lieberman et dans la plate-forme PROTOTALK [Dony *et al.*, 1992] une mise en œuvre d'évaluateurs correspondants qu'il pourra étudier, étendre et modifier à sa guise. PROTOTALK est une plate-forme SMALLTALK permettant de simuler simplement la plupart des langages à prototypes ; elle est disponible à l'adresse : *http://www.lirmm.fr/˜dony/research.html*.

23. Une description plus détaillée et plus formelle en est donnée dans [Bardou et Dony, 1996].




(a): `Point2` *est obtenu par clonage de* `point1`

(b): *Après modification de la valeur de* `x` *de* `point2`

(c): *Après ajout de la propriété* `couleur` *dans* `point2`

FIG. 8.8: *Clonage : partage ponctuel*

- Il y a partage de valeurs lorsque deux objets `o1` et `o2` ont chacun une propriété de nom *x* et que la valeur de la propriété `x` de $o_1$ et de $o_2$ est la même (égalité physique).
- Il y a partage de propriétés [24] lorsque deux objets possèdent la même propriété (même adresse et donc même nom et même valeur). Il peut y avoir partage de valeurs sans qu'il y ait partage de propriétés.

### 8.7.1 Distinction entre clonage et extension

La distinction entre clonage et extension, montrant que ces deux mécanismes ne sont pas redondants, provient de ce que le clonage et la création différentielle induisent deux formes de partage distinctes [Dony *et al.*, 1992].

• **Une caractérisation du clonage**. Tout objet cloné, partage avec son clone, au moment où celui-ci est créé, les noms et les valeurs de ses propriétés. Le clone et son original évoluent indépendamment, la modification d'un attribut du premier n'est pas répercutée sur le second : le partage est ponctuel. En effet, tout nom de propriété, par exemple `x`, `y`, `déplacer` ou `dessiner` (Fig. 8.8.a) de l'objet cloné, `point1` dans notre exemple, est aussi un nom de propriété du clone (`point2`) et, les deux propriétés désignées par chaque nom ont la même valeur. Mais il y a bien deux propriétés ; par exemple, si l'on modifie la valeur de la propriété `x` pour `point2`, la valeur de la propriété de même nom de `point1` n'est pas modifiée (Fig. 8.8.b) ; idem en cas d'ajout ou de retrait de propriété. (Fig. 8.8.c).

• **Caractérisation du mécanisme d'extension**. Un objet dans une hiérarchie d'objets hérite de ses parents un ensemble d'attributs et de méthodes. Tout parent partage avec ses extensions les propriétés [25] qu'il définit et que ces derniers n'ont pas redéfinies. Les propriétés du parent non redéfinies dans une extension sont aussi

24. L'idée de différencier le partage de noms du partage de propriétés est inspirée de la distinction entre héritage de noms et de valeurs introduite dans [Ducournau *et al.*, 1995]. Les deux distinctions n'ont en fait pas grand chose en commun car elle sont fondées sur des interprétations différentes de la notion de propriété.

25. Ce qui est hérité étant identique pour les attributs et les méthodes, il est possible d'unifier les deux sortes de propriétés ; c'est ce qu'ont fait les concepteur de SELF. On ne trouve en SELF que des *slots* dont les valeurs peuvent éventuellement être des fonctions exécutables, on y accède dans tous les cas par envoi de message ; lorsque la valeur d'un slot est une fonction alors elle est exécutée.

des propriétés de l'extension. L'extension est dépendante de son parent[26]. Cette dépendance et ce partage sont persistants, ils durent aussi longtemps qu'existe le lien entre les deux objets. Reconsidérons l'exemple de la figure 8.4 dans lequel l'objet `tortue` a pour parent l'objet `point2`. `Tortue` ne détient pas la propriété `x` mais hérite celle détenue par `point2`. Cette propriété définit tout autant `tortue` que `point2`, elle est partagée par les deux objets. Toute modification de la valeur de cette propriété pour `point2` affecte également l'objet `tortue`.

Les deux mécanismes induisent donc des partages distincts avec des durées de vie différentes (ponctuel pour le clonage, persistant pour l'extension). Leurs applications sont distinctes.

### 8.7.2 Caractérisation de la différence entre hiérarchies d'objets et hiérarchies de classes

L'étude comparative de l'héritage dans les hiérarchies de classes et dans les hiérarchies d'objets (hiérarchies de délégation) a fait l'objet de plusieurs travaux. D'après Lieberman, la délégation est un mécanisme plus général que l'héritage de classes, elle permet de le simuler. Dans [Stein, 1987], il est montré que la simulation inverse est possible à condition d'utiliser les classes comme des objets et de se servir des variables de classe à la SMALLTALK pour représenter les propriétés, ce qui est un cas très particulier. Cet article ne montre par ailleurs pas du tout ce que son titre laisse supposer, à savoir que la délégation est équivalente à l'héritage dans les hiérarchies de classes.

En fait les deux formes d'héritages sont distinctes car l'héritage dans les langages à classes n'induit aucun partage d'attributs entre instances[27]. Un objet, instance d'une classe `C` possède, via sa classe : d'une part un ensemble de méthodes (déclarées et définies dans les super-classes de `C`) et d'autre part un ensemble de noms d'attribut (déclarés dans les super-classes de `C`) dont il détient en propre la valeur. Deux objets dont les classes sont liés par un lien « sous-classe-de » partagent donc des méthodes mais uniquement des noms d'attribut. Ces objets sont indépendants en terme d'états.

Les deux formes d'héritage ne sont donc pas équivalentes. L'héritage d'attributs entre objets est caractéristique des hiérarchies d'objets ; il est à l'origine des possibilités nouvelles de représentation offertes par les langages à prototypes (cf. paragraphe 8.4), mais aussi de problèmes nouveaux.

### 8.7.3 Variations sur la dépendance entre parent(s) et extensions

L'héritage d'attributs induit un partage qui rend une extension dépendante de son parent, mais qu'en est-il de la réciproque ? Un parent est-il dépendant de ses extensions ou en d'autres termes, une extension peut-elle, en se modifiant, modifier aussi son parent? Le problème se pose lorsque l'on demande à un objet (par envoi

26. La réciproque, à savoir l'indépendance du parent vis-à-vis de ses extensions sera discutée au paragraphe 8.7.3.
27. Sauf pour le cas très particulier des variables de classe.

de message ou par un autre moyen) de modifier la valeur d'un attribut qu'il hérite. Déléguer ou ne pas déléguer les accès en écriture aux attributs, telle est la question. Considérons par exemple l'envoi à `tortue` du message `move`, qui provoque l'activation de la méthode `move` détenue par `point2`, laquelle accède en écriture aux attributs `x` et `y` du receveur initial (liaison dynamique); ce dernier (`tortue`) ne détenant en propre que l'attribut `y`. Comment interpréter l'affectation « `x := newX` » ? La réponse à cette question dépend de l'interprétation que l'on a de la notion d'extension et des possibilités que l'on veut offrir. Les langages à prototypes divergent sur ce point.

• **Interprétation No 1**. Des langages tels que YAFOOL ou GARNET ne délèguent pas l'accès en écriture aux attributs. L'affectation est alors réalisée dans le contexte strict du receveur initial : lorsque celui-ci ne possède pas l'attribut considéré, cet attribut doit être créé avant la réalisation de l'affectation proprement dite. Dans notre exemple, l'affectation de la variable `x` sera précédée par une redéfinition automatique de la propriété `x` sur `tortue`.

Cette solution limite le partage d'attributs entre objets à du partage de valeurs : le parent partage avec ses extensions uniquement le nom et la valeur de ses propriétés non redéfinies. Elle rend le parent complètement indépendant de ses extensions [28].

Dans ce contexte, une extension représente systématiquement une entité différente de celle représentée par le parent, duquel elle hérite néanmoins certaines caractéristiques. Cette interprétation est ainsi adaptée à la mise en œuvre de l'exemple « point-tortue » dans lequel deux entités différentes, représentées par les objets `point2` et `tortue` ont la même abscisse. Mais en restreignant le partage d'attributs, cette solution interdit également certaines utilisations de la délégation telle que celle utilisée dans l'exemple « personne, employé, sportif » [29] (cf. Fig. 8.6).

• **Interprétation No 2**. Des langages tels que SELF ou OBJECT-LISP délèguent l'accès en écriture aux attributs. L'affectation est alors toujours réalisée, quel que soit le receveur initial, au niveau de l'objet détenant l'attribut, dans notre exemple il s'agit de l'objet `point2`. Dans ce contexte, un parent est dépendant de ses extensions. Une extension représente la même entité que son parent, elle en décrit une partie spécifique. Cette interprétation permet ainsi de représenter l'exemple « personne-employé-sportif », dans lequel les extensions représentent des parties d'un tout, ce tout étant la représentation d'une personne.

On pourrait croire en première analyse qu'il est néanmoins possible de se ramener à la première interprétation à condition de redéfinir sur une extension tous les attributs définis par ses parents. En fait, utiliser cette seconde interprétation pose, si l'on souhaite par exemple représenter le point et la tortue, un ensemble de problèmes que nous nous proposons d'aborder maintenant.

28. On n'obtient pas pour autant un équivalent du clonage car l'état de l'extension est toujours dépendant du parent.

29. En effet, l'envoi d'un message à `Sportif` pour modifier son adresse, résulterait alors en une redéfinition de `adresse` dans `Employé` et non en une modification de `adresse` au niveau de `Personne`.

## 8.8 Discussion des problèmes relatifs à l'identité des objets.

Les premiers problèmes que pose la programmation par prototypes sont relatifs à l'identité des objets. Nous utiliserons le terme « identité » pour désigner l'entité (ou les entités) du domaine qu'un objet représente. Avec le modèle classe-instance, un objet a une identité unique, il représente une et une seule entité du domaine *chapitre 1*. L'objet y est par ailleurs une unité d'encapsulation autonome détenant l'ensemble des valeurs des attributs ; toute modification de la valeur d'un de ces attributs est sous son contrôle. Avec l'héritage entre objets, cette bijection entre entités décrites et objets peut ne plus exister. En effet, dans une hiérarchie, un même objet peut représenter à la fois une entité, plusieurs entités ou des parties de plusieurs autres entités. Par exemple, l'objet `point2` de la figure 8.4 représente un point, mais également une partie d'une tortue puisqu'il détient son abscisse. Les représentations des entités point et tortue partagent les propriétés définies dans l'objet `point2`. Dans l'autre exemple, l'entité personne est représentée par les trois objets `personne`, `employé` et `sportif`.

Dès lors qu'un objet définit des propriétés représentant plusieurs entités, se pose le problème de la modification accidentelle d'une entité suite à la modification d'une propriété partagée.

### 8.8.1 Problèmes potentiels d'intégrité

Le premier problème potentiel est la modification d'une extension par l'intermédiaire de son parent. Il se pose avec les deux interprétations de la notion d'extension. Par exemple, l'envoi à l'objet `point2` du message `move` provoque la modifications d'attributs de `point2` et subséquemment des entités point et tortue, car l'attribut `x` est partagé [30]. Ce résultat peut néanmoins être considéré comme une conséquence naturelle de l'utilisation de la description différentielle. Si le programmeur de `tortue` ne souhaite pas que cet objet dépende de `point2`, il peut utiliser le clonage. Il y a bien modification indirecte d'une entité mais elle correspond à l'intention du programmeur.

Le second problème potentiel est la modification accidentelle (non prévue par le programmeur) d'un parent via une de ses extensions. Ce problème ne se pose qu'avec la seconde interprétation de la notion d'extension utilisée pour représenter des entités différentes. L'exemple en est l'envoi du message `move` à `tortue` que nous avons décrit et qui modifie l'objet `point2`, ce qui évidemment ne correspond pas nécessairement à l'intention du programmeur. Dans cette configuration, le lien de délégation octroie, à `tortue` un accès en lecture et en écriture aux propriétés définies dans `point2`. Demander à la tortue de se déplacer entraîne également le déplacement du point.

Il est ainsi possible de modifier plusieurs entités en pensant n'en modifier qu'une. Plus généralement, il est impossible de placer une frontière nette entre des entités

30. La distinction entre objet et entité apparaît clairement ici, la tortue a bien été modifiée alors que l'objet `tortue` ne l'a pas été.

représentées par des objets appartenant à une même composante connexe d'une hiérarchie. Ces composantes devenant en pratique les réelles unités d'encapsulation des langages à prototypes ([Chambers *et al.*, 1991] emploie le terme d'*encapsulation de modules*). Une affectation peut entraîner la modification d'un très grand nombre d'entités sans qu'il soit aisé de prédire lesquelles ou même leur nombre. Briser l'encapsulation dans un tel contexte devient extrêmement simple. Il suffit, pour accéder en lecture et en écriture aux attributs d'un objet `O`, d'en créer une extension `E`, d'y définir une méthode réalisant un accès en écriture aux attributs définis sur `O` et d'envoyer le messages correspondant à `E`. Les variables d'un objet deviennent de fait des variables semi-globales, modifiables par n'importe lequel de ses descendants.

### 8.8.2 Solutions aux problèmes d'intégrité

Toutes les solutions proposées au problème précédent passent par de la limitation du partage d'attributs.

• **Responsabilité du programmeur.** Des langages comme SELF ne proposent aucune solution à ce problème. La solution standard pour le programmeur est de créer des extensions en redéfinissant systématiquement tous les attributs de ses parents et en n'héritant que les méthodes. Même avec cette précaution, il est impossible, comme nous l'avons montré, d'assurer l'encapsulation.

• **Restriction du partage d'attributs.** En restreignant le partage d'attributs à du partage de valeurs (cf. paragraphe 8.7.3), les langages comme YAFOOL ou GARNET solutionnent le problème au détriment du pouvoir d'expression du langage.

• **Distinction entre liens de délégation.** Offrir en standard les deux possibilités est tentant, une solution mixte a ainsi été implantée dans le langage *NewtonScript* [Smith, 1995] où le programmeur a le choix entre deux sortes de liens de délégation. A l'un de ces liens (lien `_proto`) est associé du partage de valeurs et une sémantique de valeurs par défaut, tandis que du partage d'attributs est associé à l'autre (lien `_parent`). L'existence de deux types de liens de délégation complique cependant considérablement le modèle de programmation, la lisibilité des programmes et la recherche de sélecteurs. En effet, bien que le lien `_proto` soit prioritaire au lien `_parent`, il est difficile de prévoir ce qui peut se passer lorsque le sélecteur recherché est accessible en suivant deux chemins différents (incluant éventuellement des liens des deux types).

• **Contrôle des extensions.** Le langage AGORA [Steyaert, 1994] permet à chaque objet de contrôler la création de ses futures extensions. Il est impossible d'étendre un objet, s'il ne possède pas de méthodes particulières, appelées « *mixin-methods* » permettant de spécifier de façon précise les droits d'accès en lecture et en écriture aux propriétés qui seront octroyés à ses descendants. Cette solution rend au programmeur le contrôle total des accès aux propriétés d'un parent, son principal inconvénient réside dans le fait que celui-ci doit systématiquement prévoir toutes les possibilités d'extension, ce qui exclut toute réutilisation non anticipée; le problème est similaire à celui du choix de la « virtualité » des méthodes en C++ (*chapitre 3*).

• **Cas des langages excluant la création d'extensions.** Tous les langages à prototypes n'incluent pas le mécanisme de délégation. Les problèmes évoqués étant directement liés au partage de propriétés qu'il induit, il va de soi que ces problèmes ne se posent pas dans ces langages (comme KEVO), qui ont en contrepartie un pouvoir d'expression plus limité.

### 8.8.3 Problème de la gestion des entités morcelées

Le partage d'attributs crée un autre problème, connexe aux précédents. Lorsqu'une entité est représentée par plusieurs objets, nous parlons alors d' « entité morcelée » [Dony *et al.*, 1992 ; Malenfant, 1996 ; Bardou et Dony, 1996] ; c'est par exemple le cas de l'entité tortue, dont la représentation utilise les objets `point2` et *tortue*. Le problème est qu'il n'existe aucun objet du langage représentant l'entité tortue dans sa globalité. On pourrait considérer que l'objet `tortue` joue ce rôle mais ceci lui confère un double statut, celui de représentant de tortue et en même temps celui de représentant d'une de ses parties. Ce double statut se manifeste si l'on demande à l'objet `tortue` de se cloner, demande-t-on un clonage de l'objet ou un clonage de l'entité ? Dans le premier cas, seul l'objet doit être copié. Dans le second cas, il est nécessaire de réaliser une copie de toutes les parties de la tortue, c'est-à-dire une copie des objets `tortue` et `point2`, similaire à celle réalisée par la primitive `c-extent` de ACT 1. Le double statut se manifeste dans cet exemple par l'existence de deux primitives de clonage, ayant des noms différents, entre lesquelles le programmeur doit choisir et qui sont sources de confusion. Le problème se révèle encore mieux sur un exemple plus complexe, par exemple celui de la personne « pierre » (cf. Fig. 8.6) représentée par les objets `pierrePersonne`, `pierreEmployé` et `pierreSportif`. Il n'existe plus dans ce cas aucun objet susceptible de représenter l'entité « pierre » dans sa globalité. Si on souhaite la cloner, aucune primitive du langage n'est capable de réaliser cette opération puisque cette entité n'est pas réifiée ; un tel clonage doit être réalisé de façon *ad hoc* par le programmeur. On ne trouve aucune solution à ce problème dans les langages existants.

### 8.8.4 Utilisation sémantiquement fondée de la délégation

Nous pouvons à cet instant faire un point sur les utilisations sémantiquement fondées du mécanisme de délégation. Tous les problèmes que nous venons d'évoquer viennent du fait qu'une même entité du domaine puisse être représentée par plusieurs objets partageant des attributs. Les utilisations cohérentes de la délégation sont celles ou une bijection est rétablie entre objets du langages et entités représentées.

Une première solution à ce problème a été décrite, elles consiste à restreindre le partage de propriétés à du partage de valeur et correspond à notre premie're interprétation de la délégation (cf. section 8.7.3).

Une seconde solution consiste à utiliser la délégation pour réaliser un partage de propriétés, non plus entre objets mais entre morceaux formant les différentes parties d'un objet, l'objet lui-même représentant une entité du domaine. On peut ainsi

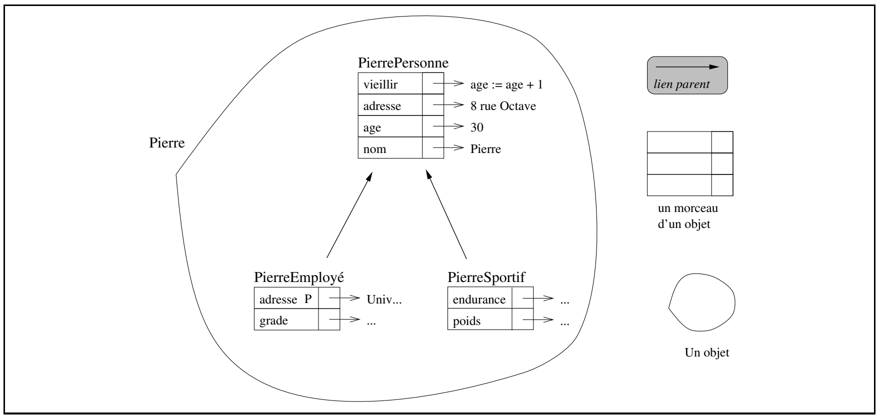
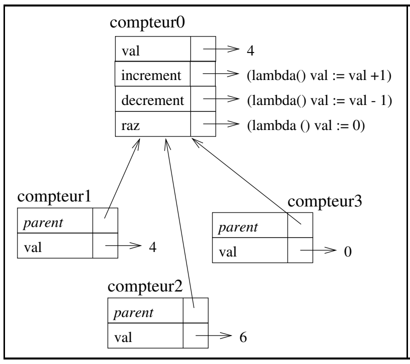



FIG. 8.9: *Objets morcelés.*

imaginer l'objet `Pierre` représentant une personne, formé de 3 «morceaux» comme dans la figure 8.9, ces morceaux ayant alors perdu leur statut d'objet. Une application évidente des objets morcelés est la représentation de points de vue [Carré, 1989 ; Ferber, 1989b] de l'entité correspondante ; mais il peut y en avoir d'autres. Des études sont en cours pour intégrer une représentation explicite d'objets morcelés dans des langages, à prototypes ou à classes [Bardou et Dony, 1995 ; Bardou et Dony, 1996 ; Bardou *et al.*, 1996] ou dans les bases de données à objets [Naja et Mouaddib, 1995]. Dans cette optique, la délégation a aussi été utilisée comme technique d'implantation du système de points de vue de *Rome* en SMALLTALK [Vanwormhoudt, 1998].



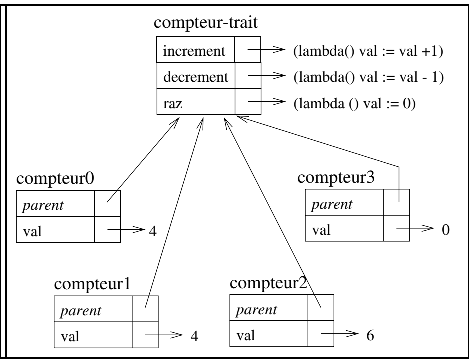




FIG. 8.10: *Solution des instances prototypiques.*

FIG. 8.11: *Solution des* `traits`.

## 8.9 Discussion des problèmes liés à l'organisation des programmes

La seconde série de problèmes de la programmation par prototypes est liée à la disparition de la représentation en intension des concepts. Confrontés à la réalisation de programmes complexes, les utilisateurs de langages à prototypes ont vite été limités par l'absence d'équivalents aux possibilités de partage offertes par les classes.

### 8.9.1 Problème du partage entre membres d'une famille de clones

Nous employons le terme de « famille de clones » pour désigner l'ensemble des objets mis en relation par la fermeture transitive de la relation « est-un-clone-de » qui lie conceptuellement un clone et son modèle. On peut assimiler une famille de clones à l'ensemble de tous les objets appartenant conceptuellement à un même type de données ou à l'ensemble des instances d'une classe.

Le premier problème est celui du partage des méthodes communes à tous les objets d'une famille de clones. Considérons par exemple un ensemble de clones de l'objet `point` de la figure 8.4, ces objets sont tous indépendants et il n'existe aucun objet représentant la famille. Considérons maintenant le problème suivant : comment doter tous les membres de la famille d'une nouvelle méthode (par exemple `moveToOrigin`). Diverses solutions ont été proposées pour introduire cette possibilité dans les langages à prototypes. Les premières sont de la responsabilité du programmeur et utilisent les constructions primitives existantes ; il s'agit des approches par instances prototypiques, représentants moyens ou *traits*. Les secondes déplacent le problème au niveau de l'implantation et proposent une gestion automatique du partage entre objets d'une même famille.

#### Gestion des familles de clones utilisant des prototypes

**La méthode dite des « instances prototypiques »**, initialement proposée dans [Lieberman, 1986], consiste à élever un des membres au statut de représentant de la famille. Dans la pratique l'instance prototypique devient le parent de tous les autres membres. La figure 8.10 montre un exemple d'instance prototypique (`compteur0`) pour une famille de « compteurs ». Doter tous les membres de la famille d'une nouvelle propriété s'effectue alors simplement en la définissant sur cet objet. Cette solution, bien qu'utilisable dans la pratique, est peu satisfaisante car l'instance prototypique peut être condidérée tantôt comme un individu, tantôt comme l'ensemble des individus qu'elle représente. Le statut particulier de représentant de la famille n'est en rien matérialisé ; rien ne distingue l'instance prototypique des autres objets. Pourtant les autres membres de la famille hériteront de toutes ses évolutions. L'évolution personnelle de l'instance prototypique peut devenir contradictoire avec son statut de représentant si cette évolution fait qu'il n'est plus représentatif. Imaginons par exemple un ministre représentant de ses congénères ayant des démélés avec la justice, caractéristique nouvelle que les autres ne souhaiterons pas hériter.

Pour la plupart des langages, on a abandonné l'idée de ne manipuler que des objets représentant des entités concrètes. La méthode des « représentants moyens » repose sur le même principe que celle des instances prototypiques mais un représentant moyen (cf. paragraphe 8.2) est choisi comme représentant de la famille, il est éventuellement doté d'attributs et éventuellement incomplet.

**La méthode des** `traits`, proposée par SELF [Ungar *et al.*, 1991], proche de la précédente consiste à créer des objets, appelés `traits`, ne contenant que les méthodes partagées par les objets de la famille (la figure 8.11 en montre un exemple). La méthodologie des `traits` suggère de diviser la représentation d'une nouvelle sorte d'entités en deux parties : un objet `trait` qui contient les méthodes factorisées et un prototype contenant les attributs et ayant le `trait` pour parent. Obtenir un nouvel objet de la famille de clones consiste alors à cloner le prototype mais pas le `trait`.

Les représentants moyens et les `traits` ont le même double statut que les instances prototypiques. Le `trait` est une bibliothèque de propriétés, mais il a aussi le statut d'objet standard, car rien dans le langage ne le distingue des autres objets. En particulier, il est possible d'envoyer des messages directement au `trait` afin d'invoquer les propriétés qu'il détient, et cela pose problème lorsque celles-ci contiennent des références à des variables. Par exemple, si on envoie le message `incr` à `compteur-trait`, l'accès à la variable `val` lèvera une exception puisque `compteur-trait` ne possède pas cette propriété. Un `trait` détient des propriétés qui lui sont applicables (on peut les activer par envoi de message puiqu'il les détient) mais pratiquement inapplicables (elles sont prévues pour être appliquées à ses extensions). Avec les représentants moyens, dans une moindre mesure, il peut se poser le même problème qu'avec les `traits` : un représentant moyen peut très bien être incomplet, c'est-à-dire détenir des méthodes faisant référence à des variables qu'ils ne possèdent pas, ou dont la valeur n'est pas définie (parce qu'il n'y a pas de valeur moyenne pour cette variable, par exemple).

Le clonage met bien en lumière les problèmes que pose le double statut des instances prototypiques, des traits ou des représentants moyens. Il est impossible d'écrire une primitive de clonage capable de cloner automatiquement et correctement un objet ayant pour parent (direct ou indirect) un tel objet. Par exemple, pour cloner le compteur représenté par les objets `compteur1` et `compteur-trait` (Fig. 8.11), il faut cloner `compteur1` mais pas `compteur-trait`. Plus généralement, le problème est de savoir, à partir d'un objet donné, jusqu'où la copie doit remonter afin d'obtenir une nouvelle entité sans dupliquer la bibliothèque de comportements, ce qui est impossible puisque rien ne distingue généralement la bibliothèque des autres objets[31].

Si l'on cumule les différents problèmes, cloner automatiquement une entité morcelée membre d'une famille de clones représentée par un `trait` supposerait de

31. Profitons de cette occasion pour discuter d'un cas particulier. YAFOOL solutionne ce dernier problème par certaines adaptations et restrictions. Les représentants moyens sont créés à l'aide d'une primitive spécifique (`defmodele`) et les objets sont dotés d'un attribut booléen indiquant s'ils ont ou non le statut de bibliothèque (ou d'abstraction). Par ailleurs il est impossible de créer des extensions d'objets non définis par `defmodele`. Ce booléen, tout en permettant de différencier les objets, met en lumière le problème du double statut. L'impossibilité de créer des extensions d'objets quelconques limite par ailleurs sérieusement les possibilités de représentations d'objets exceptionnels. YAFOOL est à la base un langage à prototypes et « au sommet », autre chose, assez proche d'un langage à classes sans en être vraiment un.

pouvoir déterminer d'une part quel est l'ensemble des objets représentant l'entité et d'autre part, parmi ceux-ci, lesquels sont concrets ou abstraits.

**Gestion automatique des familles de clones.**

Une seconde approche est l'automatisation. Dans le langage KEVO, tout objet appartient à une famille de clones automatiquement gérée par le système. Si une méthode est ajoutée ou retirée à un objet, celui-ci change de famille. SELF propose une construction appelée `map`, elle aussi gérée automatiquement et invisible au programmeur. Un `map` détient les méthodes d'un objet ainsi que pour chaque attribut, son nom et l'indice auquel la valeur est rangée dans l'objet. Les `maps` permettent d'obtenir une implantation mémoire des objets identique à celle des instances d'une classe ; ils réduisent l'espace mémoire occupé par chaque objet dans lesquels seules les valeurs des attributs sont stockées (Fig. 8.12, partie droite). Un nouveau `map` est créé à chaque création *ex nihilo* d'un nouvel objet et tous les membres d'une famille de clone dont la structure n'a pas été modifiée partagent le même `map`.

Ni les familles de clones automatisées de KEVO, ni les *maps* de SELF ne présentent l'inconvénient de conférer un double statut à des objets ; des constructions spécifiques sont utilisées pour assurer la factorisation. Toutefois, le programmeur n'a aucun contrôle sur ces constructions. Il ne peut spécifier quels objets doivent appartenir à la même famille, ni déterminer quels objets sont effectivement considérés par le système comme lui appartenant. Avec KEVO, lorsque l'on désire ajouter une propriété à toute une famille de clones, il suffit de choisir un objet membre de cette famille et d'invoquer une primitive d'ajout collectif, sans savoir avec précision quels objets vont être modifiés. En SELF, aucune primitive ne permet d'ajouter une propriété à tous les objets rattachés au même `map`.

**Conclusion**

La gestion des familles par des prototypes pose des problèmes conceptuels. Les trois solutions ont à la base le même défaut qui est d'utiliser des prototypes pour représenter des abstractions (les familles de clones), donc de réintroduire des formes d'abstraction sans support adéquat. La gestion automatique des familles de clones est insatisfaisante, elle ne fait que masquer le problème fondamental de l'absence d'objets du langage capables de représenter sans ambiguïtés des collections d'objets ou d'autres abstractions.

### 8.9.2 Problème du partage entre familles de clones

Une dernière forme de partage essentielle pour l'organisation des programmes a dû être introduite dans les langages à prototypes : il s'agit du partage entre différentes familles de clones. Si on prend l'exemple d'un programme représentant des « gaulois » et des « romains », le problème est de savoir sur quel objet définir les propriétés communes à ces deux familles d'entités comme par exemple la méthode `voyager`. Dans le monde des classes, ce type de partage est réalisé par des super-classes, souvent abstraites, communes à plusieurs classes. Dans notre exemple la méthode `voyager` serait définie sur une classe `Personnage` super-classe de `Romain`

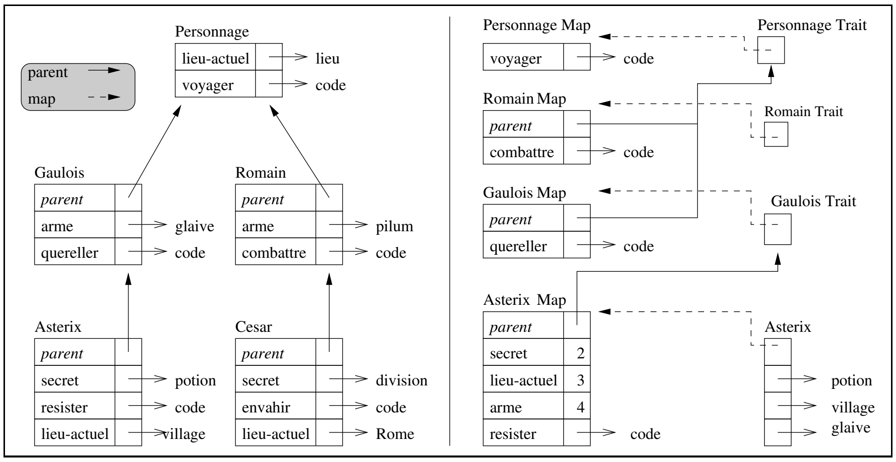



FIG. 8.12: *Hiérarchies réalisées avec des représentants moyens d'une part et des* `traits` *et des* `maps` *à la* SELF *d'autre part (d'après un exemple en* YAFOOL *de J. Quinqueton [Ducournau et Quinqueton, 1986]).*

et `Gaulois`. Dans les langages à prototypes, il a été mis en œuvre à l'aide des constructions que nous venons de décrire, instances prototypiques, représentants moyens, `traits` ou `maps`.

Utiliser la solution des instances prototypiques est à nouveau peu satisfaisant. En effet, il n'existe en principe aucun représentant typique de concepts abstraits (les concepts représentés par des classes abstraites dans les langages à classes), on ne peut alors en élever un au rang d'instance prototypique. Il est bien sûr possible de choisir un représentant d'un sous-concept comme représentant du concept abstrait, par exemple choisir un `gaulois`, que l'on doterait de la méthode `voyager`, comme instance prototypique de personnage, ce qui implique de définir tous les romains comme ses descendants. Ceci implique finalement une multiplication des masquages et des redéfinitions dans les descendants et rend les hiérarchies assez obscures.

La partie gauche de la figure 8.12 montre une hiérarchie de représentants moyens permettant de répondre au problème posé. Dans cet exemple, un représentant moyen du concept personnage détient les propriétés `lieu-actuel` et `voyager`. Cette utilisation des représentants moyens ne pose a priori pas de problèmes nouveaux.

La partie droite de la figure 8.12 montre un équivalent en SELF, les représentants moyens sont remplacés par des `traits` ne contenant que des méthodes. De plus chaque objet possède un `map`. Le lien `parent` d'un objet est stocké dans son `map`. La gestion automatique du partage entre familles pose de nouveaux problèmes. Dans KEVO, non seulement le système gère automatiquement les familles de clones, mais il doit les organiser en une hiérarchie. La maintenance automatique de cette hiérarchie est basée sur un ensemble de pré- et post-conditions à la modification d'un objet, ou de toute une famille. Lorsqu'un objet est modifié, il doit changer de famille : le système doit déterminer quelle est cette famille et éventuellement la créer. Dans le cas d'une modification collective, le problème ne consiste plus à faire migrer un seul objet, mais une famille entière : le système vérifie donc si la famille modifiée doit ou non fusionner avec l'une de ses voisines dans la hiérarchie. En ce qui concerne les `maps` de SELF, le même problème de migration se pose, mais aucune solution spécifique n'a été mise en œuvre pour le résoudre. Plusieurs `maps` redondants peuvent donc cohabiter à la suite de modifications équivalentes appliquées à des objets différents. Deux objets ayant même structure et même comportement ne sont donc pas systématiquement considérés comme appartenant à la même famille de clones.

Les méthodes des instances prototypiques et la gestion automatisée des familles supportent donc mal la réalisation de véritables hiérarchies d'héritage, incluant des niveaux abstraits capables de factoriser des propriétés communes à diverses familles d'objets.

## 8.10 Conclusion

Ce chapitre présente la genèse, les caractéristiques, les potentialités et les problèmes posés par les langages à prototypes. Nous y avons présenté la notion de prototype, telle qu'elle a été définie en sciences cognitives, montré comment cette notion a d'abord été utilisée en représentation de connaissances et en programmation distribuée. Nous avons enfin montré comment ces utilisations ont été intégrées au monde de la programmation par objets, pour former la famille des langages à prototypes. Les caractéristiques primitives de ces langages ont ainsi été décrites en en respectant la diversité.

Nous avons ensuite proposé des caractérisations du clonage, du mécanisme d'extension et de la différence entre l'héritage dans les hiérarchies d'objet et celui dans les hiérarchies de classes. Ces caractérisation nous a permis d'expliquer les différentes évolutions proposées par les langages, les différentes utilisations possibles de la délégation ainsi que de mieux appréhender les avantages et les problèmes que pose la programmation basée sur la délégation et les objets sans classes.

Quels constats peut-on tirer de cette étude?

Les langages à prototypes ont permis de découvrir ou de redécouvrir des mécanismes peu utilisés comme le clonage ou l'héritage entre objets (la délégation). Les objets sans classes, moins contraints, permettent ou simplifient la représentation d'un nombre important de situations. La délégation doit être considérée indépendamment des langages dans lesquels elle est utilisée. C'est un mécanisme utilisant un héritage entre objets ayant ses spécificités et ses applications propres, différentes des applications de l'héritage entre classes.

Une première grande série de problèmes de la programmation par prototypes est liée à la disparition de la possibilité de description en compréhension des concepts. Cette disparition se manifeste dès que l'on souhaite partager des propriétés communes à des familles d'objets. Tous les langages à prototypes ont réintroduits des objets ayant en plus du statut de membre d'une famille, celui de représentant d'une famille d'objets (les instances prototypique des OBJECT-LISP, les représentants moyens de YAFOOL ou les `traits` de SELF), ou celui de descripteur d'objets (les `maps` de SELF ou les familles de clone de KEVO), ou encore celui de bibliothèque de comportements (les `traits` ou les représentants moyens) ou plusieurs de ces statuts à la fois.

Une seconde série de problèmes est relative à des utilisations non fondées du mécanisme de délégation. La relation support au mécanisme est une relation entre objets, que l'on peut nommer «est-une-extension-de» ; un objet est en relation avec un autre, son modèle, lorsqu'il est créé par une copie différentielle, c'est-à-dire en exprimant ses différences par rapport au modèle. La délégation est un mécanisme d'héritage qui induit un partage des attributs ou des valeurs de ces attributs entre objets. Ces formes de partage, absentes en programmation par classes, sont à la base de possibilités originales de représentation. Elles sont aussi la cause de problèmes d'inter-dépendance entre objets, ou de gestion d'entités représentés par des ensembles d'objets.

Ces derniers surviennent lorsque la relation «est-une-extension-de» est utilisé pour partager des attributs entre objets représentant des entités différentes, ce qui fait voler en éclat la notion d'encapsulation. Il existe aux moins deux interprétations fondées de cette relation. La première interprétation consiste à dire qu'une extension $o2$ d'un objet $o1$ représentant une entité $E1$, représente une entité $E2$ distincte de $E1$ mais ayant par défaut des caractéristique communes. Au niveau de la représentation, ces caractéristiques sont rangées dans $o1$ et héritées donc partagées par $o2$. Nous parlons de sémantique de partage de valeur. Cette interprétation est utilisée dans divers langages comme YAFOOL ou GARNET. La seconde interprétation consiste à utiliser la description différentielle pour exprimer un partage de propriétés entre les différentes parties d'une entité. Une application en est est la représentation, par des «objets morcelés», de divers points de vue d'une même entité avec partage possible entre morceaux représentant ces points de vues (`Jean` le sportif et le même `Jean`, professeur, ont le même age) (cf. section 8.4.3).

L'héritage entre objets (fondé sur la description différentielle des objets) et l'héritage entre classes (fondé sur la description différentielle de concepts) sont complémentaires. La délégation n'est pas un mécanisme spécifique de la programmation par prototypes. Elle doit donc être considérée indépendamment des langages dans lesquels elle est utilisée. C'est un mécanisme ayant ses spécificités et ses applications propres ; il peut être mis en œuvre dans un monde ou il y a des classes. Nous avons ainsi évoqué le problème de la représentation d'entités avec points de vues.

Pour tirer un constat plus général de cette étude, reconsidérons les motivations initiales ayant amené le développement des langages à prototypes. En ce qui concerne la description simplifiée des objets et les possibilités de représentation, le constat est favorable. Si un certain nombre de langages ont été et continuent à être développés, c'est parce qu'ils répondent à des besoins. Le constat est moins favorable en ce qui concerne la simplicité du modèle de programmation. Sur ce point, l'argumentation en faveur des langages à prototypes était fondée sur le fait que le modèle des prototypes est un modèle de programmation minimaliste dans lequel tout concept jugé inutilement complexe a été supprimé[32]. La réduction du nombre et la simplicité des concepts de base ne s'est pas avérée être un facteur de plus grande simplicité de programmation. (les modèles les plus minimalistes que l'on aie pu concevoir, la machine de Turing par exemple, ne sont d'ailleurs pas ceux dans lesquels il est le plus aisé de programmer). Le monde des classes est plus rigide, celui des prototypes offre plus de liberté mais ne demande pas moins d'efforts de compréhension. Le développement d'applications importantes requiert manifestement la possibilité de décrire des abstractions. A l'heure actuelle aucun des langages à prototypes existants ne le permet de façon réellement satisfaisante.

Les problèmes relatifs à l'organisation des programmes condamnent-ils la programmation par prototypes? Si l'on réduit ce courant à des langages manipulant uniquement des « objets concrets » la réponse est certainement positive. Mais cette réduction serait une erreur. Les langages à prototypes existants, avec ou sans les solutions apportées aux problèmes d'intégrité et de partage, ont permis l'implantation de logiciels importants, on peut penser à l'environnement SELF [Agesen *et al.*, 1995], à la construction d'interfaces graphiques [Myers *et al.*, 1992] en GARNET ou AMULET, à l'architecture logicielle du *Newton* d'*Apple*). Ils ont également été utilisés comme couche basse (ou assembleurs de haut niveau) d'autres langages à objets: les langages YAFOOL ou OBJECT-LISP sont à la base des langages à prototypes mais disposent d'une couche logicielle permettant au programmeur de penser jusqu'à une certaine limite en termes de classes et d'instances[33]. Des applications importantes ont été réalisées en YAFOOL (par exemple RESYN, *chapitre 14*).

En résumé, si les langages à prototypes présentent encore des défauts, les motivations qui ont conduit à leur développement sont toujours d'actualité, les recherches les concernant restent assez peu nombreuses. Il nous semble maintenant intéressant de travailler à une réconciliation entre prototypes et abstractions, entre la souplesse de représentation qu'offrent les objets sans classes et les possibilités de structuration offertes par des objets abstraits assumant pleinement leur statut. Une des pistes à suivre pour parvenir à ce résultat est d'arriver à définir des descripteurs d'objets et des bibliothèques de comportements qui soient eux-mêmes des objets capables de recevoir des messages, sans pour autant contraindre les objets autant que ne le font les classes.

32. En parlant de la conception du langage SELF : « *We employed a minimalist strategy, striving to distill an essence of object and message* » [Smith et Ungar, 1995].

33. Dans les deux cas, ces couches logicielles ne transforment pas ces langages en véritables langages à classes, le partage de valeur d'attributs et ses conséquences étant toujours présent.

# 9

# Objets et contraintes

OBJETS ET CONTRAINTES sont deux notions informatiques qui connaissent chacune depuis vingt ans un essor considérable. La notion d'*objet* est centrale aux Langages de Programmation par Objets (LPO, voir *chapitres 1, 2 et 3*), aux Systèmes de Représentation de Connaissances par Objets (SRCO, voir *chapitres 10, 11 et 12*) et aux bases de données orientées objet (voir *chapitre 5*) [1]. Bien que cette notion ait dans ces langages et ces systèmes des interprétations, utilisations et prérogatives distinctes, le consensus se fait autour de l'intérêt de disposer de structures de données — les objets (ou, plus exactement, les classes) —, organisées de manière hiérarchique afin de factoriser des connaissances et de pouvoir en hériter.

La notion de *contrainte* est centrale à la *satisfaction de contraintes*, domaine de l'intelligence artificielle dont les bases théoriques ont vu le jour au milieu des années soixante-dix [Mackworth, 1977 ; Montanari, 1974]. S'appuyant sur des techniques élaborées par cette théorie, mais également par d'autres domaines comme l'analyse numérique ou la recherche opérationnelle, les *Langages de Programmation Par Contraintes* (LPC) sont des outils informatiques de description et de résolution de problèmes qui ont aujourd'hui un grand succès industriel. Ils sont notamment utilisés par les entreprises confrontées à des problèmes d'allocations de ressources, de gestion de production ou d'emplois du temps pour lesquels ils offrent une (idée de) réponse [Fron, 1994].

Plusieurs réalisations témoignent que les développements respectifs des deux paradigmes *objet* et *contrainte* ne se sont pas faits sans tentatives d'apporter à l'un les avantages de l'autre. Du côté des objets, il est tentant de munir un LPO ou un SRCO de capacités de description et de résolution de problèmes en y intégrant des contraintes. Du côté des contraintes, il est tentant de chercher à représenter, organiser, factoriser et contrôler à l'aide d'objets, la description et la résolution d'un problème de satisfaction de contraintes dans un langage ou un module de PPC. Poussant la combinaison à l'extrême, il est tentant de faire cohabiter dans un LPO ou dans un SRCO des objets qui représentent des contraintes et des objets sur lesquels portent des contraintes décrites en termes d'objets.

1. Nous n'aborderons pas ici le domaine des bases de données orientées objet.

Ce chapitre a deux objectifs. Premièrement, nous dressons un panorama des principaux langages et modèles qui proposent une association objets/contraintes en rappelant leurs motivations, leurs particularités et leurs différences. Deuxièmement, nous nous intéressons plus particulièrement à l'association objets/contraintes en représentation de connaissances.

Nous débutons par une présentation des principaux objectifs, définitions et techniques de ce domaine de recherche (§ 9.1). Les choix que nous faisons dans cette présentation visent à ne retenir, parmi les nombreux aspects de la programmation par contraintes, que ceux qui sont les plus fréquemment rencontrés dans les systèmes présentant une association objets/contraintes. Nous abordons alors le premier point de cette étude en proposant une classification des associations objets/contraintes. Nous choisissons de distinguer d'un côté les LPO étendus par des contraintes et, de l'autre, les langages de PPC intégrant la notion d'objet (§ 9.2). Le second point de l'étude s'inscrit dans le domaine de la représentation de connaissances (§ 9.3). Nous nous intéressons tout d'abord aux tentatives d'introduction de contraintes dans les logiques de descriptions (voir *chapitre 11*) qui reposent sur une approche formelle de la représentation hiérarchique des connaissances d'un domaine (§ 9.3.1). Puis, nous proposons une étude des tenants et des aboutissants de l'intégration de contraintes dans une représentation à objets (§ 9.3.2). Nous tentons de répondre alors aux questions : contraindre quoi ? contraindre comment ? contraindre et ensuite ? Ce faisant, nous essayons de cerner les apports et les limites d'une association entre objets et contraintes au sein d'une représentation par objets.

## 9.1 Contraintes

Du point de vue de l'utilisateur, les LPC sont devenus populaires pour leur déclarativité et leur généricité. D'une part, ils permettent à un utilisateur de décrire de manière assez intuitive (et proche du formalisme mathématique) un problème en termes d'expressions de contraintes construites à partir de la combinaison d'opérateurs de comparaison et d'opérateurs algébriques mis à sa disposition ou, parfois, créés par lui-même. D'autre part, ils offrent un ensemble de techniques de recherche de solutions que l'utilisateur peut éventuellement contrôler ou, parfois, définir (programmer). En quête de généricité, mais aussi d'efficacité dans des classes de problèmes bien déterminées, les LPC tendent de plus en plus à intégrer des techniques de résolution éprouvées de divers domaines : analyse numérique, théorie des graphes, Recherche Opérationnelle (RO) . . . Par exemple, des algorithmes ou heuristiques sophistiqués, issus de la RO, vont permettre à un langage de PPC d'être adapté à la résolution de problèmes d'ordonnancement de tâches. En ce sens, la PPC peut être vue comme une vitrine informatique et non pas comme une concurrente de la RO : les deux visent à proposer des méthodes de résolution de problèmes. La PPC est plus générique, plus flexible et donc moins performante : elle ne peut s'attaquer qu'à des problèmes de taille raisonnable. La RO, quant à elle, s'attaque, souvent par des heuristiques, à des problèmes plus spécifiques et surtout, de plus grosse taille, mais les bibliothèques de programmes qu'elle propose apparaissent comme peu flexibles et hermétiques (boîtes noires), au contraire des outils de PPC.

À la base, les LPC intègrent de nombreux résultats propres à la théorie des Problèmes de Satisfaction de Contraintes (ou CSP [2]). Cette théorie s'attache tout d'abord à identifier et à définir divers types de CSP qui sont alors autant de sujets d'études. Pour chaque type de CSP, l'objectif est de caractériser et de formaliser ses propriétés à partir d'un choix de représentation, de l'étude des graphes associés aux réseaux de contraintes ou de la nature même des contraintes, afin de proposer des techniques de filtrage de l'espace de recherche de solutions ou/et des algorithmes de résolution performants [Tsang, 1993].

### 9.1.1 CSP : définitions

**Définition 1 (d'après [Montanari, 1974 ; Mackworth, 1977])** *Un Problème de Satisfaction de Contraintes, ou* CSP*, est défini par la donnée d'un triplet* ($X$, $D$, $C$) *où :*
$X = \{x_1, \ldots, x_n\}$ *est un ensemble fini de variables,*
$D = \{d_1, \ldots, d_n\}$ *est l'ensemble fini des domaines de valeurs des variables de* $X$ *(à chaque variable* $x_i$ *de* $X$ *est associé un domaine* $d_i$ *de* $D$*),*
$C = \{c_1, \ldots, c_m\}$ *est un ensemble fini de contraintes. Chaque contrainte* $c_i$ *de* $C$ *porte sur un sous ensemble* $\{x_{i1}, \ldots, x_{ik}\}$ *de* $X$ *et définit un sous-ensemble du produit cartésien* $d_{i1} \times d_{i2} \times \ldots \times d_{ik}$.

Chaque contrainte $c_i$ de $C$ peut être décrite en *extension* comme l'ensemble des $k$-uplets $(v_{i1}, \ldots, v_{ik})$ — une valeur $v_{ij}$ pour chaque variable $x_{ij}$ de $c_i$ — qui satisfont la contrainte $c_i$, ou en *intension* sous la forme d'une relation décrite à l'aide d'opérateurs (en général un opérateur de comparaison et des opérateurs algébriques) définis sur le type des variables de $c_i$.

Par exemple, $c_i$ peut être donnée en extension par l'ensemble $\{(2, 3, 5), (2, 1, 3), (2, 2, 4)\}$ ou bien en intension par la relation $x_{i1} + x_{i2} = x_{i3}$.

Le but de la satisfaction d'un CSP est de déterminer les solutions de ce problème, si elles existent. On peut se contenter d'une seule solution ou les rechercher toutes, ou encore chercher la meilleure si l'on dispose d'une fonction d'évaluation du coût d'une solution. La notion de base liée à la recherche de solution est l'*instanciation* (partielle ou totale) qui correspond à l'affectation à un ensemble de variables $Y$ (avec $Y \subseteq X$) de valeurs de leurs domaines respectifs. Dans la recherche d'une solution, on cherche à produire des *instanciations localement consistantes*.

**Définition 2 (d'après [Dechter, 1992])** *Par rapport à un* CSP $(X, D, C)$*, une instanciation partielle d'un ensemble de variables* $Y$ *est dite localement consistante si elle satisfait toute contrainte* $c_i(x_{i1}, \ldots, x_{ik})$ *avec* $\{x_{i1}, \ldots, x_{ik}\} \subseteq Y \subseteq X$.

Dès lors, une *solution* est définie par :

**Définition 3 (d'après [Dechter, 1992])** *Une solution du* CSP ($X$, $D$, $C$) *est une instanciation de* $X$ *localement consistante. Cette instanciation est alors dite globalement consistante.*

2. Le terme anglais de *Constraint Satisfaction Problems* est conservé.

### 9.1.2 Techniques de résolution

Le problème du $k$-coloriage d'un graphe, connu comme NP-complet, pouvant s'exprimer comme un CSP, la détermination des solutions d'un CSP est un problème par nature NP-complet. La résolution d'un CSP demande donc *a priori* un temps exponentiel en la taille des données du problème. Mais, en fonction des spécificités du CSP (nombre de solutions recherchées, structure du graphe pour les CSP binaires ... ), il est possible de réduire sensiblement la complexité de sa résolution.

Parmi les méthodes de résolution, il faut distinguer les techniques d'énumération systématiques des domaines et les méthodes stochastiques. Les premières cherchent à étendre progressivement (en choisissant une valeur dans le domaine de la variable en cours d'examen) des instanciations partielles localement consistantes vers des instanciations globalement consistantes. Les secondes partent d'une instanciation totale déterminée au hasard et non forcément globalement consistante et cherchent à atteindre une solution (qu'elles n'atteindront peut être pas) [Freuder *et al.*, 1995].

Les techniques d'énumération systématiques comprennent :

- les méthodes à base de retour-arrière (*Backtracking*, *Backjumping*, *Backmarking* ... ) [Gaschnig, 1979] qui, en cas d'échec, remettent en cause une variable précédemment instanciée. Elles se distinguent par leur pertinence dans le choix de la variable à remettre en cause, c'est-à-dire par leur capacité à éviter que des affectations ayant déjà mené à un échec ne se reproduisent, en stockant des informations relatives aux échecs.
- les méthodes à base de recherche en avant (*Forward-Checking*, *(Full-)Look-Ahead* ... ) [Haralick et Elliot, 1980] qui cherchent à propager les conséquences de l'instanciation courante en éliminant des domaines des variables non encore instanciées, les valeurs d'ores et déjà incompatibles avec la valeur choisie pour la variable courante. Elles se distinguent par la puissance du filtrage qu'elles opèrent sur les domaines des variables non encore instanciées.

Il est possible d'accompagner ces méthodes d'énumération des domaines par un choix statique ou dynamique de la prochaine variable et/ou valeur à considérer. En général, une combinaison de toutes ces approches donne les meilleurs résultats [Ginsberg, 1993 ; Frost et Dechter, 1994]. Ces méthodes permettent de décider si un CSP peut être résolu ou non.

Les méthodes stochastiques, elles, connaissent un succès grandissant depuis 1990. Elles regroupent diverses techniques ou heuristiques issues en majorité de la recherche opérationnelle, telles que le *Hill-Climbing*, les méthodes connectionnistes, les algorithmes génétiques, le recuit simulé ou la méthode tabou. Si ces méthodes ne peuvent décider de la *satisfiabilité* ou de la *non satisfiabilité* d'un CSP (existe-t-il ou non une solution), elles résolvent des CSP dans des applications de taille réelle ou peuvent s'approcher d'une solution, ce que ne savent pas faire les méthodes exhaustives d'énumération [Hao *et al.*, 1997].

La tendance actuelle des recherches en CSP est d'instiller des techniques stochastiques dans les méthodes systématiques pour parvenir plus rapidement à une solution.

### 9.1.3 Maintien de solution

Le maintien de solution consiste, lors de la modification de la valeur d'un ensemble de variables et/ou de l'ajout d'un ensemble de contraintes, à déterminer une façon de parvenir à une nouvelle solution en modifiant les valeurs d'autres variables. Cette technique est notamment employée dans les systèmes ayant à répondre à une perturbation d'un état stable, comme ceux gérant des interfaces graphiques contraintes. Le choix des variables à modifier peut être fixé dès la définition de la contrainte, comme c'est le cas dans le système THINGLAB (§ 9.2.1), ou bien laissé à l'utilisateur, en exhibant toutefois les graphes de dépendances et les différents flots de propagation des modifications possibles, comme c'est le cas dans le système PROSE (§ 9.2.3).

### 9.1.4 Maintien de consistance

Les techniques systématiques d'énumération sont applicables sur le CSP à l'état brut (directement à partir de sa définition) mais il est souvent plus judicieux de chercher à réduire la taille de l'espace de recherche de solutions. Cette phase de réduction des domaines, appelée *maintien de consistance locale*, consiste à éliminer des domaines, les valeurs qui ne peuvent apparaître dans une solution. Il s'agit donc d'un *filtrage* de domaines. Le niveau de consistance atteint dépend de la procédure de maintien de consistance employée. La plus populaire de ces procédures en raison de son coût polynômial en la taille des domaines et en le nombre de contraintes du CSP est l'*arc-consistance* :

**Définition 4** *Un* CSP *est* **arc-consistant** *ssi :*
$\forall x_i \in X$, $\forall v_i \in d_i$
$\forall c_k(x_{k1}, \ldots, x_{kp}) \in C$, *telle que* $x_i \in \{x_{k1}, \ldots, x_{kp}\}$
$\exists v_{k1} \in d_{k1}, \ldots, \exists v_{kp} \in d_{kp}$, *telles que* $(v_{k1}, \ldots, v_i, \ldots, v_{kp}) \in c_k$

En général, l'arc-consistance d'un CSP est établie par une *phase de propagation* incrémentale qui consiste, lors de la pose d'une contrainte du CSP, à éliminer des domaines les valeurs non viables (celles qui, pour au moins une contrainte, ne figurent dans aucune des instanciations localement consistantes), et à tester pour toute contrainte déjà posée qui porte sur une variable dont le domaine vient d'être réduit, la viabilité des valeurs de ses autres variables. Ceci parce que la suppression d'une valeur dans un domaine peut entraîner la suppression d'une autre valeur qu'elle accompagnait — on dit alors qu'elle en est le *support* — jusque là dans une instanciation localement consistante. La propagation s'arrête lorsqu'aucune contrainte n'est plus à propager, les domaines ont alors un contenu stable et arc-consistant.

### 9.1.5 Restriction et relaxation incrémentales

Les CSP peuvent être définis d'emblée par la donnée du triplet $(X, D, C)$ mais également de manière incrémentale en ajoutant et en retirant au fur et à mesure des contraintes. On parle alors de CSP *dynamiques*. L'ajout d'une contrainte correspond à une *restriction* potentielle de l'ensemble des solutions ou/et des domaines du CSP, s'il y en a. Le retrait d'une contrainte, quant à lui, correspond à une *relaxation*, c'est-à-dire un éventuel élargissement de l'ensemble des solutions ou/et des domaines du CSP. L'ajout d'une contrainte déclenche une *propagation* qui correspond à une version incrémentale du maintien de la consistance[3]. On propage au CSP tout entier les possibles réductions de domaine occasionnées par la contrainte ajoutée. Si cette propagation vide un domaine alors le CSP est inconsistant : la contrainte ajoutée est trop forte et ne peut être acceptée. La relaxation, au contraire, ne peut mener à une impasse. Il s'agit de faire passer le CSP diminué d'une contrainte dans un autre état stable. Ce retrait peut être abordé de deux manières : soit en s'appuyant sur les domaines initiaux des variables, on pose à nouveau toutes les contraintes sauf celle que l'on désire supprimer, soit on cherche à atteindre cet état sans avoir pour autant à tout reconstruire. La première solution, radicale, peut s'avérer coûteuse en terme de propagation de contraintes selon le nombre de contraintes. La seconde solution est la moins coûteuse en terme de propagation (cela dépend de l'âge de la contrainte). Mais elle est la plus coûteuse en terme de mémorisation, puisqu'elle impose de sauvegarder pour chaque contrainte son contexte de pose, c'est-à-dire une photographie des domaines contraints du réseau au moment de la pose de la contrainte, ou bien de gérer une pile des différents états du CSP. Il s'agit ici de supprimer la contrainte du réseau, de restaurer son contexte de pose et de ne propager que les contraintes plus jeunes du même réseau, c'est-à-dire posées après.

### 9.1.6 Divers types de CSP

Les définitions et techniques précédentes s'appliquent à des CSP à domaines finis, qui sont les cas d'étude privilégiés de la théorie des CSP et auxquels se limitaient les premiers LPC. Avec l'ambition de s'attaquer à des problèmes réels, on observe depuis les années 1990, l'émergence de LPC capables de gérer et résoudre des CSP numériques à domaines continus ou infinis, des CSP booléens, des CSP symboliques, ou des CSP à objets.

#### ICSP

Les CSP *numériques*, à domaines finis ou non, continus ou non, peuvent être décrits en termes de CSP *à intervalles* (ou ICSP) [Hyvönen, 1992]). Un ICSP est un CSP dans lequel les domaines sont donnés sous la forme d'un intervalle unique ou bien d'une union d'intervalles, les variables étant numériques (entières ou réelles). Cette représentation a l'avantage de la compacité et permet également de raisonner sur des intervalles de variation ou sur des ensembles infinis de valeurs, continus ou discrets, ce qui la rend bien adaptée à la modélisation de problèmes réels. La gestion de ces CSP s'appuie sur une décomposition des contraintes (dites complexes) du problème en contraintes primitives gérées par le système et auxquelles sont associées des règles de consistance particulières. Dans l'exemple qui suit, les

3. La résolution d'un CSP dynamique n'est pas abordée ici, on pourra consulter [Verfaillie et Schiex, 1994].

contraintes complexes de gauche sont transformées en les contraintes primitives de droite :

$$\begin{cases} cc_1: & Z = X * Y \\ cc_2: & e^X + Z > 12.2 \end{cases} \iff \begin{cases} cp_1: & X_1 = X * Y \\ cp_2: & X_1 = Z \\ cp_3: & e^X = X_2 \\ cp_4: & X_2 + Z = X_3 \\ cp_5: & X_3 > 12.2 \end{cases}$$

Les règles de consistance qui calculent les domaines résultant de l'application d'une contrainte s'appuient sur l'arithmétique des intervalles [Moore, 1966]. Les plus simples de ces règles sont :

$$[a, b] + [c, d] = [a + c, b + d]$$
$$[a, b] - [c, d] = [a - d, b - c]$$
$$[a, b] * [c, d] = [min(a * c, a * d, b * c, b * d), max(a * c, a * d, b * c, b * d)]$$
$$[a, b] \,/\, [c, d] = [a, b] * [1/d, 1/c], \text{ si } 0 \notin [c, d]$$

Elles sont appliquées aux arguments des contraintes primitives qui forment l'ICSP afin de déterminer leur nouveau domaine. Soit la contrainte primitive, $Z = X + Y$ où $dom(X) = [a, b]$, $dom(Y) = [c, d]$ et $dom(Z) = [e, f]$. Après application des règles de maintien de consistance associées aux opérateurs, les domaines deviennent :

$$dom(Z) = [e', f'] = [a + c, b + d] \cap [e, f]$$
$$dom(X) = [a', b'] = ([e', f'] - [c, d]) \cap [a, b]$$
$$dom(Y) = [c', d'] = ([e', f'] - [a', b']) \cap [c, d]$$

En raison de la manipulation d'intervalles de réels (plus exactement de flottants), le niveau de consistance dans les ICSP ne peut pas atteindre dans le cas général l'arc-consistance mais s'en rapproche plus ou moins à travers les notions de *boîte-consistance* et d'*enveloppe-consistance* pour les CSP à intervalles uniques, d'*intervalle-consistance* pour les CSP à union d'intervalles [Benhamou *et al.*, 1994]. Les recherches sur les ICSP s'orientent à la fois vers la détection de cycles (lorsque 2 contraintes partagent au moins 2 variables) qui occasionnent des problèmes de terminaison lors de la propagation, vers l'application de techniques algébriques de réécriture qui visent à réduire la taille des domaines [Alander, 1985], et vers la résolution par l'étude des propriétés du graphe associé à l'ICSP [Hyvönen, 1992] ou bien par des techniques numériques telles que la méthode de Newton [Benhamou *et al.*, 1994] qui a l'intéressante propriété de converger sur les intervalles.

### CSP booléens

Les CSP *booléens* trouvent un écho surtout dans les applications relevant de la conception de circuits électroniques ou du domaine du calcul propositionnel. Les contraintes sont ici construites à partir d'opérateurs booléens. Ces CSP sont gérés par des algorithmes d'unification booléenne [Buttner et Simonis, 1987] ou bien par des méthodes numériques (bases de Gröbner).

### CSP symboliques

Les CSP *symboliques* sont des CSP pour lesquels des contraintes génériques sont disponibles (par exemple, la contrainte “plus foncé” entre deux variables dont le domaine est un ensemble de couleurs désignées par leur nom). Par extension, on y classe également les CSP *ensemblistes* qui portent sur des variables dont les valeurs sont des ensembles ou des listes de valeurs. Ces contraintes sont construites à partir d'opérateurs définis sur les listes et les ensembles. Elles permettent d'imposer que des éléments d'une liste ou d'un ensemble soient tous différents, qu'un élément appartienne à un ensemble ou à une liste, etc. L'efficacité des techniques de consistance et de résolution sur ce type de CSP dépend de la représentation choisie pour les ensembles et les listes. Souvent, on conserve pour une variable ensembliste, l'intersection et l'union des ensembles qui forment les valeurs du domaine. Ces CSP ont leur utilité dans les problèmes d'optimisation combinatoire, d'affectation de ressources et, plus généralement, dès que des symétries ou des permutations sont décelables dans les solutions [Gervet, 1997 ; Legeard *et al.*, 1993].

### CSP à objets

L'idée des CSP *à objets* est de faire porter des contraintes sur des structures de données complexes, appelées objets, auxquels sont associés une sémantique et un comportement particulier. Le reste du chapitre est entièrement consacré à l'étude des diverses associations entre contraintes et objets, aussi bien dans le domaine de la programmation par objets que dans celui de la représentation de connaissances par objets.

## 9.2 Contraintes et objets pour la programmation

Nous faisons ici une revue des principaux systèmes alliant objets et contraintes dans le domaine de la programmation. On peut mettre en évidence trois approches en étudiant les divers exemples d'association d'objets et de contraintes en programmation :

1. les extensions de langages de programmation par objets existants avec des contraintes : THINGLAB (§ 9.2.1), SOLVER (§ 9.2.2) . . .
2. les langages de programmation intégrant à la fois les principes de la programmation par objets et ceux de la programmation par contraintes : KALEIDOSCOPE (§ 9.2.1) . . .
3. les boîtes à outils de programmation par contraintes exploitant les principes de la programmation par objets : CSPOO (§ 9.2.3), PROSE (§ 9.2.3) . . .

Les deux premières approches ont clairement l'objectif de contraindre des objets alors que la troisième s'attache davantage à tirer parti d'une définition et d'une exploitation des mécanismes associés à la programmation par contraintes à travers les standards de la programmation par objets (représentation classe/instance, hiérarchies, abstraction, méthodes . . . ).

Dans le premier cas, ce sont les atouts de la programmation par contraintes — déclarativité et puissance de raisonnement — qu'il s'agit d'exploiter dans un environnement d'objets. L'idée est d'établir des relations (contraintes) entre les objets et d'assurer le maintien de la consistance et/ou la résolution de ces liens par un moteur de contraintes, c'est-à-dire un ensemble d'algorithmes empruntés à la programmation par contraintes.

Dans le second cas, ce sont les atouts de la programmation par objets — organisation hiérarchique des objets, abstraction de données — qui sont sollicités pour la définition de langages ou de boîtes à outils de la programmation par contraintes. L'idée est de définir les contraintes en termes de classes et d'instances et de leur associer, en tant qu'objets de programmation, des méthodes (de consistance ou de résolution). Autrement dit, les langages (ou bibliothèques) de contraintes se dotent avec les objets d'un outil de représentation, d'organisation et de factorisation de connaissances.

À travers ces deux visions de l'alliance objets/contraintes, il apparaît que chacun des deux paradigmes peut tirer profit de l'autre. Loin d'être opposées, ces deux visions deviennent complémentaires dans certains systèmes où, non seulement les objets peuvent être contraints, mais les contraintes sont également des objets. C'est le cas dans THINGLAB mais ça peut l'être aussi pour tout langage (ou bibliothèque) de contraintes (voir la troisième approche) couplé avec un langage à objets.

### 9.2.1 Les langages à objets avec contraintes

#### Thinglab I et II

THINGLAB [Borning, 1981] est une extension du langage SMALLTALK [Ingalls, 1978] dans lequel contraintes, hiérarchies d'objets composites et héritage sont combinés afin d'offrir un atelier de simulation de modèles dynamiques, notamment en géométrie ou en physique. L'attrait de THINGLAB est une interface graphique interactive qui permet à l'utilisateur de construire des objets, de les visualiser sous plusieurs formes, de les modifier ou de les déplacer à l'aide d'une souris. Sous cet angle, il s'inspire principalement des systèmes SKETCHPAD [Sutherland, 1963] et CONSTRAINTS [Sussman et Jr., 1980], ancêtres des LPC.

Un objet THINGLAB (classe, instance ou prototype) est composé d'autres objets — eux mêmes composés ou primitifs — qui forment ses *parties* (voir tableau 9.1). Chaque partie est décrite par un nom, sa classe d'appartenance, un ensemble de contraintes et un ensemble de *fusions* qui visent à associer des objets de même classe. Une contrainte est décrite par une *règle* (traduite en un prédicat qui indique si la contrainte est satisfaite ou non) et une liste de méthodes ordonnées par préférence. Une contrainte définie pour une classe est appliquée à toutes les instances de cette classe. Le long des liens de spécialisation, une sous-classe hérite des contraintes de ses super-classes. La règle exprime une relation entre des parties ou sous-parties atteignables à partir de l'objet par un *chemin* le long du graphe de composition. Dès qu'une contrainte est définie en THINGLAB, elle doit être satisfaite. Le maintien de la consistance se fait à l'aide d'une propagation locale qui consiste à choisir pour la contrainte à satisfaire, une de ces méthodes (la première ou celle qui peut être

| **Class MidPointLine** | **Class Quadrilateral** |
|---|---|
| **Superclasses** | **Superclasses** |
| *Geometric Object* | *Geometric Object* |
| **Part Descriptions** | **Part Descriptions** |
| *line: a Line* | *part1: a Line* |
| *midpoint: a Point* | *part2: a Line* |
| **Constraints** | *part3: a Line* |
| *midpoint = (line point1 + line point2) / 2* | *part4: a Line* |
| *midpoint ← (line point1 + line point2) / 2* | **Merges** |
| *line point1 ← (midpoint * 2 - line point2)* | *part2 point2 ≡ part3 point1* |
| *line point2 ← (midpoint * 2 - line point1)* | *part1 point1 ≡ part4 point2* |
| | *part3 point2 ≡ part4 point1* |
| | *part1 point2 ≡ part2 point1* |

TAB. 9.1: *Description de deux classes* THINGLAB. *À gauche, la classe décrit le milieu d'une ligne ; la partie* `Constraints` *donne la règle et les méthodes assurant que le point est toujours au milieu de la ligne. À droite, la partie* `Merges` *décrit la fusion entre les sommets d'un quadrilatère.*

activée). La satisfaction des contraintes en THINGLAB est lancée à la modification d'un objet de l'interface. Une phase de planification regroupe toutes les contraintes concernées par la modification. Un plan de méthodes pour l'ajustement des valeurs des objets contraints est élaboré, chaque méthode de ce plan étant compilée. Puis, chaque méthode compilée est exécutée avant d'être rangée dans un dictionnaire afin d'être réutilisable.

THINGLAB a été principalement utilisé dans des animations graphiques (ANIMUS [Borning et Duisberg, 1986]) à base de contraintes temporelles. On peut lui reprocher de ne traiter que des contraintes fonctionnelles [4] (à base d'égalité ou de fusions) et de s'appuyer seulement sur la propagation pour le maintien de solution , ce qui le rend fragile en cas de réseaux de contraintes circulaires malgré les méthodes de relaxation ou d'introduction de points de vue redondants que Borning propose pour pallier cet inconvénient. Dans ce cas, un simple *retour-arrière* pourrait résoudre le problème et réajuster les valeurs des objets contraints. De plus, les contraintes sont définies à l'intérieur des classes : la même contrainte employée dans deux classes différentes (non liées par spécialisation directe ou indirecte) doit être décrite (règle + méthodes) deux fois.

Dans THINGLAB-II [Maloney *et al.*, 1989], les objets sont des entités appelées *Things* dont les parties sont inter-connectées par des contraintes. Une bibliothèque d'objets primitifs est fournie, à partir desquels l'utilisateur élabore ses objets. Parmi les objets primitifs, les variables à historique sont capables de conserver leurs états (valeurs) précédents permettant ainsi la déclaration de contraintes temporelles utiles, notamment pour les animations (ANIMUS). L'une des différences essentielles entre THINGLAB-II et THINGLAB est qu'il intègre un algorithme incrémental de satisfaction de contraintes conçu pour prendre en compte une *hiérarchie de contraintes*

4. Une contrainte $c(x_1, \ldots, x_n)$ est fonctionnelle s'il existe au moins une variable $x_i \in \{x_1, \ldots, x_n\}$ telle que $x_i = F_i(x_1, \ldots, x_{i-1}, x_{i+1}, x_n)$ où $F_i$ est une fonction.

[Borning *et al.*, 1987]. Dans une hiérarchie de contraintes, les contraintes sont regroupées dans des ensembles par niveaux de préférence (obligatoire, forte, faible ... ). Ainsi, dans un CSP $(X, D, C)$, on a $C = \{C_0, C_1, \ldots, C_n\}$ où $C_0$ est l'ensemble des contraintes obligatoires, $C_1, \ldots, C_n$ sont les ensembles de contraintes ordonnées par ordre décroissant de préférence (fortes, faibles ... ). Les contraintes obligatoires doivent être satisfaites alors que les autres (fortes, faibles ... ) peuvent ne pas l'être. Le but de l'algorithme, appelé *DeltaBlue*, [Freeman-Benson *et al.*, 1990] est de fournir, lors de l'ajout ou du retrait d'une contrainte, la meilleure solution possible. Le critère de comparaison retenu ici est local et consiste à dire qu'une solution $x$ du CSP à hiérarchie de contraintes est meilleure qu'une solution $y$ si $\forall c \in C_1, \ldots, C_{k-1}$, $x$ satisfait $c$ si $y$ satisfait $c$ et, au niveau de préférence $k$, $x$ satisfait au moins une contrainte de plus que $y$. Comme THINGLAB, THINGLAB-II n'est capable que de propagation locale, il n'est adapté qu'aux CSP sans cycle et ne gère que des contraintes fonctionnelles.

#### Kaleidoscope

KALEIDOSCOPE [Freeman-Benson, 1990] est un langage à objets qui réalise la fusion de la programmation par objets impérative et de la programmation par contraintes. L'idée est de permettre à l'utilisateur de définir des relations durables entre des objets et de décrire des séquences de relations entre les états successifs du programme. Ainsi, chaque variable garde un historique de ses valeurs. KALEIDOSCOPE est à rapprocher de THINGLAB-II par sa gestion d'une hiérarchie de contraintes (voir tableau 9.2) à l'aide de l'algorithme *DeltaBlue*, mais ne comporte pas d'algorithme de *retour-arrière*. Les contraintes consistent en des égalités construites à partir d'opérateurs arithmétiques et booléens classiques. Un aspect intéressant du langage sont les *vues* (voir tableau 9.3) — d'où KALEIDOSCOPE tire son nom — qui sont des objets du langage qui permettent à l'utilisateur non seulement de considérer un objet sous plusieurs points de vue (par exemple, une température en Celsius, Fahrenheit ou Kelvin), mais aussi de lier chacun des objets rassemblés sous une vue (ici, Celsius, Fahrenheit et Kelvin) par des contraintes. KALEIDOSCOPE est plus particulièrement adapté aux applications graphiques interactives où les objets évoluent au cours du temps. Malgré sa limitation aux contraintes fonctionnelles et l'absence de techniques de résolution, la définition de contraintes *complexes* et la prise en compte de vues constituent les originalités marquantes de KALEIDOSCOPE. Il a été sensiblement modifié en KALEIDOSCOPE 93 [Lopez *et al.*, 1993] qui intègre des contraintes d'identité et s'ouvre à la programmation concurrente par contraintes. Surtout, il est basé sur un *modèle de perturbation* plutôt que *de raffinement* : une affectation de valeur à une variable est cette fois traitée comme un changement de valeur et non plus comme une contrainte supplémentaire qui peut ou non être encore satisfaite ; le système la prend en compte et cherche la meilleure solution. Son résolveur est basé sur l'algorithme incrémental de propagation locale *SkyBlue* qui prend en charge des contraintes à sorties multiples, maintenues par des méthodes qui permettent d'obtenir simultanément la valeur de plusieurs variables à partir d'une ou plusieurs variables d'entrée.

```
class Dash subclass of Object
  public var left, length, color;
  public virtvar right;
  initially
    always: rigth - left = length;
    always: length >= 1;
    always: length <= 127;
    weak color ← Black;
  end initially;
end Dash ;
```

```
class Point
  constructor +(q, r)
    always: self.x + q.x = r.x;
    always: self.y + q.y = r.y;
  end +;
end Point;
```

TAB. 9.2: *Exemples de classes en* KALEIDOSCOPE. *À gauche, déclaration de trois contraintes obligatoires* `always` *et d'une contrainte faible* `weak` *; à droite, définition d'une contrainte* `complexe` *à l'aide d'un constructeur : ici, l'ajout de points.*

### 9.2.2 Solver : un langage de contraintes avec des objets

SOLVER [ILOG, 1991c] est une bibliothèque de programmation par contraintes commercialisée par la société ILOG et destinée au langage C++, alors que son prédécesseur PECOS [ILOG, 1991a] était destiné au langage LELISP [Chailloux *et al.*, 1986].

Inspiré de la programmation logique par contraintes, et plus particulièrement des techniques implantées dans le langage CHIP [Van Hentenryck, 1989], SOLVER permet de définir dans une syntaxe à la C++ des CSP, applique sur eux et de manière incrémentale un algorithme d'arc-consistance, et fournit un algorithme indéterministe de recherche de solution basé sur un *retour-arrière* qui peut être contrôlé par la pose de points de choix. L'ordre d'énumération des variables et des valeurs peut être établi par des critères pré-existants ou définis par l'utilisateur.

En SOLVER, un CSP est construit à partir de la définition de *variables contraintes* à chacune desquelles est associé un domaine. Ces variables contraintes peuvent être de type quelconque. En particulier, le domaine d'une variable contrainte peut être un ensemble d'objets. Les variables contraintes entières ont un domaine associé qui s'exprime sous la forme d'un intervalle ou d'un ensemble. Les variables contraintes flottantes ont un domaine défini sous la forme d'un intervalle délimité par deux flottants. Une *précision* permet de fixer l'amplitude minimum de l'intervalle. Les variables contraintes ensemblistes ont un domaine défini par la donnée d'une borne inférieure (intersection de toutes les valeurs — ce sont des ensembles — du domaine) qui correspond à l'ensemble des éléments obligatoires et d'une borne supérieure (union de toutes les valeurs — ce sont des ensembles — du domaine)

```
view Celsius <--> Fahrenheit <--> Kelvin
  always: Celsius * 1.8 = Fahrenheit - 32;
  always: Celsius = Kelvin + 273;
end view;
```

TAB. 9.3: *Définition d'une vue en* KALEIDOSCOPE

| | |
|---|---|
| `(defctconstraint myeq`<br>`  ((x ct-var) (y ct-var))`<br>`  ((when (x = a) assert (y = a))`<br>`  (when (y = a) assert (x = a)))`<br>`))` | `(defctclass tache`<br>`  nom`<br>`  duree`<br>`  (debut (ct-fix-range-var 0 1000)`<br>`  constrained))` |

TAB. 9.4: *Dans la syntaxe lispienne de* PECOS*, à gauche, définition d'une nouvelle contrainte d'égalité qui n'est activée que lorsqu'une des deux variables a une valeur. À droite, définition d'une classe contrainte. Le champ* `début` *est associé à une variable contrainte.*

qui fournit les éléments possibles. Les variables contraintes booléennes ont pour domaine $\{0, 1\}$ par défaut. Des fonctions de manipulation des domaines de toutes ces variables contraintes sont disponibles.

Pour chacun des types prédéfinis de variables, SOLVER offre un ensemble de contraintes qui correspondent chacune à un opérateur ($+$, $-$, $*$, $/$, $log$, $exp$, $\cup$, $\cap$, $\in$, $\ldots$). Ces contraintes sont en fait des fonctions dont le résultat est une variable contrainte cachée et gérée par le système. Pour les entiers, comme pour les flottants, le calcul de l'intervalle de cette variable contrainte résultat se base sur les règles de l'arithmétique des intervalles (voir § 9.1.6). Par exemple, si le domaine de `x` est $[1, 3]$, celui de `y` est $[0, 6]$, le domaine de `x + y` est $[1, 9]$. D'autres contraintes correspondent aux comparateurs (égalité, inégalité, différence ... ) et permettent de construire des expressions contraintes (par exemple, pour les entiers, `x = y + z` ou `x + y <= z` ). Contrairement aux opérateurs, ces comparateurs n'ont pas pour résultat une variable contrainte. Ils constituent des contraintes entre deux intervalles et agissent sur l'un ou/et l'autre de ces intervalles d'après la sémantique du comparateur.

Un des atouts majeurs de SOLVER est la définition de nouvelles contraintes (voir tableau 9.4) qui permet à l'utilisateur d'étendre l'ensemble des contraintes prédéfinies. Les contraintes étant des objets du langage hôte, il est possible de définir une nouvelle classe de contrainte en indiquant son nom, la liste de ses paramètres (on donnera le type des variables contraintes sur lesquelles porte la contrainte) et une liste de *schémas* de contrainte. Les schémas de contraintes définissent les règles de la propagation. Celle-ci est basée sur un mécanisme de démons qui, dès qu'un événement de propagation survient (en général, la modification du domaine d'une variable contrainte), déclenche le traitement approprié qui vise à rétablir l'arc-consistance du réseau.

En SOLVER, les classes d'objets peuvent être contraintes par la donnée de contraintes impliquant les *champs* ou attributs de ces classes. Ainsi, on peut préserver la représentation intuitive et structurée d'un problème grâce aux objets et contraindre ces objets en disposant du même arsenal que pour des variables contraintes. De fait, les algorithmes de résolution s'appliquent également sur les CSP dont les variables sont des champs d'objets. Les contraintes de classe portent sur tous les objets d'une classe ; elles n'ont besoin d'être exprimées qu'une seule fois et sont héritées par les sous-classes.

Le processus de base de la résolution de contraintes des CSP à domaines finis consiste en une énumération du domaine des variables basée sur un retour-arrière classique. Des critères standards de choix des variables et des valeurs sont disponibles, mais d'autres peuvent être décrits par l'utilisateur. Une primitive d'optimisation permet de déterminer la meilleure solution vis-à-vis d'un critère donné. Des contraintes non déterministes permettent de dicter au résolveur un ensemble d'alternatives (construites à base de *ou* et de *et* logiques, de branchements conditionnels) à explorer. Pour ce faire, SOLVER permet à l'utilisateur de guider le retour-arrière en fixant lui-même les points de choix vers lesquels l'algorithme retournera en cas d'échec.

Les succès industriels de PECOS et de SOLVER, notamment dans les problèmes d'allocations de ressources, d'optimisation de découpe, de planification, attestent de leur puissance de modélisation — à laquelle n'est pas étrangère le recours aux objets — et de résolution. Ces bibliothèques très complètes se montrent en effet facilement paramétrables et extensibles (choix dans la résolution, ajouts de contraintes, spécifications d'algorithmes particuliers de recherche . . . ).

### 9.2.3 Les langages de programmation par contraintes décrits par des objets

**Prose**

PROSE [Berlandier, 1992] est une “boîte à outils” pour l'interprétation des contraintes qui propose un ensemble de fonctions pour la satisfaction d'un CSP à domaines finis et discrets et pour le maintien de solution. Un générateur de problèmes-tests permet à l'utilisateur de juger des performances des heuristiques ou méthodes de résolution qu'il propose. Tout le processus de satisfaction de contraintes, de l'arc-consistance à l'énumération, peut être contrôlé et paramétré par l'utilisateur.

PROSE prône une approche globale dans le maintien de solution qui repose sur un parcours du graphe des contraintes afin de découvrir les chemins dans lesquels chaque variable n'est calculée qu'une seule fois. Cette phase de planification des graphes de propagation est alors suivie par une phase d'évaluation. L'avantage de cet algorithme est qu'il prend en compte des modifications simultanées (changements de valeurs ou ajouts de contraintes). L'idée est de rétablir la consistance en ne modifiant au plus qu'une variable pour chaque contrainte sans former de cycle. Pour les réseaux contenant des cycles, des adaptations ont été proposées [Trombettoni, 1992].

En PROSE, les composants d'un CSP sont représentés par des objets du langage objet hôte à l'aide de trois classes : *Variable, Constraint, Relation* (voir figure 9.1). Un objet de la classe *Variable* correspond à un attribut contraint d'un autre objet de la base et stocke la liste des contraintes portant sur la variable qu'il représente. Un objet de la classe *Constraint* sert de lien entre un objet variable et l'objet décrivant la relation correspondant à la contrainte. Un objet de la classe *Relation* est lié à la contrainte dont il décrit la liste des types des paramètres, le prédicat de test de satisfaction, et la liste des méthodes qui permettent d'effectuer le calcul des valeurs. Grâce à cette représentation, les objets du domaine (c'est-à-dire les objets qui ne

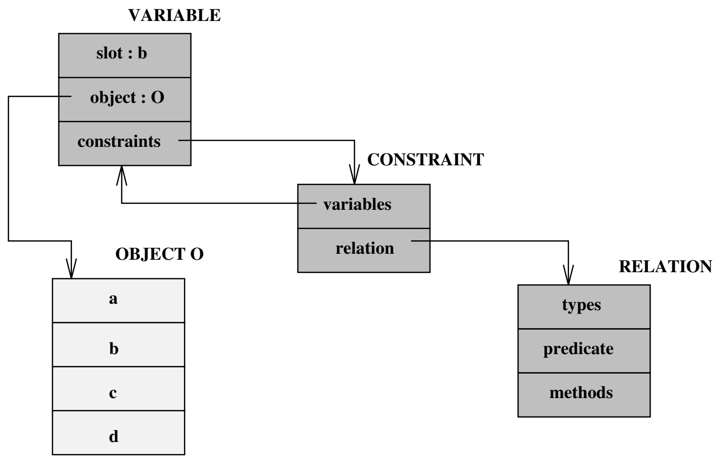



FIG. 9.1: *L'architecture proposée pour contraindre les objets à l'aide de* PROSE. *Liens entre les trois classes de composants d'un* CSP *et l'objet O contraint.*

sont pas des objets du système de contraintes) ne sont pas modifiés par la présence de contraintes. Indépendants, ils sont simplement accessibles par les objets du système de contraintes. N'importe quel objet du domaine peut être contraint à n'importe quel moment, ce qui n'est pas le cas des classes d'objets. Les contraintes, simples ou composées, peuvent être ajoutées ou supprimées à tout moment, ce qui confère un caractère dynamique aux CSP.

Une des originalités de ce système est la gestion de *méta-contraintes* dont trois formes sont possibles. Premièrement, en tant que contrainte impliquant des objets contraintes, une méta-contrainte peut, par exemple, imposer que ces contraintes portent sur les mêmes attributs, soient actives ou non en même temps, aient une relation différente. Deuxièmement, en tant que contrainte impliquant un ensemble d'attributs, une méta-contrainte permet de répondre aux exigences de cohérence d'une base de connaissances dynamiques. Par exemple, il est possible d'établir une contrainte qui assure que le `coût` du `mobilier` d'un bureau `b` est égal à la somme des `prix` des `meubles` de ce bureau (`C` : `b.coût`$= \sum_{m_i \in b.mobilier} m_i$.`prix`). Si le mobilier change alors le coût du bureau sera ré-évalué par la formule, mais aussi, si l'un des prix des meubles du mobilier du bureau change, le coût du bureau sera ré-évalué. Troisièmement, en tant que contrainte impliquant l'activité d'autres contraintes, une méta-contrainte peut subordonner l'activité d'une contrainte (sa présence dans le réseau) à la satisfaction d'une condition. Par exemple, on peut lier le budget du laboratoire `l` où se trouve le bureau `b`, de sorte que si le budget est faible, le coût de l'ameublement du bureau `b` n'excède pas une certaine limite, par la contrainte $C$ : si `l.budget` = `faible` alors `b.coût` $<$ `limite`. La méta-contrainte permet l'ajout ou le retrait d'une condition à tout moment, et n'impose le recalcul de la contrainte que lorsque les variables impliquées par la condition sont modifiées.

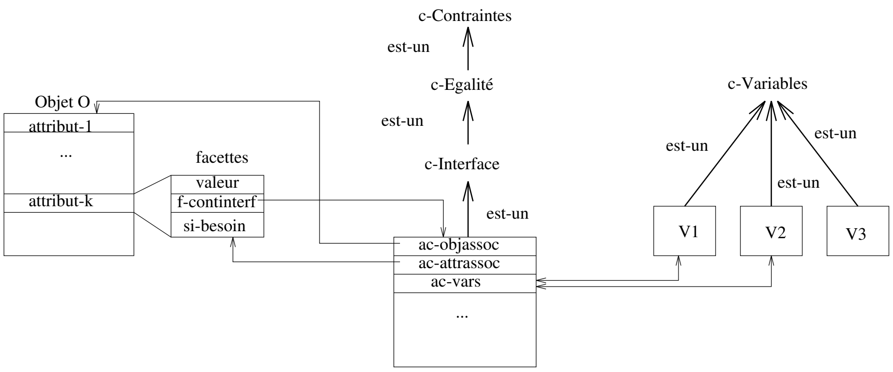



FIG. 9.2: *L'architecture proposée pour contraindre les objets à l'aide de* CSPOO. *Les contraintes portant sur les attributs des objets permettent par des liens de retrouver les variables contraintes correspondantes.*

**Cspoo**

CSPOO [Kökény, 1994] est un système voué à la représentation et à la résolution de contraintes dans un environnement à objets. L'objectif de ce travail était de trouver une représentation, aussi bien pour les composants d'un CSP (variables, domaines, contraintes) que pour les algorithmes de résolution qui soit à la fois modulaire, extensible, indépendante du domaine d'application visé et garante de l'efficacité algorithmique. Le choix d'un environnement objet semble donc naturel pour les deux premiers critères, aussi deux classes *c-Variables* et *c-Contraintes* sont considérées qui suffisent à modéliser les composants d'un CSP. La prise en compte des deux derniers critères est rendue possible par une classe *c-Solveurs* destinée à représenter les spécificités des techniques de résolution employées pour un CSP particulier (voir figure 9.2). Un objet de la classe *c-Variables* correspond à une variable du CSP et décrit le domaine de la variable, les variables adjacentes, les contraintes qui portent sur la variable, la valeur courante. Parmi les méthodes qui sont attachées à cette classe, hormis une méthode d'instanciation, on dispose de méthodes qui permettent de trouver une affectation de valeur consistante pour la variable à partir d'une affectation partielle des variables du CSP. Un objet de la classe *c-Contraintes* correspond à une contrainte du CSP et décrit les variables de la contrainte et les contraintes adjacentes. Les méthodes de cette classe, hormis la méthode d'instanciation, permettent le filtrage et la résolution globale selon la spécificité de la contrainte. Un objet de la classe *c-Solveurs* correspond à une méthode de résolution du CSP et décrit l'ensemble des variables du CSP traité, l'ensemble des variables déjà affectées, l'ensemble des contraintes, la méthode (*Backtracking, Backjumping*) et les heuristiques (ordonnancement statique ou dynamique) de résolution choisis. Cette classe fournit donc, pour un CSP donné, l'algorithme de pré-filtrage utilisé, le mode d'ordonnancement des variables à appliquer ainsi que l'algorithme d'énumération.

Une hiérarchie de classes de contraintes distinguant les contraintes selon leur arité et leur expression (en intension ou en extension), permet d'introduire pour chacune de ces classes une méthode de consistance, de filtrage ou de propagation. Con-

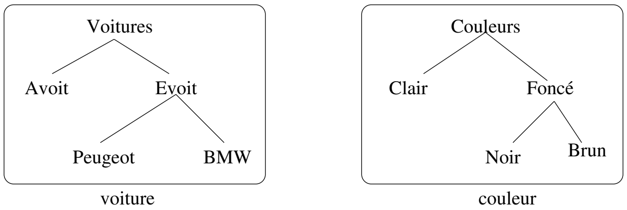



FIG. 9.3: *Les domaines hiérarchiques de deux variables* `Voitures` *et* `Couleurs`.

cernant les algorithmes de résolution et de filtrage (consistance), la représentation choisie et les méthodes attachées aux variables permettent de diriger la résolution (ordonnancement statique et dynamique de variables, points de choix dans le *retour-arrière*, filtrage partiel ou sélectif des domaines selon la contrainte ... ). Dans CSPOO, un algorithme d'arc-consistance, appelé HAC-6 et adapté aux CSP binaires à domaines hiérarchiques, est implanté. La structure hiérarchique des domaines est sous-jacente au monde des objets. Ainsi, par exemple, la pose d'une contrainte dont l'extension est le seul tuple {`(Evoit, Foncé)`} (où `Evoit` désigne la classe des voitures européennes et `Foncé` caractérise la couleur), représente, au regard des hiérarchies `Voitures` et `Couleurs`, une simplification de l'extension réelle qui est (voir figure 9.3) : { `(Evoit, Noir)`, `(Evoit, Brun)`, `(Peugeot, Foncé)`, `(BMW, Foncé)`, `(Peugeot, Noir)`, `(BMW, Noir)`, `(Peugeot, Brun)`, `(BMW, Brun)`, `(Evoit, Foncé)` }. Dans ce contexte, l'hypothèse selon laquelle : *si un objet, valeur d'une variable contrainte, satisfait une contrainte c, alors toutes les spécialisations de cet objet satisfont aussi la contrainte c, et, inversement, s'il ne satisfait pas c, alors aucune de ses généralisations ne la satisfait*, est essentielle. Elle permet de simplifier la définition des contraintes, d'améliorer les algorithmes de filtrage mais aussi de résolution et de compacter l'écriture des solutions. HAC-6 se base sur la recherche des *éléments maximaux* des domaines, ceux dont on peut dire que, s'ils ont un support, alors celui-ci est support de toutes leurs spécialisations. Il est alors inutile de chercher un support pour un élément inférieur (au sens de la hiérarchie établie) de ce domaine. En raison de la structure hiérarchique, la vérification de la compatibilité de deux valeurs est dépendante de la situation de ces valeurs par rapport à la forme abrégée de la contrainte. Par exemple, si on cherche à satisfaire `(Evoit, Foncé)` pour les valeurs `Peugeot` et `Brun`, un parcours ascendant des hiérarchies est nécessaire qui peut être coûteux. Kökény préconise de calculer la fermeture par *h-compatibilité* des contraintes (c'est-à-dire accroître l'extension par les tuples solutions des éléments inférieurs). Sur ce même principe d'élément maximal, la résolution d'un CSP à domaines hiérarchiques peut se réduire à la recherche de solutions maximales en choisissant les valeurs des domaines de l'élément le plus général au plus spécialisé, ce qui contribue à rendre les solutions plus synthétiques.

Signalons que la seule implémentation de CSPOO a été réalisée dans l'environnement objets Y3 [Ducournau, 1991]. Elle bénéficie de l'interface graphique YAFEN qui facilite la visualisation lors de la définition et de la résolution des CSP.

### 9.2.4 En conclusion

Il existe de nombreux autres systèmes associant objets et contraintes, nous avons voulu décrire ici les principes les plus communément adoptés. Des systèmes comme GARNET [Myers *et al.*, 1992], LAURE [Caseau, 1994], COOL [Avesani *et al.*, 1990], SOCLE [Harris, 1986], LIFE [Aït-Kaci et Podelski, 1991] et OZ [Smolka *et al.*, 1993] ont également des spécificités intéressantes. En fait, il est possible de les distinguer les uns des autres selon plusieurs critères :

- La finalité. On peut distinguer les systèmes dédiés à la gestion d'interface graphique (famille THINGLAB, GARNET), les bibliothèques de programmation par contraintes (SOLVER) qui étendent un langage particulier, les boîtes à outils de programmation par contraintes (PROSE) ou les modèles (CSPOO) qui sont destinés à être immergés dans n'importe quel langage à objets, les extensions de langage à objets vers les contraintes (COOL, SOCLE) et les langages hybrides (LAURE).
- Le type des variables contraintes. Entiers et réels sont souvent représentés. En règle générale, les systèmes pour lesquels les contraintes sont à définir traitent des types quelconques. Des contraintes peuvent également être fournies sur les types prédéfinis (entiers, réels, booléens ... ).
- Le domaine des variables contraintes. Tous ces systèmes n'associent pas forcément d'ensembles de valeurs possibles aux variables contraintes (famille THINGLAB, GARNET, SOCLE). Lorsque c'est le cas, ce domaine peut être fini (PROSE) ou continu (SOLVER), sous forme d'énumération ou d'intervalle. Les domaines peuvent également avoir une structure hiérarchique qui est exploitée par les algorithmes de traitement du CSP (CSPOO).
- Les contraintes qui, selon le type des variables, peuvent être numériques, booléennes, symboliques. D'autre part, il est possible de distinguer ces systèmes par la forme des contraintes qu'ils sont capables de traiter : contraintes fonctionnelles (famille THINGLAB, GARNET, SOCLE, PROSE) ou non fonctionnelles (SOLVER, LAURE), monodirectionnelles (GARNET) ou multidirectionnelles, linéaires ou non linéaires. Certains systèmes proposent également d'établir des hiérarchies de contraintes construites sur les préférences à accorder lors de la satisfaction (THINGLAB-II, KALEIDOSCOPE).
- Les techniques de consistance et de résolution employées. On peut distinguer divers traitements non forcément exclusifs :
  - la propagation de valeur qui consiste, lors de la modification de la valeur d'une variable, à propager ce changement aux variables attenantes. Les systèmes basés sur un modèle de perturbation emploient cette technique. Lorsqu'une solution est perturbée par une modification, le système tente de retrouver une stabilité à travers un graphe d'activation de méthodes à exécuter. Les contraintes sont ici fonctionnelles. Ces systèmes (famille THINGLAB, GARNET, KALEIDOSCOPE, PROSE) réalisent du maintien de solution.

  - la propagation de contraintes ou propagation de consistance est appliquée dans les systèmes où les variables ont un domaine associé (SOLVER, LAURE, CSPOO, COOL). Elle consiste, lors de la définition d'une contrainte, à établir la consistance des domaines des variables de la contrainte et à propager ces changements aux variables attenantes en réactivant les contraintes voisines.
  - la résolution ou satisfaction de contraintes (SOLVER, LAURE, CSPOO, COOL) proprement dite consiste à déterminer les solutions du CSP. Diverses techniques peuvent être employées qui dépendent du type et du domaine des variables, mais aussi des contraintes.

- Le type de liaison objets/contraintes. Les contraintes peuvent être définies et décrites à l'intérieur des objets (famille THINGLAB, GARNET, KALEIDOSCOPE). On peut alors parler d'intégration de contraintes dans les objets. Les contraintes peuvent être posées sur les objets. Plusieurs configurations sont alors possibles :
  - un couplage entre un langage de programmation par contraintes et un langage à objets. On distingue alors les couplages faibles (SOCLE) qui sont assurés par une interface entre les deux systèmes, et les couplages forts (COOL, PROSE, CSPOO) dans lesquels les contraintes sont décrites elles-mêmes par des objets du langages à objets.
  - une extension d'un langage à objets existant par la définition d'une bibliothèque de programmation par contraintes (SOLVER).
  - un langage hybride intégrant objets, contraintes et d'autres paradigmes (LAURE).

À travers cette étude, il ressort que d'un côté le paradigme "objet" offre :

- des entités qui rassemblent les connaissances propres à une famille ou à un seul élément du domaine d'application. Dans les langages à objets intégrant des contraintes, ce principe est exploité dans la définition de contraintes à l'intérieur des classes. Les contraintes de classe induisent alors des contraintes sur chacune des instances de la classe. Dans les langages de contraintes utilisant les objets pour représenter les composants (voire les techniques de résolution) d'un CSP, la classe est un moyen de décrire les caractéristiques de ces composants — c'est le rôle des attributs —, mais aussi les méthodes à activer, comme la propagation lors de la modification du domaine d'une variable ou la pose (instanciation) d'une contrainte.
- une organisation hiérarchique des descriptions sur laquelle se greffe un mécanisme d'héritage qui permet une factorisation de la connaissance. Dans les langages à objets intégrant des contraintes, ce principe est souvent exploité à travers l'héritage des contraintes de classe qui permet de factoriser la déclaration de contraintes. Dans les langages de contraintes utilisant les objets comme éléments de représentation, l'organisation des divers composants en hiérarchies permet d'associer des comportements communs (par héritage) ou spécifiques (par redéfinition d'attributs, de méthodes) aux divers types de variables ou

de contraintes. Lorsque les domaines sont eux-mêmes organisés hiérarchiquement, cet ordre est exploité pour accélérer les processus de maintien de consistance et/ou de résolution.

- des mécanismes de création et de manipulation d'objets. Dans les langages à objets intégrant des contraintes de classe, les contraintes d'une classe sont associées à une instance dès sa création. La modification d'un objet donne lieu à une propagation. Dans les langages de contraintes utilisant les objets comme éléments de représentation, l'instanciation correspond à la définition d'une variable, à la pose d'une contrainte ou à la définition d'un CSP et d'un algorithme de résolution. Dans les langages qui font les deux, cela permet la définition de méta-contraintes en tant que contraintes sur des attributs de contraintes.
- des liens entre objets modélisés par les attributs de ces objets dont la valeur est un (ou plusieurs) autre(s) objet(s). Dans les langages à objets intégrant des contraintes, ces liens (relations), parcourus par des chemins, permettent de contraindre des attributs d'autres objets. Les contraintes viennent se greffer sur les relations.
- une représentation de la connaissance par divers niveaux d'abstraction ou de *décomposition* : un `triangle` est formé de `points` dont les coordonnées sont des `entiers`. Les contraintes portant sur les entités des plus hauts niveaux se décomposent récursivement en contraintes de niveaux inférieurs. Deux triangles sont égaux si leurs points sont égaux, deux points sont égaux si leur coordonnées sont égales. Retrouver ces niveaux d'abstraction sur les contraintes réduit considérablement l'effort de définition des CSP : la contrainte complexe n'est définie qu'une seule fois en fonction des contraintes de niveau inférieur qui la composent. À sa création, ses sous-contraintes sont créées également. Dans les langages à objets intégrant des contraintes, cela permet la définition de contraintes sur des objets ou sur des attributs dont la valeur est un objet. Les variables contraintes ne se limitent donc plus ici aux types simples habituels, mais peuvent être de n'importe quel type. Dans les langages de contraintes utilisant les objets comme éléments de représentation, la description de la contrainte composée doit faire référence à ses contraintes composantes. L'instanciation se charge de la pose des contraintes composantes de plus bas niveau.

De l'autre côté, le paradigme "contrainte" offre :

- la possibilité de décrire un problème par l'énoncé de relations entre les données de ce problème. Dans les objets, on utilise la déclarativité pour définir les relations qui doivent être maintenues entre les attributs (d'un même objet, d'objets différents, de tous les objets d'une classe ... ) ou entre des objets.
- un arsenal de techniques (propagation de valeur, propagation de contraintes, arc-consistance, algorithmes et heuristiques de résolution) sous-jacentes à ces énoncés, qui permettent, sinon de résoudre le problème, tout au moins de réduire son espace de recherche de solution. Les associations objets/contraintes

se font sur un modèle de *perturbation* — c'est le cas la plupart du temps dans les systèmes pour le développement d'interfaces — ou sur un modèle de *raffinement*. Dans le premier cas, le système a une solution et doit réagir à une perturbation en établissant un plan de méthodes à exécuter. Dans le second cas, les contraintes sont posées et agissent par réduction sur les domaines des valeurs possibles des variables contraintes, la résolution constituant l'étape suivante.

## 9.3 Contraintes et objets pour la représentation

Après nous être focalisés dans la section précédente sur les langages de programmation, nous nous intéressons ici aux associations d'objets et de contraintes dans le domaine de la représentation de connaissances par objets ou RCO. Dans ce domaine, la distinction est souvent faite entre les logiques de descriptions (voir *chapitre 11*) et les langages de classes (voir *chapitre 10*), appelés SRCO [Chouvet *et al.*, 1996]. Nous adoptons ici cette distinction pour étudier l'intégration de contraintes.

### 9.3.1 Contraintes et logiques de descriptions

Nous décrivons deux travaux pour l'intégration de contraintes dans des logiques de descriptions (voir *chapitre 11*) : l'un [Baader et Hanschke, 1991] vise à introduire des valeurs et des prédicats dans les logiques de descriptions, et l'autre consiste en un langage de contraintes appelé CONTAX [Meyer, 1992b] couplé au dessus d'une logique de descriptions appelé TAXON [Baader et Hanschke, 1991]. Baader et Hanschke ont constaté que les logiques de descriptions ne permettaient pas de manipuler aisément des domaines concrets (comme les entiers, les réels, par exemple) et des prédicats sur ces domaines. En effet, la solution à base de *descriptions structurelles* proposée pour KL-ONE [Brachman et Schmolze, 1985], dans laquelle les prédicats à vérifier entre rôles sont décrits par des concepts abstraits, n'est pas satisfaisante. Cependant, d'autres solutions ont été envisagées comme les *restrictions de rôles* de CLASSIC [Brachman *et al.*, 1991]. Les restrictions sont des opérateurs spéciaux qui imposent des conditions sur le domaine des attributs (`all`, `at-least`, `at-most`, `fills`, `same-as`), les arguments de ces opérateurs devant être des concepts définis. Par exemple `(at-least 1 child)` permet de définir un concept pour lequel le rôle `child` a au moins une valeur.

La motivation de Baader et Hanschke se rapproche davantage de celle qui est à l'origine de l'extension de la programmation logique vers la programmation logique par contraintes : introduire les domaines concrets dans les concepts (les *domaines abstraits*) et pouvoir définir les rôles de ces concepts en faisant référence à des valeurs de ces domaines concrets, voire même, lier les rôles de concepts par des prédicats sur ces domaines. Leur contribution réside en l'extension du langage $\mathcal{ALC}$ [Schmidt-Schauß et Smolka, 1991] vers le langage $\mathcal{ALCFP}(\mathcal{D})$ de sorte qu'il soit possible de définir les concepts représentant une restriction généralisée $(\forall u_1, \ldots, u_n.p)$, ou une restriction existentielle $(\exists u_1, \ldots, u_n.p)$ avec $u_i$ représentant une séquence de rôles ou d'attributs (un rôle fonctionnel qui n'a qu'une valeur), et

$p$ est un terme qui est soit un prédicat d'arité $n$ sur le domaine concret $\mathcal{D}$, soit un concept ($n = 1$), soit $=$ ou $\neq$ ($n = 2$).

Par exemple, on peut définir les concepts :
`Mother = Human` $\sqcap$ `Female` $\sqcap$ $\exists$ `child.Human`: une mère est un humain femelle qui a un enfant humain.
`Femme = Human` $\sqcap$ `Female` $\sqcap$ `(Mother` $\sqcup$ $\geq_{21}$`(age))` : une femme est une mère ou est un humain femelle de plus de 21 ans, en posant que, sur les entiers, $\geq_{21}(x) \equiv x \geq 21$.

Sur ce langage, les auteurs ont prouvé qu'il existe un procédure de décision qui permet d'établir la satisfiabilité d'une base d'assertions sur les instances des concepts ou termes, et la subsomption de deux termes de l'ensemble des concepts ou termes. L'insertion de prédicats dans les définitions de termes augmente le pouvoir de représentation des logiques de descriptions vers la description de relations entre rôles, relations qui sont prises en compte pour vérifier l'appartenance d'une assertion à un terme, mais aussi la subsomption entre termes. Cependant, on ne peut parler ici de contraintes, au sens où aucune propagation ou résolution de contraintes n'est proposée, le système demeure au niveau simplement *prédicatif*. On trouvera dans [Buchheit *et al.*, 1993] une poursuite de l'intégration d'opérateurs de restrictions de rôle et quelques contraintes simples de typage qui préserve la décidabilité du raisonnement classificatoire dans une logique de descriptions.

CONTAX, quant à lui, est la composante contrainte d'un système hybride appelé COLAB qui comporte également une composante terminologique : le langage TAXON. CONTAX permet de résoudre des CSP dont les variables sont des termes de la logique de descriptions. Autrement dit, CONTAX doit gérer des domaines à structure hiérarchique. À cette fin, CONTAX se base sur une version étendue de l'algorithme d'arc-consistance hiérarchique HAC [Mackworth *et al.*, 1985]. Le système établit l'arc-consistance hiérarchique des domaines en exigeant que ces domaines contiennent le concept le plus général qui soit arc-consistant. Cette information est établie à partir de la hiérarchie de subsomption maintenue par TAXON sur les éléments des domaines. Les domaines peuvent être des ensembles de termes définis et organisés par TAXON ou des intervalles d'entiers. Les contraintes peuvent être définies en extension ou en intension. Des contraintes composées peuvent être formées. Des poids sont associés aux contraintes afin de permettre une relaxation en cas d'échec. Variables et contraintes sont des objets de classes prédéfinies du langage CLOS [Keene, 1989]. La déclaration d'un CSP se fait donc en instanciant ces classes. CONTAX propose un algorithme de propagation visant à rendre les domaines hiérarchiquement arc-consistants, et cherche une instanciation complète ou non des variables qui soit globalement consistante vis-à-vis d'un ensemble de contraintes pondérées. On cherche à résoudre d'abord les contraintes les moins fortes jusqu'aux plus fortes. CONTAX propose une approche de l'arc-consistance hiérarchique proche de celle de CSPOO. Cependant, les contraintes et les variables ne sont pas exprimées ici en tant que concepts ou assertions de la logique de descriptions, mais dans les termes du langage CLOS. La seule liaison qui existe ici entre TAXON et CONTAX se fait au niveau des domaines.

### 9.3.2 Contraintes et systèmes de représentation de connaissances par objets

Nous traitons ici des moyens d'exprimer, de vérifier, de maintenir, voire de résoudre des contraintes entre plusieurs objets ou sur un objet unique au sein d'un système de représentation de connaissances par objets (SRCO). Tout d'abord, il convient de distinguer la notion de *contrainte* de celle de *relation*. Dans le contexte de la représentation de connaissances par objets (RCO), on entend par *relation* la modélisation du ou des liens qui existent entre plusieurs objets. En général, l'établissement de ce lien n'est soumis qu'à la compatibilité des types des objets liés au sein de la relation et, éventuellement, à quelques contraintes dites d'intégrité, qui vérifient la cohérence du lien.

Par *contrainte*, on entend la modélisation d'une propriété décrite par une expression mathématique, à l'aide d'opérateurs de comparaison (égalité, inégalité, différence, appartenance ... ) et d'autres opérateurs algébriques. Alors que la relation lie des objets distincts dans leur ensemble, la contrainte peut ne porter que sur les caractéristiques d'un seul objet. On associera à une contrainte des mécanismes de maintien de consistance et de résolution, alors qu'on pourra associer à la relation des mécanismes d'établissement et de maintien de cohérence des liens [Fornarino et Pinna, 1990].

Il faut également distinguer *contraintes* et *prédicats*. Les prédicats ne sont là que pour vérifier une condition et ne sont activables qu'en présence de tous leurs arguments. Dans SHIRKA [Rechenmann, 1988], par exemple, la facette `$à-vérifier` introduit un prédicat (une fonction booléenne) qui permet d'exprimer une propriété de l'objet mettant en cause un ou plusieurs attributs. Son rôle n'est que *prédicatif* (même s'il se peut que l'instanciation soit refusée) alors que l'on peut entrevoir trois rôles pour une contrainte : *préventif* (par filtrage des domaines, elle ne retient que les valeurs localement consistantes vis-à-vis de la propriété à observer), *inférentiel* (lorsque les domaines filtrés se réduisent à des singletons, une seule valeur est possible ; elle est inférée en devenant valeur effective de l'attribut) et *prédictif* (chercher à satisfaire la contrainte revient à énumérer les configurations d'objets contraints possibles). Les deux premiers rôles ont trait au maintien de consistance, le dernier concerne la résolution.

La gestion de contraintes au sein d'un SRCO passe donc par la mise à disposition du SRCO d'un ensemble de techniques issues de la programmation par contraintes telles que celles présentées au § 9.1. Un réseau de contraintes dans un SRCO forme ce que nous appelerons ici un CSP RCO. Le reste de cette section s'attache à définir ce que sont ces CSP RCO, ce que leur gestion sous-tend et ce que leur présence implique et apporte au SRCO.

### 9.3.3 Les CSP RCO

#### Les composants

Un CSP est défini par trois composants : un ensemble de variables, l'ensemble des domaines de ces variables et un ensemble de contraintes impliquant les variables. Il nous faut donc déterminer la nature de ces trois composants dans le contexte d'une RCO afin de définir les CSP RCO.

Dans une RCO classique, prônant une approche classe/instance, il existe cinq niveaux distincts de représentation : la base de connaissances, la classe, l'instance, l'attribut et la facette.

Aussi, peut-on *a priori* considérer ces entités de représentation comme des variables contraintes potentielles : il suffit de disposer d'opérateurs de manipulation et de comparaison pour construire des contraintes. Chacune de ces entités de représentation possède des caractéristiques ou propriétés (une classe, par exemple, possède son nom, son nombre d'instances, ses super-classes, etc.) qui permettent sa description en tant qu'entité informatique et non pas en tant que représentation d'un objet du monde réel, et une valeur (par exemple, pour une classe, l'ensemble de ses instances, donc son extension) qui, elle, l'assimile à un objet du monde réel. En fonction de la représentation et de l'accès aux caractéristiques et à la valeur d'une entité de représentation, on peut distinguer les contraintes de contrôle ou d'intégrité, qui régissent l'évolution des entités de représentation, des contraintes dites classiques qui régissent la valeur des entités de représentation. Par exemple, dans la première catégorie, on classe la contrainte qui impose que le nombre d'instances d'une classe soit inférieur au nombre d'instances d'une autre classe, dans la seconde catégorie, on classe la contrainte de la classe `Rectangle` entre les attributs `surface, longueur, largeur` qui fixe par là-même le calcul de la surface de tout rectangle.

En considérant qu'il est possible de fournir une définition réflexive d'un SRCO dans laquelle les différentes entités de représentation (bases de connaissances, classes, instances, attributs et facettes) sont décrites par des classes (plus précisément des attributs) et effectivement représentées par des instances, il est possible de restreindre l'ensemble des entités de représentation potentiellement contraignables aux seules instances et attributs. En effet, exprimer une contrainte de contrôle sur une entité peut être fait, moyennant des opérateurs adéquats, sur l'instance qui représente cette entité (cette base de connaissance, cette classe . . . ). On agira ici directement sur les attributs de cette instance qui déterminent les caractéristiques de l'entité (par exemple, sur l'attribut `nombre d'instances` d'une instance de la méta-classe `Classe` . . . ).

Exprimer une contrainte entre des instances qui ne soit pas une contrainte de contrôle revient à lier ces instances par le biais de leurs attributs. Aussi, une contrainte sur des instances se traduit-elle en contraintes sur des attributs de ces instances. C'est pourquoi nous pouvons conclure que les seules variables contraintes qui sont considérées *ultimement* dans les CSP RCO sont les attributs, que ces attributs contribuent à la description purement informatique de l'objet (comme les attributs `identificateur, self, nombre d'instances` . . . ) ou à la description de l'objet du monde réel équivalent (commes les attributs `âge, nom, poids`, etc.). Cependant, il est souvent commode de rassembler dans l'expression simple d'une contrainte sur des instances l'ensemble des contraintes sur attributs auquel elle est équivalente (par exemple, une contrainte d'égalité entre deux instances de la classe `Point` se traduit en deux contraintes d'égalité sur les attributs représentant les abscisses et les ordonnées de ces points). La possibilité de définir des contraintes sur des instances à partir de contraintes sur des attributs doit donc être offerte pour des besoins de déclarativité et de factorisation. Les instances sont, dans ce cas, des pseudo-variables contraintes : elles sont bien arguments de la contrainte mais ce sont réellement leurs attributs qui sont contraints. On peut remarquer que les contraintes sur des attributs dont la valeur est une instance sont équivalentes à des contraintes sur des instances qui, elles-mêmes, se décomposent en contraintes sur des attributs. En prolongement de l'idée de contraindre *réellement* des objets, il faut signaler deux approches connexes et récentes en programmation par objets. La première [Pelenc et Ducournau, 1997] consiste à considérer que le domaine d'une variable contrainte peut être l'extension d'une classe, ce qui conduit naturellement à tenter l'instanciation de cette classe pour déterminer une valeur consistante pour cette variable lorsque le domaine n'en contient. La seconde approche [Pachet et Roy, 1995a] montre l'intérêt de combiner une modélisation classique (à plat) et une modélisation à l'aide d'objets d'un CSP dont les variables sont des objets composites, en vue de réduire à la fois le nombre de contraintes à exprimer et le nombre d'objets à contraindre. Ces deux approches préconisent l'utilisation des méthodes de classes dans l'expression des contraintes.

Les variables des CSP RCO étant reconnues comme les attributs, la détermination des domaines des CSP RCO en découle immédiatement et, par là même, la nature des CSP RCO. Les domaines des attributs d'une RCO étant *a priori* quelconques et donc possiblement infinis et continus, il en va de même pour les domaines des CSP RCO.

Une fois définis les variables et les domaines des CSP RCO, on peut imaginer un éventail de contraintes qui pourraient être utilisées dans des CSP RCO. Sans connaissances ou restrictions préalables des entités à contraindre — et donc dans un souci préservation de la généricité d'un tel système — la gestion de CSP aussi bien numériques, booléens, ensemblistes, que symboliques peut être *a priori* envisagée, puisque les domaines contraints sont quelconques. Il est clair que la nature des CSP RCO dépend des types autorisés pour la description des attributs, c'est-à-dire des variables contraintes.

À un niveau supérieur dans la description de contraintes, il est également envisageable d'introduire, à l'image des contraintes non déterministes de SOLVER ou des méta-contraintes de PROSE, un peu de flexibilité dans la définition des CSP RCO à l'aide de *contraintes disjonctives* qui, selon la configuration proposée, autorisent qu'une contrainte ou bien une autre soit vérifiée, ou encore à l'aide de *contraintes conditionnelles* qui associe la pose de telle ou telle contrainte à une condition particulière.

À ce stade, les caractéristiques générales des CSP d'une RCO ont été énoncées, mais il doit être gardé à l'esprit que des limitations dans la définition de ces CSP apparaissent lors de la tentative d'intégrer des contraintes à un SRCO particulier. Elles proviennent, d'une part des capacités de représentation du SRCO lui-même, en vertu des domaines (types) d'attributs autorisés, et, d'autre part, des capacités déclaratives et de traitement de données du système de contraintes (intégré ou non) chargé de maintenir les CSP établis. Ainsi, il est tentant, mais sans doute illusoire à faible prix, d'étendre la vocation de généricité d'un SRCO (tout représenter) à la gestion des CSP de ce SRCO (tout contraindre, tout maintenir et tout résoudre).

**Portée et statut des contraintes**

Une fois cernés les divers composants des CSP RCO (variables, domaines et contraintes), la question se pose de la portée effective des contraintes. Classiquement, l'approche classe/instance suggère que soient distinguées les *contraintes de classe* qui, définies au niveau de la classe, portent uniformément sur toutes les instances de cette classe, des *contraintes d'instance* qui portent ponctuellement sur une ou plusieurs instances. De même qu'un attribut défini dans une classe est attribut de toute instance de cette classe et de ses sous-classes (où sa description pourra être affinée), la contrainte de classe s'inscrit dans la définition de la classe et devient contrainte d'instance de toutes les instances de cette classe. Ainsi, les contraintes de classe ont des prérogatives "universelles" que n'ont pas les contraintes d'instance. Aussi, la description d'une classe contrainte est maintenant formée de la description de tous ses attributs (hérités, redéfinis ou définis) et de la description de ses contraintes. Au contraire, une contrainte d'instance n'a pas ce caractère définitionnel, et doit être vue comme une propriété qui permet de distinguer l'instance d'autres instances de la même classe et non pas comme une condition présidant à l'existence de l'instance. En tant que traits définitionnels, les contraintes d'une classe seront donc héritées par ses sous-classes comme le sont les attributs. Mais, alors que l'héritage d'un attribut est lié au fait que sa définition soit ou non affinée dans la classe, l'héritage d'une contrainte d'une super-classe, *a priori* systématique, s'avère sujet à caution. En effet, on peut remarquer que si dans la classe est définie une contrainte $c'$ portant sur les mêmes attributs qu'une contrainte $c$ d'une super-classe mais avec une extension plus petite, l'héritage de $c$ est inutile. Cependant, détecter de telles redondances s'avère coûteux à mettre en place.

Enfin, il faut noter que la notion de contrainte entre classes qui vise à contraindre des instances de classes distinctes, lorsqu'il ne s'agit pas d'une contrainte de contrôle (par exemple, exiger que deux classes aient le même nombre d'instances), peut se ramener soit à une contrainte de classe sur un attribut typé par une classe, soit à une contrainte sur un objet relation, si les relations sont réifiées.

**Désignation d'attributs contraints**

Du fait que dans un SRCO les valeurs d'attributs peuvent être des objets, il est possible d'atteindre, à partir d'un objet et par une succession d'indirections appelée *chemin* (de longueur $\geq 1$), un attribut dans un autre objet. Dans un contexte où les attributs sont monovalués, le chemin $O1.a_1.a_2.\ \ldots\ a_{n-1}.a_n$ a pour longueur $n$ et permet d'atteindre à partir d'un objet $O_1$ contenant l'attribut $a_1$, la valeur de l'attribut $a_n$ de l'objet $O_n$ en passant, grâce aux valeurs des attributs intermédiaires $a_j$ (avec $j \in [1..n-1]$) par $n-1$ objets intermédiaires. La valeur du chemin est désignée comme la valeur de $a_n$ dans $O_n$.

Dans les associations objets/contraintes, cette notion de chemin a été introduite dans THINGLAB où elle fut limitée à des chemins longueur 1 et permet de contraindre les attributs d'une partie d'un objet. Elle a été étendue dans CSPOO aux chemins de longueur quelconque mais le long desquels les attributs $a_j$ atteints sont monovalués, leur valeur étant un objet. Dans un contexte plus général, il est possible que les attributs $a_j$ soient multivalués et qu'ils aient donc pour valeur un en-

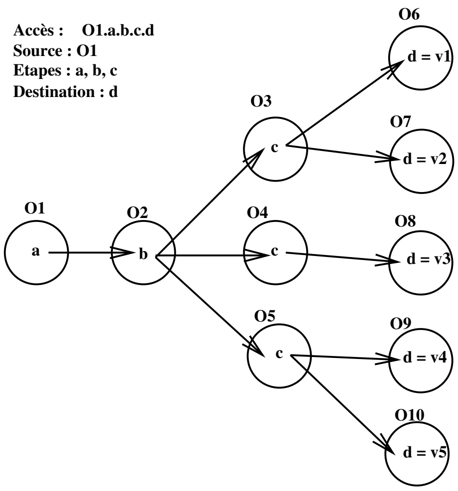



FIG. 9.4: *Exemple de chemin. Sur le chemin O1.a.b.c.d, il y a diffraction au niveau des attributs étapes b et c qui sont des attributs multivalués.*

semble ou une liste d'objets. Dès lors, on n'atteint plus un attribut dans un objet mais un ensemble d'attributs dans des objets. Poursuivre le chemin en n'empruntant qu'un seul des objets de l'ensemble ou de la liste serait arbitraire et impossible sans qu'une telle sélection ne soit représentée dans l'expression même du chemin. En général, la nature ensembliste de la valeur de l'attribut intermédiaire permet de rassembler des objets partageant une propriété commune (au moins celle d'être atteignables directement) avec l'objet courant. Ce tout devant être préservé dans l'interprétation du chemin, un chemin s'éclate en plusieurs chemins possibles à partir d'un attribut intermédiaire multivalué. Cependant, de telles *diffractions* introduisent une indétermination dans l'interprétation du chemin. Ainsi, dans l'exemple de la figure 9.4, la valeur du chemin est-elle l'ensemble $\{v1, v2, v3, v4, v5\}$ des 5 valeurs atteintes ou bien l'ensemble $\{\{v1, v2\}, \{v3\}, \{v4, v5\}\}$ des 3 ensembles de valeurs atteints par les 3 chemins présentés par la diffraction ?

La réduction et l'anti-réduction sont deux opérateurs introduits par [Gensel, 1995] qui permettent à l'utilisateur de lever cette ambiguïté. Ainsi, si la structure de l'attribut multivalué rencontré ne doit pas être préservée, une réduction (de symbole associé ":") est à opérer, si la structure doit être préservée, une anti-réduction (de symbole associé ".") est à opérer. Dans l'exemple donné, le chemin désigné par *O1:a:b:c:d* a pour valeur $\{v1, v2, v3, v4, v5\}$, alors que *O1:a:b.c.d* vaut $\{\{v1, v2\}, \{v3\}, \{v4, v5\}\}$.

Les chemins ainsi définis apparaissent comme une facilité d'écriture permettant de désigner un attribut d'un objet comme devant être impliqué dans une contrainte appartenant à un autre objet. On les trouvera dans l'expression des arguments d'une contrainte. Cependant, si statiquement, il est possible de s'assurer de la validité d'un attribut contraint après avoir parcouru le chemin qui le désigne, il n'est pas garanti, en revanche, qu'un chemin soit entièrement défini au moment de la pose d'une contrainte. En clair, cela signifie qu'un attribut placé sur le chemin n'a pas de valeur et que la contrainte ne porte que partiellement sur l'ensemble de ces arguments. La spécificité des CSP RCO fait que l'existence d'une contrainte est d'abord liée à celle de l'objet sur lequel elle porte avant que d'être soumise à l'existence de ses arguments qui ne sont que des informations accessibles ou non depuis cet objet. Des précautions sont donc à prendre dans la gestion des chemins.

En fait, une contrainte dont un argument est représenté par un chemin non entièrement défini ne peut agir que statiquement sur cet argument, au moment de sa pose, par réduction de son domaine. La gestion de chemins à attributs multivalués doit prendre en compte à la fois le fait que ces chemins peuvent être à un moment donné non entièrement définis, le fait qu'à tout moment la valeur d'un attribut intermédiaire peut être changée, le fait que la valeur du chemin peut être inférée compte-tenu du type de la contrainte et de la connaissance disponible.

### Choix d'une représentation des contraintes

Concernant le choix d'une représentation pour les contraintes portant sur les objets, il paraît intéressant de considérer qu'une instance de contrainte témoigne de l'existence d'une contrainte et donc de la réussite de la pose de cette contrainte. La question qui se pose alors est[5] : les contraintes doivent-elles être des objets à part entière de la RCO et, par conséquent, côtoyer dans une base de connaissances les objets dédiés à la représentation du domaine modélisé ?

Un des avantages de représenter les contraintes comme des objets du modèle est qu'il est dès lors possible de manipuler de manière uniforme les objets (et donc les contraintes) à l'aide des primitives du modèle. Cependant, le comportement associé aux contraintes selon leur portée conduit à de sérieuses restrictions, et notamment à abandonner la solution qui consiste à représenter chaque contrainte, qu'elle soit de classe ou d'instance, portant sur un objet. En effet, on ne pourrait, par exemple, modifier ou supprimer un objet particulier représentant une contrainte de classe présente sur une instance particulière de cette classe sans rompre avec le caractère de généricité attaché aux contraintes de classe. Aussi, mieux vaut extraire les contraintes de la base de connaissances et n'y laisser que les objets dédiés à la modélisation des entités du domaine d'application.

La définition d'une contrainte de classe peut se faire, comme celles des attributs, au moment de la définition de la classe ou, plus tard, dans l'évolution de la base de connaissances. La définition d'une contrainte d'instance se fait sur une ou des instances existantes. La syntaxe choisie doit permettre une expression déclarative des contraintes. Elle peut également avoir intérêt à être liée à la syntaxe choisie par le module de PPC chargé de la gestion des CSP RCO, ne serait-ce que pour éviter une phase de traduction.

5. Une discussion plus générale sur la distinction programmation/représentation est faite dans [Carré *et al.*, 1995].

### 9.3.4 Contraintes, typage et dynamicité

#### Contraintes et types

La tâche essentielle dans la gestion d'un CSP RCO est l'établissement d'un certain niveau de consistance locale qui consiste à éliminer des domaines des attributs contraints les valeurs qui ne pourront pas apparaître dans une solution. Aussi, à un moment donné, un attribut contraint possède-t-il vis-à-vis de la consistance locale un domaine — appelons-le *domaine contraint* — inclus ou égal à celui que lui a attribué l'utilisateur dans la définition de la classe — appelons-le *domaine effectif* (voir *chapitre 10*). En cas d'interrogation de l'utilisateur sur le domaine de l'attribut à un moment donné, le domaine initial peut lui être fourni (il aura éventuellement le loisir d'en modifier la définition à l'aide des primitives du système, ce qui pourrait avoir des répercussions sur le domaine contraint) mais le SRCO doit surtout être en mesure de fournir le domaine contraint qui est le reflet cohérent de ce qui constitue, au regard du niveau de consistance locale atteint, les seules possibilités de valuation de l'attribut en présence des contraintes de classe et des contraintes d'instance.

Également, il faut distinguer le domaine contraint d'un attribut dans une classe, qui est le domaine résultant de l'établissement de la consistance locale en ne tenant compte que des contraintes de classe, du domaine contraint d'un attribut dans une instance qui, lui, est le domaine résultant de l'établissement de la consistance locale tenant à la fois compte des contraintes de classe et des contraintes d'instance portant sur l'instance.

Ces constatations suggèrent que les domaines contraints des attributs dans les classes et les instances soient disponibles après calcul. La réduction de tels domaines incombe au système de gestion de contraintes, mais le stokage et la gestion de ces domaines contraints peuvent être affectés au SRCO ou encore à un système de gestion de types associé au SRCO. Un tel stockage vise à rendre incrémentale et minimale la phase de maintien de la consistance [Capponi et Gensel, 1997].

#### Contraintes et dynamicité

L'une des caractéristiques essentielles des SRCO est de permettre la représentation de connaissances évolutives, c'est-à-dire de permettre à tout moment la création, la modification ou la suppression d'un élément de connaissance descriptif (classe) ou factuel (instance). Aussi, dans le souci d'une intégration harmonieuse des contraintes dans un contexte évolutif, la gestion de CSP RCO dynamiques, autorisant à tout moment l'ajout ou le retrait d'une contrainte, peut être envisagée. Nous étudions ici la répercussion de cette dynamicité au niveau du maintien de la consistance.

L'ajout d'une contrainte est une opération naturelle (la définition de CSP étant la plupart du temps incrémentale) de réduction de domaines. Elle échoue lorsque l'un des domaines devient vide. Dans le cadre de la RCO, l'échec d'un ajout de contrainte peut être détecté statiquement à partir de la définition (donc des domaines) de l'entité contrainte. Ainsi, si le maintien de la consistance locale d'une contrainte de classe vide un des domaines des attributs sur lequel elle porte, il est clair qu'aucune instance ne pourra être attachée à cette classe ; la classe est sur-contrainte : la contrainte trop forte ne peut pas être prise en compte. En revanche, si la contrainte peut être supportée par la définition de la classe, il n'en est pas forcément de même pour toute instance de son extension. Que faire alors d'une instance de la classe qui ne satisfait pas la nouvelle contrainte de classe ? Trois mesures sont envisageables : supprimer la contrainte (on privilégie l'extension de la classe), supprimer l'instance (on privilégie l'intension de la classe), tenter une reclassification de l'instance (on rompt le lien d'appartenance — ce qui n'est possible que si la classe n'est pas la racine de la hiérarchie). L'échec d'une contrainte d'instance, quant à lui, ne peut mener qu'à la suppression de la contrainte.

Le retrait d'une contrainte nécessite de mettre en place un mécanisme de relaxation (voir § 9.1.5) qui peut être coûteux en place mémoire. Le retrait d'une contrainte de classe entraîne la suppression de la contrainte pour toutes les instances de la classe, celui d'une contrainte d'instance concerne ces seules instances. Vis-à-vis de la classification, il est possible que des instances de super-classes puissent, à l'issue de ce retrait, être attachées à la classe.

La dynamicité des autres entités de représentation (classe, instance, attribut), lorsqu'elles sont contraintes, implique également certaines règles de cohérence. Ainsi, la modification de la valeur d'un attribut contraint dans une instance n'est possible que si la valeur proposée est dans le domaine contraint de l'attribut dans l'instance et si elle est consistante avec les autres valeurs.

Lorsqu'une classe est ajoutée à une hiérarchie, elle hérite des contraintes de ses super-classes et fait hériter ses contraintes à ses sous-classes. Pour tester si les contraintes de cette classe n'entraînent pas la non viabilité d'une de ses sous-classes, la propagation de ces contraintes doit se faire depuis les sous-classes feuilles. Si les définitions acceptent les réductions de domaines imposées par les contraintes de la nouvelle classe, alors la classe peut être insérée sans nuire à ses sous-classes.

Le retrait d'une classe doit entraîner le retrait de toutes les contraintes de classe posées sur ses instances qui, si elles lui survivent, seront rattachées à sa ou ses super-classes.

Enfin, lors du retrait d'une instance de la base, les contraintes d'instance qui portent sur elles sont supprimées. En particulier, si celles-ci impliquent d'autres instances, ces dernières en sont libérées.

### 9.3.5 Maintien de consistance et résolution

**Maintien de consistance**

Le maintien de consistance et la résolution des CSP RCO dépendent de la nature des domaines des attributs contraints. Si ceux-ci sont finis alors assurer l'arc-consistance est possible, si ceux ci sont infinis et réels, la consistance locale sera moins forte, si ceux-ci sont des ensembles d'ensembles, il faudra axer les efforts sur la représentation des ensembles-valeurs, si ceux-ci sont infinis et d'un type non simple mais ordonné, il faudra axer les efforts sur les propriétés de cet ordre . . .

**Résolution**

Lorsque l'on parle de CSP RCO, il faut distinguer les CSP RCO constitués de contraintes de classe des CSP RCO constitués de contraintes d'instance. Les premiers sont gérés comme autant de CSP RCO à structures identiques qu'il existe d'instances dans les classes contraintes. Il est clair que la résolution d'un CSP RCO à contraintes de classe se traduit par les résolutions de tous ces CSP RCO à structures identiques. Ceci a tendance à rendre le coût de la résolution prohibitif dans le cas de classes contraintes à grande extension. Chercher à compléter une instanciation localement consistance d'un nombre réduit de ces CSP RCO issus d'un CSP RCO à contraintes de classe paraît être un but plus raisonnable qui nécessite de circonscrire les instances à considérer lors de la résolution.

Au cours de la recherche d'une solution, on peut être amené à procéder à l'énumération des domaines d'attributs contraints non valués. Une telle énumération a pour but, soit de créer des instances qui correspondent aux diverses complétions localement consistantes d'une instance contrainte partiellement valuée (raisonnement hypothétique), soit de créer intégralement, à partir des classes, des instances contraintes globalement consistantes (complétion de la base de connaissances). De plus, l'énumération est souvent le moyen ultime de déterminer si une classe contrainte peut effectivement admettre des instances (détection d'incohérence). Enfin, on peut se tourner vers des techniques de maintien de solution pour chercher à atteindre la solution la plus proche en cas de modification d'une solution.

### 9.3.6 Conséquences pour un SRCO contraint

#### Contraintes et mécanismes d'inférence

L'intégration de contraintes dans une RCO ne doit pas remettre en cause la sémantique propre des mécanismes d'inférence que sont l'instanciation, la classification ou l'héritage.

Ainsi, si une instance en cours de création ne satisfait pas une des contraintes de sa classe, on doit considérer que la contrainte est un trait définitionnel de la classe et donc que l'instance n'est pas viable. Avant que de conclure définitivement à un échec de l'instanciation, on pourra chercher à proposer une autre classe d'appartenance en lançant une classification.

Lors de la classification, les contraintes de la classe candidate doivent se contenter d'un rôle prédicatif puisque, par principe[6], on ne peut accepter qu'une valeur de l'instance à classer soit inférée par un mécanisme — ici, une contrainte — de la classe pour laquelle on cherche précisément à valider l'appartenance. Cela n'empêche pas pour autant qu'une propagation de contraintes soit lancée en fonction des informations disponibles dans l'instance afin de juger de son appartenance. Lors de la classification, pour une instance donnée (voir *chapitre 12*) une classe est déclarée *impossible* si une des valeurs des attributs de l'instance ne se trouve pas dans le domaine de l'attribut, *possible* s'il manque des valeurs d'attributs dans l'instance mais si toutes celles qui sont présentes sont dans les domaines associés, *probable* (voire *sûre*) si toutes les valeurs de tous les attributs de l'instance sont présentes et se trouvent dans les domaines associés. En présence des contraintes, décider de l'étiquette à accorder à une classe peut être plus complexe. On peut décider qu'une classe est *impossible* pour une instance si son contenu viole une

6. Voir une discussion sur ce sujet dans le *chapitre 10*.

des contraintes de cette classe, qu'elle est *possible* si le contenu incomplet de l'instance ne viole jusqu'alors aucune contrainte, qu'elle est *probable* si le contenu complet de l'instance satisfait toutes les contraintes de cette classe (et les contraintes d'instance). Cependant, dans le cas d'une instance incomplète, il est tout à fait envisageable que les domaines des attributs de l'instance soient localement consistants, que les valeurs données jusqu'à présent soient consistantes entre elles, mais ne constituent pas une instanciation localement consistante menant à une solution ou, pire encore, qu'il n'existe pas de solution, c'est-à-dire qu'aucun contenu complet de l'instance ne rende cette classe probable. La classe est possible parce que son contenu est incomplet mais correct alors qu'en réalité elle est déjà *impossible* voire sans extension possible. Tout ce problème de décision, qui peut être fortement combinatoire, repose en fait sur les capacités de résolution du système de contraintes.

Vis-à-vis de l'héritage, les contraintes de classe sont héritées par les sous-classes, après avoir pris soin de vérifier par un parcours de bas en haut de la sous-hiérarchie concernée, qu'elles peuvent effectivement l'être.

**Contraintes et cohérence**

Il faut être conscient du fait que la cohérence d'une base de connaissances contrainte est dévolue au niveau de consistance atteint sur les CSP RCO, qui dépend de la discrétisation et de la représentation des domaines contraints. Ainsi, redondances et incohérences introduites par les contraintes ne sont pas toujours détectables statiquement et n'apparaîtront bien souvent que lors d'une phase de résolution. Elles sont une véritable nuisance car conserver des ensembles de contraintes redondants ou inconsistants alourdit considérablement les mises à jour de la base de connaissances en lançant des propagations inutiles. La détection des incohérences ou des redondances peut se faire par l'emploi de méthodes de simplifications de systèmes de contraintes [Lassez *et al.*, 1993] ou par une représentation explicite des incompatibilités de contraintes [Capponi *et al.*, 1995].

## 9.4 Conclusion

Contraintes et objets sont souvent associés afin de bénéficier de leurs qualités respectives. Les objets permettent de structurer des connaissances alors que les contraintes sont un moyen déclaratif d'énoncer et de résoudre des problèmes.

Parmi les systèmes associant objets et contraintes, on trouve tout d'abord des gestionnaires d'interfaces graphiques dans lesquels les contraintes gèrent la cohérence des relations établies sur les objets graphiques, des langages de programmation par objets intégrant des contraintes, des langages de programmation par contraintes permettant de contraindre objets, et des boîtes à outils de programmation par contraintes dans lesquelles variables, contraintes et méthodes de résolution sont décrites en termes d'objets.

Dans les systèmes de représentation de connaissances par objets, les contraintes portent sur des instances et plus généralement sur des attributs d'instances. Elles peuvent concerner toutes les instances d'une classe ou une instance particulière. Le résolveur couplé au SRCO est donc chargé de maintenir la consistance des domaines des attributs contraints, au niveau des classes et au niveau des instances. Dans un contexte de réseaux de contraintes évolutifs, l'ajout et le retrait de contraintes doivent être répercutés au niveau des domaines contraints. Leur gestion est alors soumise à des règles fixant la prépondérance des contraintes vis-à-vis des classes et/ou des instances et peut conduire à réorganiser une base de connaissances par classification.

L'introduction de contraintes dans un SRCO entraîne quelques limitations. Tout d'abord, en présence de contraintes de classe dans des grandes bases de connaissances, il apparaît préférable que la résolution des CSP RCO soit circonscrite à un nombre restreint d'instances. De même, la classification devient incertaine et peut même conduire à la construction de connaissances incohérentes ou redondantes en raison de l'insuffisance intrinsèque des techniques de consistance locale.

Malgré ces limitations intrinsèques, il est indéniable que les contraintes permettent d'augmenter de manière conséquente la déclarativité et les capacités d'inférence du système. Notamment parce que le SRCO peut mettre à contribution le résolveur du module de programmation par contraintes auquel il est couplé dans la maintenance de plusieurs fonctionnalités. Parmi les illustrations des usages collaboratifs qui peuvent être faits des contraintes dans une RCO, citons : le partage de propriétés au sein d'objets composites [Gensel *et al.*, 1993 ; Puig et Oussalah, 1996], le flot de données dans un modèle de tâches [Gensel et Girard, 1992], l'exploration d'une base de connaissances par raisonnement hypothétique, l'expression et la gestion de la sémantique de relations entre objets [Gensel, 1995], et l'expression et la gestion de filtres permettant l'inférence de valeurs d'attributs [Capponi et Gensel, 1997]. Ainsi, introduire des contraintes dans une RCO, c'est permettre l'expression et la maintenance de relations entre objets mais c'est aussi bénéficier de ces contraintes pour accroître l'expressivité et contrôler la sémantique d'autres fonctionnalités du modèle.

Aujourd'hui, les efforts de recherche dans les associations d'objets et de contraintes s'orientent, entre autres, vers les *contraintes génériques* et vers des modules de programmation gérant des CSP de types quelconques. Les contraintes génériques sont une transposition aux contraintes des fonctions génériques de certains langages de programmation par objets [Pelenc et Ducournau, 1997]. Elles permettent de rassembler plusieurs contraintes de sémantique semblable mais portant sur des variables de types différents (par exemple, une contrainte générique d'égalité de points rassemblant les égalités de points dans des espaces à une, deux, ou trois dimensions). La construction d'un module de programmation par contraintes de types quelconques [Capponi et Gensel, 1997], quant à elle, passe par le couplage d'un système de gestion de types avec un module de programmation par contraintes afin de faire en sorte que les règles de consistance et les techniques d'énumération soient imposées par la signature des opérateurs associées aux types des variables contraintes ainsi qu'à l'ordre défini sur ces types. Ces deux axes participent à l'expression de contraintes d'un plus haut niveau d'abstraction.

# Troisième partie

# Objets pour la représentation de connaissance

# 10

# Représentation de connaissance par objets

LES OBJETS SONT UTILISÉS non seulement pour la programmation mais aussi pour la représentation de connaissance. Cette perspective est présentée ici tout d'abord au travers de son histoire mais surtout au travers de ses traits particuliers. Ces derniers sont illustrés par la conception du système TROEPS et ses particularités. Cependant, plus que sur un langage précis, c'est sur la variété (le mot polymorphisme étant interdit ici) qu'insistera ce chapitre. Comme le lecteur peut s'en rendre compte à la lecture de ce livre, pour certains problèmes posés par la conception de systèmes à objets, il existe plusieurs solutions. En ce qui concerne les représentations par objets, les solutions à retenir ne sont pas forcément celles des langages de programmation ou des bases de données.

## 10.1 Introduction

### 10.1.1 Motivations

Le but de la représentation de connaissance est de rendre compte d'un « domaine » particulier de telle sorte que cette représentation soit manipulable par une machine. Elle sert, en particulier, dans les *systèmes à base de connaissance*. Ce but qui dirige une partie des travaux sur les représentations de connaissance ne peut être validé de manière théorique (le « domaine » étant inaccessible à la théorie). Ce but n'est pas non plus distinctif, il peut être revendiqué par d'autres disciplines (bases de données par exemple). Par contre, il est forcément trahi lorsque les objets sont utilisés pour se représenter eux-mêmes (si l'on peut dire que la pile — au sens informatique — implémentée en C++ se représente elle-même, il n'est plus possible d'affirmer quoi que ce soit d'intéressant sur cette représentation : la représentation est forcément correcte puisque l'objet représente ce qu'il est).

Les représentations de connaissance offrent donc des langages permettant de modéliser le « domaine ». Les constructions de ce langage ayant un sens particulier, elles peuvent être manipulées par un ordinateur en respectant ce sens et, par conséquent, la cohérence interne de la modélisation du « domaine ». Ainsi, les employés d'une entreprise, les bureaux et les équipes pourront être représentés par des objets et jamais un bureau ne pourra être utilisé à la place d'une équipe. Dans les développements qui sont présentés ici, et dans d'autres (voir *chapitre 12*), le rapport entre ce qui est représenté et les expressions du langage de représentation de connaissance est analysé au moyen d'une sémantique de ce langage. Celle-ci permet de justifier la validité des opérations du système de représentation de connaissance par rapport au sens des expressions introduites dans le système. Si, par exemple, une base de connaissance décrit la classe des comptables, représentant les employés qui sont des comptables, et la classe des agents administratifs, représentant les employés qui sont agents administratifs, et que la première est sous-classe de la seconde, alors il sera impossible d'être un objet de la classe comptable sans être un agent administratif.

### 10.1.2 Histoire

Historiquement, les modèles généraux de représentation de connaissance ont été nombreux. Par général, on entend un modèle de représentation qui peut être appliqué à toute sorte d'entités (objets physiques, idées, actions, relations ... ) par opposition à un modèle qui se concentre sur un domaine particulier (les relations temporelles, les tâches, les taxonomies biologiques — voir *chapitre 15*). C'est à ce genre de modèles généraux qu'est consacré ce chapitre.

L'un des premiers systèmes de représentation de connaissance générique est le modèle des *réseaux sémantiques* [Quillian, 1968]. Il se caractérise par une organisation de la connaissance sous forme d'un graphe orienté dont les nœuds et les arcs sont étiquetés. Il est doté d'un mécanisme d'inférence : la propagation de marqueurs ("*spreading activation*", poussée à ses limites dans [Fahlman, 1979]) qui consiste à parcourir le graphe dans le sens des arcs à partir d'un nœud précis (ou de deux nœuds). Il permet donc formellement de déduire un ensemble de nœuds accessibles à partir d'un (clôture transitive) ou de plusieurs nœuds. Il est limité car le modèle ne distingue pas *a priori* d'étiquettes particulières alors qu'il serait utile de distinguer certains arcs (par exemple, les relations générique-spécifique) ou certains nœuds (en tant qu'individu ou espèce). Par ailleurs, la signification des arcs et des nœuds n'est pas précisée clairement [Woods, 1975]. Par exemple, l'arc `marié-à` entre le nœud `homme` et le nœud `femme` signifie-t-il qu'un homme est marié à une femme, que tous les hommes sont mariés à toutes les femmes ... ? Bref, le principal grief à l'encontre des réseaux sémantiques est leur absence de sémantique justement.

Le modèle ultérieur est celui des *schémas* ou *frames* [Minsky, 1975 ; Masini *et al.*, 1989] introduit dans le cadre de la représentation de connaissance acquise par la vision. Ce modèle organise la connaissance autour de la notion de schéma : un nom auquel est associé un ensemble d'attributs. Chaque attribut se voit associer à son tour un ensemble de *facettes* permettant de le caractériser. Cette notion de facette inspirée de la métaphore des différentes faces sous lesquelles il est possible de voir un objet a été utilisée à des fins multiples et variées : rendre compte de la position par rapport à un observateur, d'un rôle joué par l'objet ou de différentes acceptions du nom de l'attribut. Les facettes, à leur tour, contiennent des *valeurs*. La *mise en correspondance* (ou *appariement*, "*matching*") est le moyen d'inférence privilégié sur les schémas. Il consiste à comparer deux objets en fonction de leurs facettes et valeurs pour vérifier si elles correspondent ou non (si l'un est plus complet que l'autre ou si les deux peuvent être complétés de manière compatible). C'est un moyen très puissant pour comparer les objets. Par exemple, si un robot voit une voiture rouge garée devant lui et qu'on lui a demandé de se garer derrière une voiture rouge de marque Renault, il peut apparier les deux objets (car ils sont tous les deux rouges et garés, ce qu'il ne peut faire avec les voitures bleues) et poursuivre sur l'hypothèse que c'est derrière l'auto qu'il voit qu'il doit se garer.

Les modèles des schémas et des réseaux sémantiques ont été rationalisés dans un travail de longue haleine par Ronald Brachman (sous l'impulsion de William Woods [Brachman, 1977 ; Brachman, 1979 ; Brachman, 1983 ; Brachman, 1985]). Ce travail isole un certain nombre de problèmes de ces modèles qui doivent être impérativement tranchés. Dans sa thèse [Brachman, 1977] ainsi que dans plusieurs articles publiés plus tardivement [Brachman, 1983 ; Brachman, 1985], Ronald Brachman énonce de précieux principes de bonne utilisation de ces systèmes (qui seront précisés plus loin) :

- ne pas considérer la même entité, tantôt comme un individu (une auto rouge particulière), tantôt comme un ensemble d'individus (la classe des autos rouges) ;
- ne pas confondre propriétés descriptives (les oiseaux doivent avoir des plumes) et propriétés typiques (typiquement les oiseaux volent) ;
- ne pas confondre définition (des facettes sur les attributs exprimant des conditions nécessaires et suffisantes pour appartenir à la classe), description (les conditions sont seulement nécessaires) et prototype (les valeurs données par les facettes sont uniquement typiques).

Cependant, dans le même temps, les développeurs continuent à implémenter de nouvelles idées. Ainsi, vers la fin des années 1970, des systèmes de représentation de connaissance par objets que nous qualifierons de "*post-frames*" voient le jour (KRL [Bobrow et Winograd, 1977] FRL [Roberts et Goldstein, 1977], SRL2 [Wright et Fox, 1983]). Ces systèmes sont inspirés à la fois des schémas et des réseaux sémantiques mais aussi des langages de programmation par objets. Ils sont influencés par les travaux autour des systèmes de fenêtres des langages et machines LISP qui eux-mêmes le sont par SMALLTALK. Ils se caractérisent par l'adoption de la notion de classe (c'est-à-dire qu'ils distinguent les classes des instances) et d'un lien générique-spécifique fort (c'est-à-dire dont le sens est inclus dans le système) ainsi que par une rationalisation de l'utilisation des facettes. Ces dernières sont conservées en nombre restreint (mais parfois extensible) à des fins de typage ou d' inférence des valeurs d'attributs. De plus, la notion d'envoi de message est adoptée par certains et ils utilisent parfois un méta-modèle. Certains d'entre eux conservent cependant une notion de prototype. La programmation par réflexes a, par ailleurs, un certain succès. Ces systèmes ont connu une diffusion importante sous la forme de produits commerciaux qui ont été utilisés dans le monde entier KNOWLEDGE CRAFT [Carnegie Group, 1988], ART [Williams, 1984], KEE [Filman, 1988], NEXPERT OBJET). Ils ont eu des successeurs en France sous la forme de prototypes de recherche (SHIRKA [Rechenmann, 1985], OBJLOG [Dugerdil, 1988] ... ) comme de produits commerciaux (KOOL [Albert, 1984], SMECI [ILOG, 1991b], YAFOOL [Ducournau et Quinqueton, 1986 ; Ducournau, 1991]). Une comparaison de ces systèmes se trouve dans la table 10.1 à la fin de la section 10.2.3. À côté, et parfois au sein, de ces systèmes pour lesquels la notion d'objet est centrale, se sont développés des systèmes permettant d'exploiter conjointement des règles et des objets. Ces systèmes sont soit des extensions de systèmes d'objets existants (par exemple OPUS [Atkinson et Lausen, 1987]), soit des extensions de systèmes de règles existants (par exemple CLIPS).

Parallèlement, des recherches plus fondamentales ont été poursuivies. Dans la lignée des travaux de Ronald Brachman, mais renforcées par un courant formel (Hector Levesque, puis bientôt Alexander Borgida et Bernhard Nebel pour nommer les plus notables), les représentations de connaissance par objets évoluent vers une formalisation croissante qui conduit à définir la sémantique dénotationelle de ces langages (voir *chapitre 12*), à évaluer la décidabilité et la complexité de certains mécanismes de déduction et à concevoir des algorithmes corrects pour ces mécanismes. Un regret peut cependant être formulé : alors que de nombreux choix de langages sont possibles, les chercheurs se focaliseront sur un seul type de langages nommés maintenant logiques de descriptions (voir *chapitre 11*). Dans la même veine, mais plus directement inspiré des réseaux sémantiques, le modèle des *graphes conceptuels* défini par John Sowa [Sowa, 1984] se voit doté d'une sémantique formelle [Chein et Mugnier, 1992].

### 10.1.3 Situation actuelle

Si la sémantique des langages est une préoccupation de certains développeurs de systèmes depuis très longtemps elle a mis du temps à s'imposer en tant que telle. Ainsi, depuis les premiers travaux relativement peu formels de Ronald Brachman (voir ci-dessous) jusqu' aux logiques de descriptions (voir *chapitre 11*), une lente évolution a pris place. Il y a d'abord des systèmes dont les développeurs ont conçu le comportement de la manière la plus raisonnée possible, puis des théories rendant compte de ces systèmes et mettant en évidence leurs déficiences. Par ailleurs, l'effort de formalisation a porté sur une simplification des systèmes existants. Ainsi, de nombreuses opérations et innovations (souvent très utiles) sont laissées dans l'ombre par le travail de formalisation. En conséquence, même les systèmes développés à partir des travaux théoriques offrent des traits qui sortent de la théorie et rendent le système utile. Il est donc important de s'intéresser à toutes les fonctions d'un système de représentation de connaissance et non uniquement à celles qui sont dûment formalisées. Le présent chapitre s'attache à décrire, au travers d'un exemple récent, les fonctionnalités et les problèmes des représentations de connaissance par objets. Les aspects théoriques seront présentés dans les deux chapitres qui suivent.

Actuellement les langages les mieux définis sont, sans conteste, les logiques de descriptions suivies par les graphes conceptuels. Mais les modèles qui ont été les plus utilisés sont les systèmes de représentation de connaissance “*post-frames*” offrant en général plus d’expressivité, mais dont il est difficile de cerner les aspects théoriques. Dans ce qui suit on va s’attacher à présenter ces systèmes. Il existe une manière de les présenter qui laisse une large place aux aspects d’implémentation [Karp, 1993] : faut-il représenter les objets par des vecteurs ou des listes de propriétés, faut-il conserver les classes lors de l’exécution et opérer une manipulation dynamique des objets et des classes ou compiler statiquement les accès aux objets et supprimer les classes lors de la compilation ? Ici on s’attachera plutôt aux principes qui gouvernent les représentations de connaissance par objets, sachant qu’il est inutile de comparer les implémentations autrement qu’en comparant les performances (en temps et en mémoire nécessaires).

Le *chapitre 11* présentera plus en détail les logiques de descriptions alors que le *chapitre 12* présentera les travaux en cours pour décrire de manière formelle ces systèmes de représentation de connaissance par objets. Leurs principes sont d’abord rapidement brossés (§ 10.2) avant de focaliser sur un système particulier : TROEPS. TROEPS sera présenté tout d’abord au travers des choix faits dans certaines alternatives de modélisation (§ 10.3) avant de décrire son langage de description (§ 10.4) et les inférences possibles (§ 10.5).

## 10.2 Principes des représentations de connaissance par objets

Il n’y a pas, à proprement parler, de principes uniques, généraux et inflexibles des représentations de connaissance par objets. On peut cependant tracer à gros traits ce qu’est un système typique. Tout d’abord, la fonction d’un tel système est :

- de stocker et d’organiser la connaissance autour de la notion d’objet ;
- de fournir des services inférentiels de bas niveau destinés à compléter l’information disponible.

La présentation des principes sera donc organisée suivant ces deux axes.

### 10.2.1 Structure

La structure sur laquelle se fonde un système de représentation de connaissance par objets est très proche de celle d’un langage de programmation par objets (voir *chapitres 3 et 2*). À l’instar des langages “*post-frames*”, les objets sont décrits à l’aide de classes organisées dans une hiérarchie de spécialisation. La connaissance est organisée autour de la notion d’objet. Elle est ici brièvement rappelée.

Les objets sont des ensembles de couples attributs-valeurs associés à un identifiant. En général cette identification se fait à l’aide d’un nom (que celui-ci soit extérieur aux couples attributs-valeurs ou la valeur d’un attribut particulier). La valeur d’un attribut peut soit être un objet, soit être une valeur d’un type primitif du langage (par exemple, une chaîne de caractères ou un entier). Elle peut être connue ou pas. Souvent les attributs peuvent contenir une collection (par exemple, un ensemble) de valeurs. Il est très important de définir le sens de telles collections, par exemple le fait que l'attribut `membres` de la classe `projet` soit un ensemble peut signifier qu'il s'agit :

- d'un ensemble fermé d'objets (par exemple l'ensemble des `membres` est exactement ceux connus du système),
- d'un ensemble ouvert d'objets (l'ensemble des `membres` connus du système est inclus dans l'ensemble des `membres` réels),
- d'un ensemble d'objets possibles (on peut alors parler du membre d'un `projet` et celui-ci pourra être l'un des `membres` connus du système).

La première approche est souvent utilisée dans les représentations "*post-frames*" (inspirée des langages de programmation), la seconde dans les logiques de descriptions et la dernière dans les systèmes plus proches de l'esprit des schémas.

Les objets sont regroupés en *classes* qui, à l'instar de celles des langages de programmation par objets (voir *chapitre 3*), permettent de regrouper des traits communs à ces objets. Contrairement au modèle original des schémas, seules les classes associent des facettes (ou descripteurs) aux attributs et seul un ensemble restreint de descripteurs est autorisé et clairement identifié (même si celui-ci est parfois extensible par la programmation — SRL2 [Wright et Fox, 1983], YAFOOL [Ducournau et Quinqueton, 1986 ; Ducournau, 1991]). Dans les systèmes les plus restreints, elles servent principalement à deux objectifs : préciser les valeurs d'attributs admissibles dans une classe (*typage*) et déclarer des mécanismes capables de déduire la valeur d'un attribut manquant (*inférence*). Les systèmes les plus riches offrent un ensemble plus complet de facettes permettant de spécifier des comportements lorsqu'une valeur est écrite ou lue (« programmation dirigée par les accès » [Stefik *et al.*, 1986]) de gérer des relations entre objets (réciproque par exemple [Ducournau, 1991 ; Fornarino et Pinna, 1990 ; Wright et Fox, 1983 ; Albert, 1984]) ou d'assurer la liaison avec le monde extérieur (une interface graphique par exemple [Bobrow et Stefik, 1983 ; ILOG, 1991b ; Ducournau, 1991]). On se limitera ici à ce qui concerne l'aspect représentation de connaissance : typage et inférence.

Les facettes de typage permettent de restreindre les valeurs possibles d'un attribut en précisant les valeurs admissibles par une énumération (`domaine`) ou une réunion d'intervalles, (`intervalles`) introduisant des exceptions (`sauf`)ou restreignant la cardinalité dans le cas de collections. (`cardinal`) Lorsque la valeur d'un attribut peut être un autre objet, les restrictions peuvent se faire soit en précisant les classes auxquelles l'objet doit appartenir (`type`) soit grâce à un filtre permettant de n'accepter que les objets qui satisfont ce filtre (`filtre` — voir ci-dessous). La figure 10.1 présente deux définitions de classes.

Les classes sont organisées par la relation de *spécialisation* (parfois nommée relation d'héritage ou de généricité). La représentation de connaissance par objets privilégie donc au moins ce lien particulier entre les objets. La spécialisation est une relation d'ordre. Dans certains systèmes, le graphe de la réduction transitive de cette relation peut être restreint à un arbre, contraint à être connexe ou à disposer

```
<employé
    attributs = {
         <nom type = chaine ; constructeur = un ;>,
         <prénom type = chaine ; constructeur = un ;>,
         <date-naissance type = date ; constructeur = un ;>,
         <diplômes type = diplôme ; constructeur = liste ;>,
         <projet type = projet ; constructeur = un ;>,
         <chef type = employé ; constructeur = un ;>,
         <subordonnés type = employé ; constructeur = ensemble ;>,
         <bureau type = bureau ; constructeur = un ;>,
         <salaire type = entier ;
                  constructeur = un ;
                  intervalles = { [5000 20000], [23000 23000] };>,
         <régime type = chaine ; constructeur = un ;>};>

 <chercheur
     sorte-de = employé ;
     attributs = {<chef type = { cadre };>};>
```

FIG. 10.1: *Une classe et une sous-classe.*

d'une unique source (racine dans le cas d'un arbre). La sémantique de cette relation est celle de l'inclusion ensembliste : les individus appartenant à l'interprétation d'une classe doivent appartenir à celle de ses super-classes. On dispose donc d'un ensemble de sous-ensembles imbriqués (voir figure 10.2).

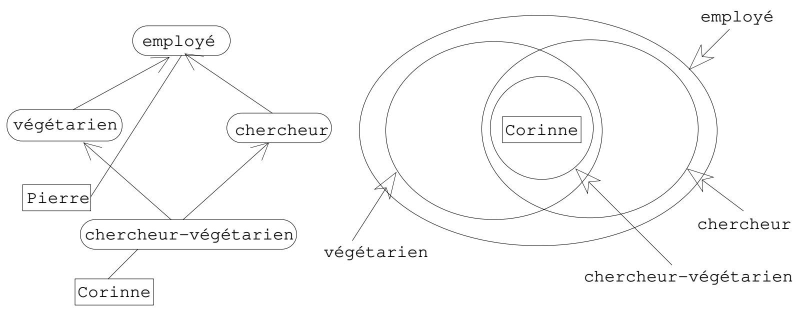



FIG. 10.2: *Signification des relations entre classes. Les boîtes rectangulaires représentent les instances, les boîtes arrondies les classes. Les traits sans flèche représentent l'instanciation et les flèches vides la spécialisation. Le dessin de droite est une représentation ensembliste de la situation. Il faut noter que* `Pierre` *n'y figure pas car si l'on sait qu'il appartient à l'ensemble employé, on ignore s'il fait partie d'un de ses sous-ensembles.*

De cette interprétation de la relation de spécialisation, tout découle : plus une classe est spécifique, plus les domaines de ses attributs doivent être restreints ; si une classe est sous-classe de deux classes, les domaines de ses attributs sont forcément inclus dans (ou égaux à) l'intersection des domaines de ces deux classes ; si un objet est instance d'une classe, il l'est aussi de toutes ses super-classes, etc. Ceci permet de faciliter l'expression du domaine de valeur des attributs (que l'on qualifiera d'*effectif*) en ne précisant dans les sous-classes que les restrictions par rapport à la super-classe (on parlera alors de domaine *exprimé*). Par exemple, s'il y a une contrainte sur le salaire des `chercheurs`, celle-ci pèse aussi sur le salaire des `chercheurs-végétariens` : elle fait partie du domaine effectif de l'attribut salaire de la classe `chercheur-végétarien` mais il est inutile de l'exprimer. Par contre, et c'est l'un des enseignements de Ronald Brachman, il n'est pas question d'introduire des exceptions dans les domaines de classes car cela permettrait inévitablement de définir « un éléphant comme un singe qui ne grimpe pas aux arbres » [Brachman, 1985].

Dans ce cadre (celui de la structure) certains problèmes n'apparaissent pas. C'est le cas des problèmes de conflits dus à la *multi-généralisation* — aussi nommé héritage multiple — , c'est-à-dire le fait d'avoir plusieurs super-classes incomparables par la relation de spécialisation qui sont très discutés dans le monde de la programmation par objets [Ducournau *et al.*, 1995]. C'est aussi le cas des problèmes de non monotonie traités par l'intelligence artificielle. Ainsi, si l'utilisateur exprime que les `végétariens` sont (ont pour valeur de l'attribut `moyens`) `pauvres`, que les `chercheurs` sont `riches` et qu'il existe des individus à la fois `chercheurs` et `végétariens`, il s'est tout simplement trompé quelque part.

### 10.2.2 Inférence

Les représentations de connaissance par objets ne proposent pas forcément de messages ni de méthodes. Lorsqu'ils en offrent[1], leur fonctionnement est le même que dans les langages de programmation par objets (voir *chapitre 3*).

Un des traits qui différencie les langages de programmation des représentations de connaissance par objets est l'existence de mécanismes d'inférence permettant de calculer une valeur pour un attribut lorsque celle-ci n'est pas connue, c'est-à-dire lorsqu'elle n'est pas présente dans l'objet. Ces mécanismes sont attachés aux classes au moyen de facettes d'inférence. Les moyens d'obtenir cette valeur peuvent être très nombreux et toutes les méthodes de l'informatique (fonctions, procédures, règles, tâches, contraintes . . . ) peuvent être mises à contribution. On se bornera ici à évoquer des mécanismes qui s'intègrent bien dans le contexte de la représentation de connaissance par objets. Indépendamment du calcul de la valeur, il est important de déterminer en fonction de quoi celle-ci est calculée. Une des règles en vigueur, idéalement, dans les représentations de connaissance par objets est que la procédure se borne à calculer cette valeur (elle ne fait pas d'effets de bord) et n'est pas influencée par autre chose que par l'objet lui-même (et indirectement par les objets auxquels il fait référence). Cette fonction, au sens informatique, s'interprète comme une fonction, au sens mathématique, de l'ensemble des objets vers le domaine de valeur de l'attribut. La restriction d'absence d'effets de bord est cependant très dif-

1. C'est introduire le LOO dans la bergerie (contribution conjointe de Gilles Bisson — CNRS — et de François Rechenmann — INRIA).

ficile à vérifier et, bien qu'une bonne discipline de conception dans un modèle à objets doive s'y plier, elle peut facilement être mise en défaut.

Afin spécifier les paramètres des mécanismes d'inférence, on utilise des chemins à partir de l'objet sur lequel il porte. Un *chemin* commence par l'objet lui-même (correspondant au `self` des langages de programmation, voir *chapitre 3*) et se poursuit par une suite de noms d'attributs séparés par des indicateurs. Dans TROEPS, l'objet lui-même est dénoté par le symbole « `!` » et les noms d'attributs sont séparés par des « `.` » ou des « `:` » en cas d'attributs multi-valués dont on va faire l'union des valeurs (voir § 9.3.3). Ainsi, `!.chef.subordonnés` désigne la liste des subordonnés de son chef et `!.chef.subordonnés :chef` est un ensemble qui contient pour unique valeur `!.chef` puisqu'il s'agit de l'union des chefs qu'ont les subordonnés de son chef.

Les facettes d'inférence de valeurs d'attributs que l'on trouve communément sont :

***valeur par défaut*** qui permet de fixer une valeur donnée en l'absence d'autres valeurs (ce mécanisme, lorsqu'il est activé, retourne toujours une valeur) ;

***passage de valeur*** qui permet de contraindre la valeur à être celle d'un autre attribut ;

***héritage latéral*** qui permet d'hériter la valeur du même attribut d'un objet référencé par l'objet courant ;

***filtrage*** qui permet de donner pour valeur d'un attribut l'ensemble des objets qui satisfont certaines contraintes ;

***attachement procédural*** qui permet d'exécuter une fonction (prenant pour paramètre des valeurs d'autres attributs de l'objet) retournant la valeur demandée.

Ces facettes seront plus détaillées dans le cadre de TROEPS (voir § 10.4).

À l'opposé des facettes de typage, les facettes d'inférence sont héritées par défaut, c'est-à-dire qu'elles ne fournissent pas la « véritable » valeur de l'attribut recherché mais permettent de suggérer une valeur possible. Par défaut, signifie « si l'on n'a pas d'autres valeurs candidates » : ainsi, si une valeur vient à être affectée à l'objet, la valeur par défaut n'est plus considérée. Cette interprétation pourrait être remplacée par d'autres : valeur la plus probable ou la plus typique. Toutes ces interprétations ne s'appliquent pas de manière brutale : la valeur la plus probable pour une classe n'est pas forcément la plus probable pour ses sous-classes par exemple. C'est pourquoi les facettes d'inférence ne sont utilisées que s'il n'existe aucune méthode applicable définie dans une classe plus spécifique. Par exemple, s'il existe une méthode pour calculer la paye du mois en fonction du `salaire-de-base` et des `heures-supplémentaires` pour la classe `employé`, cette méthode pourra être masquée au profit d'une méthode plus rapide (la paye est le `salaire-de-base` , sans tenir compte des `heures-supplémentaires`) pour les `cadres`.

Par conséquent, l'héritage des mécanismes d'inférence n'est pas absolu : il peut poser des problèmes. Il est, en particulier, lié au phénomène de *non monotonie* : l'ajout de connaissance ne préserve pas nécessairement celle issue des anciennes inférences. Par exemple, dans le cas de la figure 10.3, si `Pierre` est un `employé`, un

```
<employé
     attributs = {
         <salaire-de-base type = entier ; constructeur = un ;>,
         <heures-supplémentaires type = entier ; constructeur = un
;>,
         <salaire
             methode = !.salaire-de-base
                       + !.salaire-horaire
                         * !.heures-supplémentaires ;>,...};>

<cadre
     sorte-de = employe ;
     attributs = {
         <caisse-de-retraite défaut = "caisse-des-cadres" ;>,
         <salaire valeur = !.salaire-de-base ;>,
         <subordonnés
             filtre = <employé
                          attributs = {
                             <projet valeur =
                                 <projet
                                     attributs = {<chef valeur = !
;>};
                                  >;>};>;>,...} ;>
```

FIG. 10.3: *Mécanismes d'inférence. La méthode pour obtenir le salaire dans la classe `employé` consiste à additionner le salaire de base (`!` représente l'objet lui-même et `!.salaire-de-base` la valeur de son attribut `salaire-de-base`) au salaire horaire multiplié par le nombre d'heures supplémentaire. Dans la classe `cadre`, cette méthode n'est plus utilisée : le `salaire` est le `salaire-de-base`. Le filtre permettant d'obtenir la liste des `subordonnés` consiste à rechercher parmi les `employés` ceux dont l'attribut `chef` a pour valeur le `cadre` lui-même.*

salaire particulier est déduit, alors que s'il est un `cadre`, un autre salaire qui peut être différent du précédent est déductible. En cas de multi-généralisation, l'héritage par défaut peut aussi poser un problème puisque plusieurs méthodes peuvent être applicables car aucune n'est en relation de spécificité avec les autres. Ce type de fonctionnement a donné lieu à toute une littérature sur les « conflits de valeurs » [Ducournau *et al.*, 1995] ou exceptions et est l'une des causes de la création de logiques non monotones [Etherington, 1986] et des théories de l'héritage non-monotone [Touretzky, 1986 ; Simonet, 1994].

Il ne faut pas confondre les mécanismes d'inférence des représentations de connaissance par objets et des mécanismes permettant de compléter la base de connaissance de manière implicite (pour éviter à l'utilisateur de le faire puisqu'il est possible de calculer ce qui manque, par exemple, lorsque la `caisse-de-retraite` est calculée — de manière absolue — automatiquement en fonction du statut de la personne). Pour simplifier, une inférence de connaissance implicite est une inférence réalisée à la création de l'objet : si la valeur n'a pas été explicitement fournie c'est qu'elle est implicite ; les inférences des représentations de connaissance par objets sont généralement réalisées lors de l'accès à la valeur : si elle est toujours inconnue, alors il est possible de rechercher une valeur par des moyens valides ou hypothétiques. À notre connaissance, il n'existe aucun système de représentation de connaissance par objets utilisant cette notion de connaissance implicite, mais les utilisateurs utilisent souvent l'héritage par défaut pour inférer de la connaissance implicite. Utiliser le même mécanisme pour deux fonctions différentes est cependant problématique.

À cet ensemble de mécanismes s'ajoutent des mécanismes plus rares. C'est par exemple le cas de la classification où une instance est comparée aux contraintes posées sur les classes d'une taxonomies pour déterminer à quelles classes l'instance peut appartenir. Elle est utilisée dans CLASSIC [Granger, 1988], SHIRKA [Rechenmann, 1988], FROME [Dekker, 1994] ou dans TROEPS (voir 10.4).

### 10.2.3 Variations

Le tableau ci-dessous donne un aperçu de la variabilité des systèmes de représentation de connaissance par objets que l'on peut rencontrer dans la littérature (certains de ces systèmes sont opérationnels, d'autres restent à l' état de prototypes). Il ne prétend pas à l'exhaustivité (ni dans les systèmes, ni dans les critères) ; une vue plus apocalyptique de cette variabilité peut être trouvée dans [Karp, 1993].

Les systèmes sont divisés par des lignes horizontales qui identifient des groupes. Les trois premières lignes, formant le premier groupe, permettent de contraster la variabilité avec des systèmes concurrents moyens — langages de programmation à objets (LPO), logiques de descriptions (LD), bases de données à objets (BDO). Le second groupe contient certains des systèmes "*post-frame*" américains : SRL2 [Wright et Fox, 1983], KRL [Bobrow et Winograd, 1977] et FRL [Roberts et Goldstein, 1977]. Le troisième groupe contient les systèmes commerciaux américains ART [Williams, 1984] et KEE [Filman, 1988]. Puis viennent en quatrième trois systèmes commerciaux français SMECI [ILOG, 1991b], KOOL [Albert, 1984] et YAFOOL [Ducournau et Quinqueton, 1986 ; Ducournau, 1991] et en cinquième des systèmes académiques français : SHIRKA [Rechenmann, 1985], OBJLOG [Dugerdil, 1988], FROME [Dekker, 1994] et TROEPS [Projet Sherpa, 1995]. Le dernier sera beaucoup plus détaillé dans la suite.

Les différents critères de comparaisons des langages et systèmes reprennent les notions présentées plus haut.

- Le premier groupe concerne les aspects généraux des langages — voir § 10.2.1 : y a-t-il des messages ? y a-t-il une distinction entre classe et instance ? y a-t-il un méta-modèle ? y a-t-il la possibilité de faire de la programmation par réflexes ?
- Le second ensemble considère les caractéristiques graphiques de la spécialisation et de l'instanciation (simple ou multiple).
- Le groupe suivant évalue si les attributs sont typés ou non.

TAB. 10.1: *Comparaison des caractéristiques d'un ensemble de systèmes de représentation de connaissance par objets (•= présent ; ○=sous une forme dégradée ; +=extension).*

| | message | classe/instance | metamodele | réflexes | multi-spec. | multi-inst. | types | attach. proc. | filtres | variables | défaut | héritage lat. | symétrique | contraintes | classification | catégorisation | règles | points de vue | commercial |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LPO | • | • | • | ○ | • | | | | | | ○ | | | | | | | | • |
| LD | | • | | | • | • | | | • | • | | ○ | ○ | + | • | • | + | | |
| BDO | • | • | | + | • | | • | | • | | | | | | | | + | ○ | • |
| SRL2 | | | | • | • | | | | • | | • | • | • | | | | | | |
| KRL | | • | | | | | • | | • | • | • | ○ | ○ | ○ | | | | | |
| FRL | | • | | • | | | | • | | | • | | | ○ | | | | | |
| ART | | ○ | • | | • | • | | • | | | • | | • | | | | • | | • |
| KEE | • | • | | | • | | • | • | | | • | | | | | | • | | • |
| SMECI | • | • | | • | | | • | • | | | | • | • | | | | • | | • |
| KOOL | • | • | • | • | | | • | • | | | • | • | • | | | | • | | • |
| YAFOOL | • | ○ | ○ | • | • | | • | • | ○ | • | • | • | • | + | | | + | | ○ |
| SHIRKA | | • | • | • | • | | • | • | • | • | • | | | ○ | • | + | + | | |
| OBJLOG | • | • | ○ | • | • | | • | • | | • | • | • | | ○ | | | • | • | |
| FROME | • | • | | • | • | | • | • | • | | • | | | ○ | • | | + | • | |
| TROEPS | | • | | | | • | • | • | • | • | • | ○ | ○ | • | • | + | | • | |

- Le quatrième groupe énumère différents types de facettes d'inférence possible (attachement procédural, filtre, passage de valeurs, valeurs par défaut, héritage latéral, attributs symétriques et contraintes — voir § 10.2.2).
- Le pénultième groupe considère des mécanismes plus sophistiqués d'inférence non forcément liés à des facettes (classification, catégorisation, règles de production — voir § 10.5) et la notion de points de vue — voir § 10.4.3.
- En dernier lieu on a noté les systèmes connaissant ou ayant connu une diffusion commerciale.

## 10.3 Les problèmes approchés par TROEPS

Comme on peut le voir ci-dessus, et à l'opposé des logiques de descriptions (voir *chapitre 11*), les représentations de connaissance par objets offrent une grande latitude dans la conception du langage. La suite de l'exposé décrit plus en détail le système TROEPS. TROEPS [Mariño *et al.*, 1990 ; Projet Sherpa, 1995] est le successeur du langage SHIRKA (développé par l'équipe de François Rechenmann, à Grenoble, vers 1985 [Rechenmann, 1985]). TROEPS permet de se faire une idée des principales possibilités à considérer. L'évolution de SHIRKA vers TROEPS est exposée plus en détail dans [Euzenat et Rechenmann, 1995]. La présente section étudie quatre grands problèmes des représentations de connaissance qui furent à l'origine de la conception de TROEPS et les solutions que l'on tente d'y apporter.

### 10.3.1 Problème d'évolution

Un problème posé par la structure des représentations de connaissance par objets (et des langages de programmation par objets) est que les objets attachés à une classe ne peuvent pas en changer. Pourquoi changer de classe ? Parce qu'une personne considérée tout d'abord comme un étudiant n'est pas fondamentalement un étudiant, c'est d'abord une personne (en effet, elle peut cesser d'être un étudiant sans cesser d'exister mais elle ne peut cesser d'être une personne). Ce problème relève de la différence entre « être » et « être vu comme ». Il est intéressant de pouvoir considérer la même personne comme un étudiant en maîtrise de japonais lorsque l'on a acquis plus d'informations. Il serait intéressant de considérer cette même personne plus tard comme un employé. De même, la personne en question peut aussi être un footballeur. Il est donc nécessaire de pouvoir la considérer comme tel. Sous l'aspect de ce que l'on appelle la migration d'instance, apparaissent donc trois problèmes distincts :

- un problème d'*enrichissement* de la connaissance sur la personne (ou *complétion*), qui apparaît lorsque l'on dispose de connaissance plus spécifique sur la personne et qui nécessite un affinement de la représentation,
- un problème d'*évolution* de la personne (ou *mise à jour*), qui apparaît lorsque la personne change et qui nécessite une modification de la représentation, et
- un problème de *point de vue* sur la personne (ou *perspective*), qui apparaît lorsque la personne peut être vue sous plusieurs aspects et qui nécessite un changement de représentation.

Seul le dernier point peut être mécaniquement résolu dans les systèmes de représentation de connaissance classique par l'utilisation de la multi-instanciation ou de la multi-généralisation. Cependant, la classe `étudiant` devra être spécialisée en `étudiant-footballeur`, `étudiant-gymnaste` ... puis `étudiant-footballeur en japonais`, `étudiant-footballeur en malgache` ... ce qui n'est pas très satisfaisant [Goodwin, 1979]. D'autres mécanismes particulièrement *ad hoc* (destruction puis recréation de l'objet) permettent de traiter les deux autres problèmes [Banerjee *et al.*, 1987 ; Nguyen et Rieu, 1989]. Ils sont particulièrement utilisés lors de la classification.

TROEPS reprend ce problème à son origine. Il contraint à distinguer entre l'aspect *ontologique* des classes et leur aspect *taxonomique*. Par ontologique on entend ce qui est liéà l'essence même de la notion représentée et par taxonomique ce qui est lié à une classification des objets dépendant d'un point de vue extérieur. Il sépare alors la notion de concept (ontologique, voir § 10.4.1) de la notion de classe (taxonomique, voir § 10.4.4). Ainsi, un étudiant est défini ontologiquement en tant que personne et non plus en tant qu'étudiant. Cependant, il peut exister une taxonomie professionnelle permettant de classer cette personne parmi les étudiants, puis les étudiants en japonais. Ainsi, le problème de complétion est résolu. Le problème de perspective connaît aussi un début de solution : dés lors que les objets sont créés hors de leur classe, il est possible de définir plusieurs taxonomies (par exemple, profession, sport) sur un même concept (celui des personnes). Un même objet peut alors être attaché à plusieurs classes pourvu qu'elles appartiennent à des taxonomies différentes.

### 10.3.2 Problème d'interprétation des classes et de la spécialisation

Un problème important signalé par William Woods [Woods, 1975] est celui de l'interprétation des classes : que signifie une classe, un objet, un lien entre classes ou objets ? Ce problème est en partie tranché par la distinction classe-instance, mais pas complètement.

Les classes peuvent être interprétées de manière soit *descriptive* , soit *définitionnelle* — aussi qualifiée de prescriptive. Une classe descriptive introduit les conditions nécessaires pour qu'un objet soit une instance de la classe, alors qu'une classe définitionnelle décrit des conditions nécessaires et suffisantes. Ceci est crucial pour la classification (voir § 10.5.2) : dans un système définitionnel, en général, la classification détermine (lorsque les objets ont des valeurs pour tous leurs attributs) les classes auxquelles un objet appartient, alors que, dans un système descriptif, elle détermine les classes auxquelles un objet *peut* appartenir. Dans les logiques de descriptions (voir *chapitre 11*), la distinction entre concept primitif et concept défini, tout d'abord fondée sur le fait que les concepts définis le sont en fonction d'autres alors que les concepts primitifs ne peuvent être définis, correspond sémantiquement à la distinction entre classe descriptive et classe définitionnelle. À l'opposé, le caractère descriptif ou définitionnel des langages de programmation par objets n'a pas été clairement établi (voir, par exemple la discussion dans [Davis, 1987]) : même si les classes sont considérées comme descriptives, dès qu'un mécanisme de classification automatique est introduit, il repose souvent sur une interprétation définitionnelle.

Il y a deux intérêts à disposer de classes descriptives.

- Tout d'abord, il y a beaucoup de classes descriptives à modéliser (soit parce qu'il n'existe pas de conditions suffisantes, soit parce que l'on n'est pas capable de les exprimer) et il faut donc avoir les moyens de le faire.
- Ensuite, tout résultat d'inférence valide dans un monde descriptif est valide dans un monde définitionnel. Les inférences pourront donc y être faites en toute sécurité.

En effet, l'interprétation descriptive est plus faible que l'interprétation définitionnelle : le résultat des inférences est plus restreint même si plus de notions peuvent être représentées. Ainsi, si une tasse est caractérisée comme un contenant avec une anse, alors, d'un point de vue définitionnel, il est possible d'inférer qu'un contenant avec une anse est une tasse. D'un point de vue descriptif, il est seulement possible d'inférer que celui-ci peut être une tasse, mais est-ce bien une tasse ?

TROEPS, issu de SHIRKA, s'inscrit dans une tradition héritée des langages de programmation par objets dans laquelle les classes sont descriptives (il est possible de décrire deux classes ayant les mêmes propriétés et une instance d'une de ces classes ne sera pas instance de l'autre). La popularité des langages de programmation par objets atteste que cette sémantique est bien comprise des utilisateurs actuels des objets. Par ailleurs, il y a de nombreux « domaines » dans lesquels il est possible d'exprimer des conditions nécessaires mais très difficile d'exprimer des condition suffisantes (voir *chapitre 15*). Pour ces raisons, TROEPS est, pour l'instant, restreint à une interprétation descriptive des classes.

### 10.3.3 Problème de nommage et d'identité

L'attachement d'un objet à plusieurs classes incomparables pose un problème connu dans les systèmes à multi-généralisation : le conflit de nom [Ducournau *et al.*, 1995]. Ce dernier s'énonce comme suit : si deux attributs différents peuvent être nommés de la même manière dans deux classes différentes et que l'objet appartient à ces deux classes, il y a conflit. Il faut noter que le problème des conflits de nom est un problème à double tranchant : soit l'on considère qu'un attribut ne peut être nommé qu'une fois dans une base et on se prive de la possibilité d'utiliser son nom dans des classes incomparables dont on sait qu'elles ne partageront jamais une instance (on se prémunit par avance de « trop de conflits ») ; soit on décide que deux attributs nommés de la même façon, introduits dans deux classes incomparables, ne sont pas les mêmes attributs et alors on multiplie les attributs (on passe à côté de conflits réels) [Dekker, 1994].

Le problème est résolu dans TROEPS en ramenant la définition des attributs et leur nommage au niveau du concept. Ainsi, toutes les classes de ce concept doivent utiliser le nom défini au niveau du concept pour nommer l'attribut et ce nom ne peut nommer un autre attribut. Par exemple, le concept `projet` et le concept `employé` peuvent avoir un attribut nommé `nom`, il n'y a aucune ambiguïté sur le fait qu'il ne s'agit pas du même attribut. Par contre, dans la classe `chercheur` et dans la classe `administrateur`, l'attribut `nom` est le même et n'a rien à voir avec celui d'un `projet`.

Au problème des noms s'ajoute le problème d'*identité* des objets. L'identité est aussi du ressort de l'aspect ontologique, c'est-à-dire du concept. Les objets de TROEPS sont nommés au niveau du concept et ce dernier se charge de l'identité des objets, c'est-à-dire qu'il garantit que deux objets différents ne peuvent être identifiés de la même manière et qu'il se charge de retrouver l'objet si l'identificateur est valide. Ceci permet de nouveau de se libérer des conflits de noms au sein d'un même concept et de donner le même nom à deux objets différents, pourvu qu'ils soient dans deux concepts différents. Les bases de données à objets ont promu la notion d'identifiant d'objet par opposition à la *clé* qui permettait de trouver les $n$-uplets dans les bases relationnelles. Dans TROEPS, la décision d'identifier les objets par une clé a été prise afin de disposer de noms plus naturels pour l'utilisateur que les identifiants engendrés par le système ou qu'un nom dont l'utilisateur avait du mal à gérer l'unicité. Les objets sont donc nommés par l'intermédiaire d'un sous-ensemble de leurs attributs (valides pour tout le concept et pouvant varier d'un concept à l'autre).

### 10.3.4 Problème d'inférences

Un dernier problème délicat est celui du statut des mécanismes d'inférence permettant de calculer une valeur d'attribut non disponible dans l'objet lui-même. Traditionnellement, dans les systèmes de représentation de connaissance par objets, les mécanismes d'inférences sont attachés aux classes et permettent d'obtenir une valeur pour les attributs des objets appartenant à la classe. Par ailleurs, ces mécanismes sont « hérités par défaut », c'est-à-dire que le mécanisme calculant une valeur dans la classe la plus spécifique est réputé fournir la meilleure valeur. Dans les systèmes n'autorisant pas la multi-généralisation, il est possible de définir un ordre stable sur les valeurs obtenues. Mais un autre problème apparaît pour la classification : a-t-on le droit d'utiliser les mécanismes d'inférence pour déterminer une valeur qui va être ensuite utilisée lors de la classification ? Le mécanisme ne sera pas forcément adapté (car on ne sait pas si l'instance en cours de classification appartient à la classe) ni le plus adapté (car peut-être appartient-elle à une classe plus spécifique qui possède un mécanisme plus adapté). Sur ce sujet, on peut consulter l'important travail de Lenneke Dekker [Dekker, 1994] qui conduit à distinguer les ensembles d'attributs définissant (au sens de « définitionnels pour ») une classe des autres (descriptifs).

Pour résoudre ce problème, François Rechenmann [Rechenmann, 1993] propose la notion de *détachement procédural*. Le détachement procédural permet d'invoquer des procédures auxquelles sont passés des arguments définis à partir des attributs de l'objet et qui retournent la valeur d'un attribut particulier. Il est possible de proposer plusieurs méthodes dans un concept pour calculer le même attribut (à partir d'arguments différents par exemple), mais ces procédures doivent retourner la même valeur. Une attitude médiane adoptée dans TROEPS fait coexister les deux approches :

- Le détachement procédural retourne des valeurs qui sont bien les valeurs implicites des attributs.
- Des mécanismes d'inférence hérités par défaut sont associés aux classes. Ils retournent des valeurs hypothétiques qui ne sont pas utilisés lors de la classifications et sont activés par une primitive spécifique (`tr-guess-value`).

## 10.4 Description de TROEPS

TROEPS [Mariño *et al.*, 1990 ; Projet Sherpa, 1995] est donc un système de représentation de connaissance par objets développé au début des années 90 par le projet SHERPA de l'INRIA Rhône-Alpes[2].

Par rapport à d'autres systèmes de représentation de connaissance par objets, TROEPS peut se caractériser par quatre points :

- l'ensemble des objets est partitionné en concepts ;

2. TROEPS (prononcer « trop' ») fut désigné sous le nom de « TROPES » de 1990 à 1997. Le nom a été changé en « TROEPS » pour des raisons juridiques.

Le logiciel, ainsi que sa documentation, est disponible sur le serveur ftp de l'INRIA Rhône-Alpes (`ftp.inrialpes.fr`) dans le catalogue `/pub/sherpa/logiciels` (voir aussi `http://co4.inrialpes.fr`).

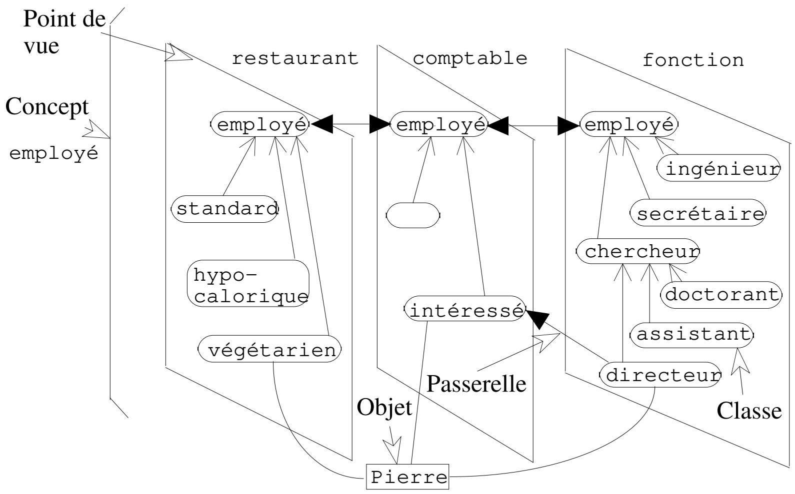



FIG. 10.4: *Les différentes sortes d'entités de* TROEPS. *Le concept* `employé` *peut être vu sous les points de vue* `restaurant`, `comptable` *et* `fonction`. *Chacun d'entre eux détermine une hiérarchie de classes dont la racine est nommée* `employé`. *Par exemple, sous le point de vue* `fonction`, *l'ensemble des employés est décomposé selon leur fonction. L'objet* `Pierre` *est attaché à la classe* `végétarien` *sous le point de vue* `restaurant`, *à la classe* `intéressé` *sous le point de vue* `comptable` *et à la classe* `directeur de recherche` *sous le point de vue* `fonction`.

- un concept peut être vu sous plusieurs points de vue ;
- chaque point de vue détermine une hiérarchie de classes ;
- les rapports d'inclusion entre classes de différents points de vue peuvent être exprimés à l'aide de passerelles entre ces classes.

Dans ce contexte, un objet est instance d'un seul concept, est visible sous plusieurs points de vues où il est attaché à une seule classe plus spécifique.

La présentation de TROEPS suit les différentes étapes de la conception d'une base de connaissance à objets (voir aussi *chapitre 4*) :

1. inventaire des concepts et de leurs instances (concepts et clés) ;
2. inventaire des relations entre concepts (attributs composés) ;
3. inventaire des autres attributs ;
4. inventaires des points de vue sur les concepts (et attributs visibles) ;
5. construction des taxonomies dans les points de vue (et contraintes sur les attributs) ;
6. inventaire des relations entre points de vue (passerelles).

Bien entendu, cet ordre n'est pas immuable d'autant plus qu'il est possible d'ajouter à tout instant de nouveaux attributs et points de vue.

```
<employé
    cle = {
        <nom type = chaine ; constructeur = un ;>,
        <prénom type = chaine ; constructeur = liste ;>,
        <date-naissance type = date ; constructeur = un ;>};
    attributs = {
        <salaire type = entier ; constructeur = un ;>,
        <diplômes type = diplôme ; constructeur = liste ;>,
        <projet type = projet ; constructeur = un ;>,
        <chef type = employé ; constructeur = un ;>,
        <subordonnés type = employé ; constructeur = ensemble ;>,
        <bureau type = bureau ; constructeur = un ;>};
    contraintes = { !.chef = !.projet.chef };>
```

FIG. 10.5: *Description du concept `employé`. Elle contient trois parties : les deux premières définissent les attributs dont le premier sous-ensemble est la clé (ici `nom`, `prénom` et `date-naissance`); la partie suivante décrit les contraintes portant sur les attributs (pouvant lier plusieurs attributs).*

### 10.4.1 Concepts

Un des principes nouveaux de TROEPS est le *concept* (à l'état embryonnaire dans SHIRKA sous le nom de *famille*). L'ensemble des objets présents dans une base TROEPS est partitionné en un ensemble de concepts (un objet fait partie d'un et un seul concept). Le concept rassemble l'aspect ontologique des objets, c'est-à-dire qu'il définit :

- La structure des objets : les attributs qu'il peut avoir sont décrits et typés sous forme d'attributs de concepts (voir § 10.4.2).
- L'identité et l'intégrité des objets : un sous-ensemble de ces attributs forme la clé qui doit être unique et qui permet d'identifier l'objet. Le système garantit cette unicité comme il garantit que cette clé ne sera pas modifiée lors de la vie de l'objet.
- L'espace des noms d'attributs, d'objets et de points de vue.
- Des mécanismes d'inférence qui seront développés au § 10.5.

Ainsi, un `employé` est doté des attributs `nom`, `prénom`, `date-naissance`, `salaire`, `projet`, `diplômes`, etc. Il est identifié par ses `nom`, `prénom` et `date-naissance`.

### 10.4.2 Attributs de concepts et types

L'ensemble des *attributs* applicables aux instances du concept sont définis dans celui-ci. Il est possible d'ajouter des attributs de concepts à n'importe quel moment. Les attributs définis dans le concept sont applicables à n'importe quel objet du concept même si celui-ci est attaché à des classes qui ne les mentionnent pas. Chaque attribut est décrit au niveau du concept par son type et son constructeur.

Le *type* d'un attribut est un concept ou un type externe au système TROEPS. Il n'y a pas, dans TROEPS, de type primitif : tous les types sont externes et sont déclarés au système par un prédicat d'appartenance, un prédicat d'égalité entre deux valeurs, une fonction de lecture et d'écriture et, si le type est ordonné ou discret, un prédicat d'ordre ou une fonction successeur. Ceci permet à l'utilisateur d'importer n'importe quel type (comme entier mais aussi date ou séquence nucléotidique) dans le système tout en voyant ce type pris en compte par des mécanismes tels que la vérification des types des attributs ou la classification des objets (voir § 10.5.2).

Le *constructeur* d'un attribut permet de préciser si la valeur doit être une valeur unique ou une collection de valeurs (un ensemble ou une liste par exemple).

Ainsi, les quatre premiers attributs de la figure 10.5 sont typés respectivement par une chaîne de caractères, une liste de chaînes de caractères, une date et un entier. Les deux attributs suivants sont typés respectivement par un objet du concept `projet` et une liste d'objets du concept `diplôme`.

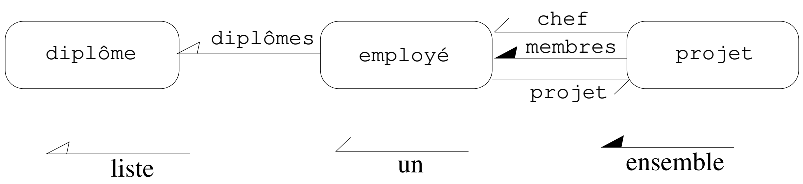



FIG. 10.6: *Relations entre concepts.*

Des *contraintes* peuvent aussi être attachées aux concepts de manière à exprimer les relations entre attributs. Ces relations sont exprimées par un prédicat (ici =) entre des objets atteignables à partir de l'objet courant par le moyen de chemins (voir § 10.2.2). Ainsi, les facettes `herit-by` et `lien-inverse` de YAFOOL (voir *chapitre 14*) peuvent-elles être exprimées par les contraintes :

```
chef herit-by projet <=> !.chef = !.projet.chef
époux lien-inverse épouse <=> !.époux.épouse = !
```

### 10.4.3 Point de vue

Un *point de vue* est une structure destinée essentiellement à accueillir une taxonomie de classes sur les objets du concept. Il décrit donc cette taxonomie en commençant par sa classe racine puis en y ajoutant des sous-classes. Les taxonomies sont de strictes hiérarchies (c'est-à-dire qu'elles n'autorisent pas la multi-généralisation). La présence de plusieurs taxonomies dans un même concept rend compte plus simplement de nombreuses situations dans lesquelles les langages de programmation par objets utilisent la multi-généralisation (voir figure 10.7).

Les attributs pertinents pour un point de vue sont ceux dont il est fait mention au sein des classes du point de vue. Un objet est attaché à exactement une classe par point de vue. Par conséquent, les taxonomies sont supposées *exclusives* (c'est-à-dire qu'un objet ne peut appartenir à des classes incomparables) et tous les objets du

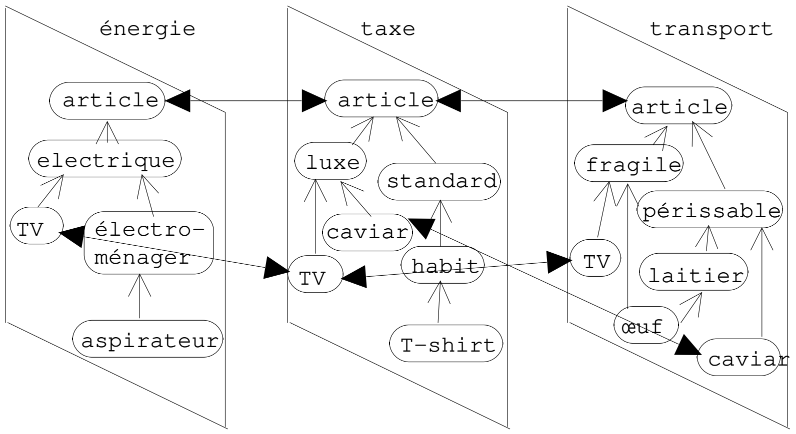



FIG. 10.7: *L'exemple 2.5 (héritage multiple) de [Masini* et al.*, 1989] repris en* TROEPS. *Comme on le voit sur cette figure, il reste un cas de multi-généralisation qui ne se traite pas avec des points de vue.*

concept appartiennent à la racine de chaque point de vue, ce qui justifie que toutes les racines portent le nom du concept.

La notion de point de vue rappelle celle de vue dans les bases de données relationnelles (reprises actuellement dans le cadre des bases de données à objets, voir *chapitre 5*). En effet, les deux notions permettent d'identifier un ensemble d'attributs pertinents et de masquer les autres. Elle sont cependant différentes sous deux aspects :

- Les *vues* des bases de données permettent d'agréger plusieurs relations dans une nouvelle relation (la vue), ce que TROEPS ne pourrait permettre qu'au prix de l'utilisation additionnelle de contraintes.
- Les points de vue de TROEPS permettent de disposer d'une taxonomie différente sous chaque point de vue, ce que les bases de données n'autorisent pas. Cette dernière caractéristique permet de traiter de nombreuses applications qu'il est difficile de mettre en œuvre autrement.

Les langages de programmation par objets utilisent des classes purement ontologiques alors que les logiques de description manipulent des classes taxonomiques. Cependant, ROME [Carré, 1989 ; Dugerdil, 1988] fut le premier système à séparer explicitement les deux aspects. Un objet est alors instance d'une *classe d'instanciation* (rassemblant la description ontologique de l'objet) et permet de la classer sous des *classes de représentation* (correspondant plus à l'aspect taxonomique). Mais ROME ne sépare pas la base en points de vue différents : il n'y a qu'une taxonomie. Par contre, VIEWS [Davis, 1987] offre la possibilité d'organiser la connaissance sous différents points de vue, mais la notion de taxonomie n'est pas centrale dans ce modèle. Par ailleurs, la partie ontologique est présentée comme du « platonisme inutile », c'est-à-dire que l'auteur s'oppose à la possibilité de décrire *a priori* les caractéristiques des objets (mais cette opposition est présente ailleurs dans ce livre).

FRAMETALK [Rathke, 1993] est une implémentation de schémas en CLOS qui offre une notion similaire aux points de vue [Rathke et Redmiles, 1993].

### 10.4.4 Classes et attributs de classes

Les *classes* décrivent un ensemble d' objets particuliers. Pour cela, elles définissent les attributs pertinents pour ces objets et un ensemble de restrictions sur les types des attributs. Cela ne signifie pas que les attributs non pertinents ne sont pas associés aux objets, mais que la classe est indépendante de ces attributs (par exemple, sous le point de vue `restaurant`, l'attribut `projet` n'est pas pertinent, ce qui ne signifie pas qu'un employé ne fasse pas aussi partie d'un projet).

Les classes sont définies en référence à leur super-classe. Toute classe peut préciser les types des attributs en introduisant des *attributs de classes*. Ceux-ci sont décrits par un ensemble de descripteurs (correspondant aux facettes des représentations de connaissance par objets). Les descripteurs de TROEPS sont :

`type` (pour les objets) qui introduit un ensemble de classes auxquelles les valeurs de l'attribut doivent appartenir (ces classes obéissent à la syntaxe `<classe>@ <point de vue>`) ;

`intervalles` (pour les types ordonnés) qui introduit une réunion d'intervalles de valeurs acceptables ;

`domaine` qui introduit un ensemble de valeurs acceptables ;

`sauf` qui introduit un ensemble de valeurs non acceptables pour l'attribut ;

`cardinal` (pour les collections) qui introduit un intervalle de cardinaux acceptables pour la valeur.

```
<employé
    attributs = {
        <nom>,
        <prénom>,
        <date-naissance>,
        <poste intervalles = {[1 266]} ;>} ;>,

<cadre
    sorte-de = employé@fonctionnel ;
    attributs = {
        <diplômes cardinal = [2 +inf[ ;>,
        <chef type = { cadre@fonction } ;>,
        <poste intervalles = {[1 72]}; sauf = {1, 42, 53, 54} ;>,
        <salaire intervalles = { [10000 20000] } ;>} ;>
```

FIG. 10.8: *Description de classes en* TROEPS. *La première classe,* `employé`*, n'est pas le concept décrit à la figure 10.5, mais la racine du point de vue* `fonction`*. C'est pour cela qu'elle n'a pas de super-classe (*`sorte-de`*).*

La grammaire des expressions de type pour les attributs de classe s'exprime comme ci-dessous (les $i$ sont des entiers, les $v_i$ des valeurs et les $c_i$ des classes sous la forme `<classe>@<point de vue>`) :

$$\begin{cases} domaine = \{v_1, \ldots\ v_n\}; \\ sauf = \{v_1, \ldots\ v_n\}; \\ intervalles = \{[v_1\ v_1'], \ldots\ [v_n\ v_n']\}; \\ type = \{c_1, \ldots\ c_n\}; \\ cardinal = [i\ i']; \end{cases}$$

Les autres descripteurs sont destinés aux inférences et sont décrits ci-dessous (voir § 10.5). Des contraintes sont par ailleurs associées aux classes que leurs instances doivent respecter.

### 10.4.5 Passerelles

Les *passerelles* permettent d'exprimer des contraintes entre les classes de points de vue différents. Ainsi, si les points de vue sont théoriquement indépendants, il est possible que l'appartenance à une classe sous un point de vue implique l'appartenance à une autre classe sous un autre point de vue. Par exemple, la passerelle entre `directeur` et `intéressé` dans la figure 10.5 indique que les employés qui sont directeurs de recherche sont forcément intéressés aux bénéfices.

### 10.4.6 Objets

Les *objets* sont semblables aux instances des langages de programmation par objets. Ce sont des ensembles de couples attributs-valeurs. Ils sont identifiés par le concept dont ils sont instance et un sous-ensemble de leurs attributs (défini pour chaque concept) formant la clé. Par ailleurs, les objets de TROEPS, s'ils appartiennent à un et un seul concept, sont attachés à une (et une seule) classe par point de vue. Les contraintes qui pèsent sur un objet sont donc celles du concept et celles qui concernent toutes les classes auxquelles l'objet appartient.

Les objets peuvent être manipulés pour changer leurs valeurs d'attributs ou leurs classes d'appartenance.

## 10.5 TROEPS : inférence

Parmi les services que peut offrir une représentation de connaissance, on s'attachera à l'aspect complétion uniquement. La complétion permet de compléter la base de connaissance automatiquement de manière à ce qu'elle soit plus fidèle au « domaine » modélisé.

Il existe deux types principaux de complétion dans les représentations de connaissance par objets : l'inférence de valeur qui permet d'obtenir une valeur pour un attribut et l'inférence de position qui permet de déterminer une place possible (pour un objet ou une classe) dans la hiérarchie des classes. Ces deux aspects sont bien entendu étroitement liés puisque l'inférence de position se fait sur la foi des valeurs d'attributs alors que l'obtention de valeurs d'attributs dépend de mécanismes attachés à la classe.

Par ailleurs, il existe deux statuts pour ces complétions : soit elles sont considérées comme *valides* (c'est-à-dire qu'elles sont toujours vraies), soit elles sont considérées comme des *hypothèses* (voir les discussions sur les valeurs par défaut au § 10.2.2).

Dans TROEPS, le rapport entre les différents mécanismes d'inférence est clarifié par deux propositions : (1) les inférences valides sont définies dans les concepts (ontologiques) et les inférences hypothétiques le sont dans les classes ; (2) seules les inférences valides — donc indépendantes de la classe — peuvent être utilisées lors de la classification — et les classes d'attachements sont utilisées pour les inférences hypothétiques. Il y a donc une stricte hiérarchie des mécanismes d'inférence lié à l'aspect ontologique/taxonomique du couple concept/classes.

### 10.5.1 Inférence de valeurs

Les inférences de valeurs doivent donc être distinguées suivant qu'elles sont nécessaires à l'objet (on parle aussi de connaissance implicite) ou qu'elles sont des hypothèses sur la valeur de l'objet (plus ou moins étayées en fonction de la situation de l'objet). Les premières sont décrites au niveau du concept au titre des aspects nécessaires des objets, alors que les autres sont attachées aux classes.

Dans les deux classes d'inférence, les paramètres nécessaires aux méthodes sont spécifiés par des chemins qui permettent d'indiquer n'importe quel objet atteignable à partir de l'objet de départ (voir § 10.2.2).

Les inférences attachées aux concepts ressortent de l'idée de détachement procédural (voir § 10.3.4). Les inférences attachées aux classes sont plus diversifiées ; elles sont introduites à l'aide des descripteurs suivants :

`défaut` spécifie une valeur prise par défaut (c'est-à-dire si aucune valeur n'est connue) pour l'objet ;

`affecte` spécifie un chemin par lequel la valeur de l'attribut peut être atteinte ;

`méthode` spécifie un ensemble de méthodes qui peuvent être utilisées pour calculer une valeur (à noter que, contrairement à ce qui se passe pour les concepts, les valeurs retournées ne sont qu'hypothétiques et peuvent parfaitement être différentes d'une méthode à une autre).

`filtre` (pour les collections) spécifie un filtre (voir plus bas) dont l'ensemble des valeurs le satisfaisant formera la valeur de l'attribut. Les chemins sont aussi utilisés pour spécifier des contraintes sur le filtre.

Ces valeurs hypothétiques ne sont soumises à l'utilisateur de la base que si cela est explicitement demandé (elles ne se substituent pas automatiquement aux valeurs d'attributs). La primitive invoquant ces valeurs hypothétiques retourne l'ensemble des valeurs obtenues. En effet, l'appartenance d'un objet à plusieurs classes peut conduire à l'existence d'une certaine valeur sous un point de vue et une autre sous un autre point de vue. Par exemple, sous une interprétation probabiliste, ces deux valeurs peuvent être aussi probables l'une que l'autre.

Les méthodes d'inférences, définies dans les classes, sont héritées d'une classe vers l'autre. Mais, alors que l'héritage des descripteurs de types est strict (voir § 10.2.1), celui des mécanismes d' inférence se fait par défaut (voir § 10.2.2). C'est-à-dire qu'un mécanisme capable de retourner une valeur hypothétique est utile tant qu'il n'y en a pas, dans les sous-classes, qui soit capable de retourner une valeur. Ainsi, pour obtenir la valeur d'un attribut d'un objet, TROEPS cherche les mécanismes susceptibles de l'inférer dans ses classes d'attachement puis, tant qu'aucune valeur n'a été trouvée dans un point de vue, dans les super-classes. Si une autre valeur d'attribut est nécessaire, comme paramètre d'un mécanisme d'inférence (obtenu par un chemin), elle est inferée de la même manière.

### 10.5.2 Classification

L'inférence de position pour un objet est réalisé par le mécanisme de *classification* — plus précisement de classification d'instance. Le mécanisme de classification disponible en TROEPS est particulier, car il manipule des classes descriptives (c'est-à-dire constituées de conditions nécessaires mais non suffisantes, voir § 10.3.2). Classer une instance $i$ sous une classe $c$, c'est trouver les sous-classes de $c$ auxquelles $i$ peut appartenir compte tenu des valeurs des attributs de $i$. Une partition des classes en trois ensembles est alors obtenue ($\tau_{f,c}$ est le type de l'attribut $f$ dans la classe $c$, valeur?($i$,$f$) est la valeur de l'attribut $f$ pour l'instance $i$ et $\iota$ dénote la valeur inconnue) :

**Possibles** : si toutes les valeurs d'attributs de l'instance satisfont les contraintes de la classe : $\forall f \in attributs(c), valeur?(i, f) \in \tau_{f,c}$,

**Inconnues** : si aucune valeur d'attribut de l'instance ne viole une contrainte de la classe mais certaines valeurs d'attributs sont inconnues : $\forall f \in attributs(c)$, $valeur?(i, f) \in \tau_{f,c} \cup \{\iota\}$,

**Impossibles** : s'il existe une valeur d'attribut de l'instance qui viole une contrainte de la classe : $\exists f \in attributs(c), valeur?(i, f) \notin \tau_{f,c} \cup \{\iota\}$.

La classification de TROEPS possède un certain nombre de traits notables :

- Elle prend en compte l'incomplétude des objets (le classement d'instances incomplètes apparaît dans l'ensemble *Inconnues*).
- Parallèlement aux logiques de descriptions (voir le *chapitre 11*) qui ne classifient pas dans les concepts primitifs (correspondant à l'interprétation descriptive des classes) — sauf cas très particuliers (voir *chapitre 12*) —, et comme SHIRKA, TROEPS dispose d'un mécanisme de classification agissant uniquement dans des concepts primitifs. Cependant, la classification est ici restreinte aux individus, elle correspond donc à une opération beaucoup plus simple que la classification dans les logiques de descriptions qui classe des concepts (se rapprochant plus des classes).

- L'algorithme prend en compte récursivement la « classifiabilité » des objets placés en valeurs d'attributs dans la classe caractérisant le type de l'attribut, c'est-à-dire la possibilité pour un objet valeur d'attribut d'être classé dans les classes définissant le type de l'attribut. Ainsi, si un `ingénieur-informaticien` est un `employé` avec un diplôme d'ingénieur travaillant dans un `projet-informatique`, la classification, pour classer un `employé` comme `ingénieur-informaticien`, doit classer la valeur de son attribut `projet` comme `projet-informatique`.
- L'algorithme prend en compte les points de vue et les passerelles. Il opère la classification simultanément dans chaque point de vue. Les passerelles interfèrent dans ce processus en introduisant des contraintes sur les classes d'appartenance.

La classification de classes, autre inférence de position, bien que possible et spécifiée, n'est pas encore disponible dans TROEPS.

### 10.5.3 Filtres

Une classe est une description d'un ensemble d'objets. Ainsi, tout objet satisfaisant la description d'une classe peut y être attaché, mais rien ne l'y contraint. En particulier, il peut exister deux classes incomparables pour la spécialisation dont la description est identique. Ces descriptions constituent donc des conditions nécessaires mais non suffisantes à l'appartenance à la classe.

Les filtres sont formés des mêmes éléments que les classes mais constituent des conditions nécessaires et suffisantes à l'appartenance. Ils sont définis à partir de plusieurs classes appartenant à des points de vue différents auxquels sont ajoutées des contraintes. Ils ne peuvent alors filtrer (et se voir attacher) que les objets attachés aux classes à partir desquelles ils sont définis. La figure 10.3 présente ainsi un filtre permettant d'obtenir l'ensemble des subordonnés d'un employé en recherchant ceux qui l'ont pour chef.

Les filtres sont utilisés pour deux usages : (1) la recherche d'un ensemble d'instances d'une classe satisfaisant un certain nombre de critères et, en particulier, (2) l'inférence de valeur (associée au descripteur filtre décrit plus haut). Dans le premier cas, les filtres sont alors créés et manipulés individuellement, l'ensemble des objets filtrés, alors attachés au filtre, peut être manipulé en tant que tel. C'est ainsi que sont construits les filtres dans l'interface d'interrogation de TROEPS. Dans le second cas, la valeur retournée (qui est toujours un ensemble d'objets) est considérée comme la valeur (hypothétique) de l'attribut.

### 10.5.4 Catégorisation

L'objet de la *catégorisation* est l'obtention d'une taxonomie de classes (décrites à l'aide de leurs descripteurs) à partir d'un ensemble d'objets. On peut trouver ailleurs des techniques permettant cela [Fisher et Langley, 1986]. Nous présentons seulement l'intérêt de TROEPS dans ce contexte.

TROEPS offre plusieurs avantages pour la catégorisation. Les objets étant définis indépendamment de leurs classes, il est possible de disposer des objets puis des classes. De plus, les points de vue étant indépendants, il est possible de construire plusieurs taxonomies avec des méthodes différentes ou des paramètres différents de la même méthode et de les comparer. À cette fin, le système T-TREE a été défini comme une extension de TROEPS [Euzenat, 1993a]. Celui-ci implémente un algorithme simple (de type « plus proches voisins ») et construit des taxonomies à partir des valeurs numériques des objets.

Pour aller au delà des valeurs numériques, il est nécessaire de définir une métrique sur les objets qui permette de les comparer entre eux. C'est l'objet des travaux de Petko Valtchev [Valtchev et Euzenat, 1996] qui tirent parti du système de types de TROEPS. Il est en effet possible de définir une distance associée aux types de valeurs. Cette distance est prolongée en une dissimilarité sur les objets fondée uniquement sur la structure des taxonomies existantes. Ainsi, deux objets appartenant à deux classes ayant une super-classe commune seront plus proches entre elles que d'une classe plus éloignée dans la taxonomie. Ces distances ne sont utilisées qu'une seule fois dans le processus de construction de taxonomies : une fois que les objets sont organisés en une (ou plusieurs) taxonomies, il est possible de catégoriser les objets qui y font référence en utilisant la métrique définie à partir de cette dernière taxonomie.

À des fins de comparaisons de diverses taxonomies, nous avons aussi développé une méthode d'inférence de passerelles qui permet d'obtenir toutes les passerelles possibles entre un ensemble de taxonomies (source) et une autre taxonomie (cible) [Euzenat, 1993a]. Cet algorithme, utilisé à l'origine sur les ensembles d'objets attachés aux classes, peut maintenant utiliser les descriptions des classes elles-mêmes.

Il reste cependant beaucoup de travaux à faire sur les méthodes de catégorisation pour qu'elles puissent être intégrées dans les systèmes de représentation de connaissance par objets.

## 10.6 Conclusions

Les apports de la représentation de connaissance par objets sont à rechercher dans les deux termes utilisés :

- les principaux avantages des objets sont bien entendu exploités : concentration des informations concernant un individu au sein de l'objet, exploitation des rapports générique-spécifique ;
- ceux de la représentation de connaissance aussi : une sémantique (au moins dans l'intention) contraignant la cohérence de la représentation.

Mais la place des représentations de connaissance par objets est inconfortable entre les langages de programmation par objets d'une part et les bases de données à objets d'autre part. La sémantique des représentations de connaissance par objets est différente de celles des langages de programmation par objets, même s'il peut y avoir un noyau commun (voir *chapitre 12*) : les exigences des entités de programmation et de représentation n'ont pas de raison d'être les mêmes (les uns cherchent avant tout à créer un programme, les autres cherchent avant tout à représenter un « domaine » extérieur). Par rapport aux bases de données à objets, deux distinctions peuvent être faites :

- l'aspect système des bases de données à objets, c'est-à-dire le stockage des objets en mémoire secondaire (sur disque par exemple) et les contraintes que cela impose sur les modèles de données, n'est pas traité par les représentations de connaissance par objets ;
- même si le domaine des bases de données à objets s'étend tous les jours et menace d'englober à la fois les langages de programmation par objets et les représentations de connaissance par objets (voir *chapitre 5*), les modèles de représentation de connaissance restent plus riches.

Mais il n'y a pas, *a priori*, de raison de penser que ces deux disciplines ne soient pas confondues un jour.

Au sein de la représentation de connaissance aussi, la situation des représentations de connaissance par objets est particulière. Elle apporte cependant une réponse à des besoins auxquels ne répondent pas les autres systèmes modernes de représentation de connaissance (graphes conceptuels et logiques de descriptions). En effet, la focalisation sur l'aspect formel de ces systèmes n'a pas encore permit d'atteindre une facilité d'utilisation nécessaire aux concepteurs de bases de connaissances. Par ailleurs, l'accent mis sur la définition des concepts et le calcul de subsomption comme unique mécanisme d'inférence hypothèque l'utilisabilité de ces systèmes : il existe trop de concepts dont il n'est pas possible de donner la définition (ou dont il n'est pas possible de donner la définition dans un langage restreint de manière à permettre la polynomialité du calcul de subsomption) [Doyle et Patil, 1991].

Bien que TROEPS ait été présenté plus en détail ici, il existe de nombreux systèmes de représentation de connaissance par objets (voir tableau 10.1). En général, la syntaxe utilisée pour exprimer ou pour manipuler la connaissance n'est pas la même. Cependant, ces systèmes partagent entre eux les mêmes concepts fondamentaux. Il est donc tentant de chercher à standardiser l'expression de ces représentations, comme il est tentant de le faire pour les objets de programmation. Ceci a été tenté diversement au travers des langages KIF ("*Knowledge Interchange Format*") [Genesereth et Fikes, 1992] et "*Generic Frame Protocol*" [Karp *et al.*, 1995]. Cependant, l'absence d'une compréhension profonde des différences entre langages fondée sur la sémantique formelle de ceux-ci (plutôt que sur des malentendus) rend illusoire l'obtention d'un résultat de qualité (outre qu'il risque de geler les recherches sur les modèles de représentation de connaissance [Ginsberg, 1991]).

Enfin, l'intérêt des représentations de connaissance par objets dans le cadre d'applications très diverses est largement avéré (voir par exemple [Euzenat et Rechenmann, 1995] pour SHIRKA). Ce succès devrait permettre la pérennité de ce type de systèmes de représentation de connaissance.

En dernier lieu, l'attention portée à la sémantique de la représentation est le trait à faire progresser au sein des recherches sur les représentations de connaissance par objets. C'est pourquoi un chapitre entier lui est consacré (voir *chapitre 12*).

# 11

# Une introduction aux logiques de descriptions

Les logiques de descriptions forment une famille de langages de représentation de connaissances. Elles permettent de représenter les connaissances relatives à un domaine de référence à l'aide de « descriptions » qui peuvent être des concepts, des rôles et des individus. Les concepts modélisent des classes d'individus et les rôles des relations entre classes. Une sémantique est associée aux descriptions par l'intermédiaire d'une fonction d'interprétation. La relation de subsomption permet d'organiser les concepts et les rôles en hiérarchies ; la classification et l'instanciation sont alors les opérations qui sont à la base du raisonnement sur les descriptions, ou raisonnement terminologique. La classification permet de déterminer la position d'un concept et d'un rôle dans leurs hiérarchies respectives, tandis que l'instanciation permet de retrouver les concepts dont un individu est susceptible d'être une instance.

Ce chapitre présente les éléments de base de la théorie des logiques de descriptions (LD). Les deux premiers paragraphes du chapitre introduisent la syntaxe et la sémantique d'un concept, la relation de subsomption, les notions de bases de connaissances et de raisonnement terminologiques. La détection des relations de subsomption et le raisonnement terminologique s'appréhendent principalement de deux façons. La première façon s'appuie sur des algorithmes de normalisation–comparaison, appelés ici algorithmes NC. La seconde façon s'appuie sur la méthode des tableaux sémantiques et son application aux logiques de descriptions. Les algorithmes NC sont discutés dans le deuxième paragraphe et la méthode des tableaux sémantiques l'est dans le troisième paragraphe. Une revue de la famille des logiques de descriptions, une analyse de l'apport de ces logiques dans le cadre général de la représentation des connaissances et une analyse des rapports que ces logiques entretiennent avec les représentations par objets font l'objet du quatrième paragraphe, qui termine le chapitre.

## 11.1 Introduction

Un système à base de connaissances est un programme capable de raisonner sur un domaine d'application pour résoudre un certain problème, en s'aidant de connaissances relatives au domaine étudié. Les connaissances du domaine sont représentées par des entités qui ont une description syntaxique à laquelle est associée une sémantique. Il n'existe pas de méthode universelle pour concevoir des systèmes à base de connaissances, mais un courant de recherche très actif s'est développé autour des idées dont le système KL-ONE [Brachman et Schmolze, 1985] est à l'origine. Ce courant de recherche, qui s'est nourri d'études effectuées sur la logique des prédicats, les réseaux sémantiques et les langages de *frames*, a donné naissance à une famille de langages de représentation appelés *logiques de descriptions*, ou encore *logiques terminologiques* [1].

Dans le formalisme des logiques de descriptions, un *concept* permet de représenter un ensemble d'*individus*, tandis qu'un *rôle* représente une relation binaire entre individus. Un concept correspond à une entité générique d'un domaine d'application et un individu à une entité particulière, *instance* d'un concept. Concepts, rôles et individus obéissent aux principes suivants :

- Un concept et un rôle possèdent une *description structurée*, élaborée à partir d'un certain nombre de *constructeurs*. Une sémantique est associée à chaque description de concept et de rôle par l'intermédiaire d'une *interprétation*. Les manipulations opérées sur les concepts et les rôles sont réalisées en accord avec cette sémantique.

- Les connaissances sont prises en compte selon plusieurs niveaux : la représentation et la manipulation des concepts et des rôles relèvent du niveau *terminologique* ; la description et la manipulation des individus relèvent du niveau *factuel* ou niveau des *assertions*. Le niveau terminologique est aussi qualifié de *TBox* et le niveau factuel de *ABox*.

- La relation de *subsomption* permet d'organiser concepts et rôles par niveau de généralité : intuitivement, un concept `C` *subsume* un concept `D` si `C` est plus général que `D` au sens où l'ensemble d'individus représenté par `C` contient l'ensemble d'individus représenté par `D`. Une base de connaissances se compose alors d'une *hiérarchie* de concepts et d'une (éventuelle) hiérarchie de rôles.

- Les opérations qui sont à la base du *raisonnement terminologique* sont la *classification* et l'*instanciation*. La classification s'applique aux concepts, le cas échéant aux rôles, et permet de déterminer la position d'un concept et d'un rôle dans leurs hiérarchies respectives ; la construction et l'évolution de ces hiérarchies est ainsi assistée par le processus de classification. L'instanciation

1. En anglais, ces logiques ont été désignées entre autres par les expressions *description logics*, *terminological logics* et *concept languages*. La première expression semble être devenue leur nom définitif, qui est donc adopté ici.

permet de retrouver les concepts dont un individu est susceptible d'être une instance [2].

La recherche sur les logiques de descriptions est très active aux États-Unis, en Allemagne et en Italie, et connaît un intérêt grandissant en France. L'ouvrage de référence est toujours [Nebel, 1990a], tandis que [Woods et Schmolze, 1992], [Donini *et al.*, 1996] et [Donini *et al.*, 1997] sont des synthèses très complètes. Ce chapitre est aussi une synthèse [3], qui a pour but de mettre en valeur les idées générales sur lesquelles reposent les logiques de descriptions et de montrer la richesse et la rigueur du formalisme associé, ainsi que les possibilités de représentation qui sont offertes.

## 11.2 Les bases des logiques de descriptions

### 11.2.1 Un exemple introductif

Les entités de base qui sont définies et manipulées dans une logique de descriptions sont les *concepts* et les *rôles*. Un concept dénote un ensemble d'individus – l'*extension* du concept – et un rôle dénote une relation binaire entre individus. Un concept possède une *description structurée* qui se construit à l'aide d'un ensemble de *constructeurs* introduisant les rôles associés au concept et les *restrictions* attachées à ces rôles. Les restrictions portent généralement sur le *co-domaine* du rôle, qui est le concept avec lequel le rôle établit une relation, et la *cardinalité* du rôle, qui fixe le nombre minimal et maximal de *valeurs élémentaires* que peut prendre le rôle. Les valeurs élémentaires sont des instances de concepts ou bien des valeurs qui relèvent des types de bases comme *entier*, *réel*, et *chaîne de caractères*.

Les concepts peuvent être *primitifs* ou *définis*. Les concepts primitifs sont comparables à des atomes et servent de base à la construction des concepts définis, c'est-à-dire qui possèdent une *définition*. À l'image d'un concept, un rôle peut être primitif ou défini et peut posséder une description structurée, où figurent les propriétés associées au rôle.

À la figure 11.1, nous donnons un exemple de concepts et de rôles primitifs et définis. Par convention, un nom de concept commence par une majuscule ; un nom de rôle est en minuscules et un nom d'individu est en majuscules. Les concepts `Personne` et `Ensemble` sont des concepts primitifs : ils sont introduits par le symbole $\dot{\leq}$ et sont *subsumés* par `Top`, qui dénote le concept le plus général, appelé aussi la *racine* de la hiérarchie des concepts. Le constructeur `and` indique qu'un concept est construit à partir d'une conjonction de concepts — qui sont les ascendants directs du nouveau concept — et le constructeur `all` précise le co-domaine d'un rôle. Le constructeur `not` exprime la négation et ne s'applique qu'à des concepts primitifs. Les constructeurs `atleast` et `atmost` précisent la cardinalité du rôle auquel ils sont associés, et ils indiquent respectivement le nombre minimal et maximal de valeurs élémentaires du rôle.

2. Dans les langages à objets, l'instanciation a un sens différent : elle désigne l'opération qui permet d'engendrer des instances à partir d'une classe (voir par exemple les *chapitres 1, 2 et 3*).

3. Le rapport de recherche [Napoli, 1997] est une version étendue et plus complète de ce chapitre.

```
Personne ≤̇ Top
Ensemble ≤̇ Top
Homme ≤̇ Personne
Femme ≤̇ (and Personne (not Homme))
membre ≤̇ toprole
chef ≤̇ membre

Équipe ≐ (and Ensemble
              (all membre Personne)
              (atleast 2 membre))

Petite-équipe ≐ (and Équipe
                     (atmost 5 membre))

Équipe-moderne ≐ (and Équipe
                      (atmost 4 membre)
                      (atleast 1 chef)
                      (all chef femme))
```

FIG. 11.1: *Concepts primitifs, rôles primitifs et concepts définis, d'après [Nebel, 1990a].*

Les concepts `Homme` et `Femme` sont primitifs et *incompatibles* : l'ensemble des femmes est dans le complémentaire de celui des hommes. Les trois concepts `Équipe`, `Petite-équipe` et `Équipe-moderne` sont des concepts définis introduits par le symbole $\doteq$. Une équipe est définie comme un ensemble de personnes qui compte au moins deux membres. Une petite équipe est définie comme une équipe qui compte au plus cinq membres. Une équipe moderne est définie comme une équipe qui compte au plus quatre membres, au moins un chef et dont tous les chefs sont des femmes. Du côté des rôles, `membre` et `chef` sont primitifs : `chef` est subsumé par `membre`, et `membre` par `toprole`, qui dénote le rôle le plus général.

Deux remarques principales peuvent être faites sur ce premier exemple. La première est que la description d'un concept — `Top` excepté — *dérive* toujours d'un ou de plusieurs autres concepts, qui sont ses *subsumants* dans la hiérarchie des concepts (comme c'est le cas pour les sous-classes qui *spécialisent* les classes dans un langages à objets, voir *chapitre 1*). La seconde remarque concerne le statut des concepts : certains concepts, comme `Équipe`, `Petite-équipe` et `Équipe-moderne` possèdent une *définition* ; d'autres, comme `Personne`, `Ensemble`, `Homme` ou encore `Femme` sont primitifs. Les caractéristiques associées à un concept primitif sont *nécessaires* : un individu `x` qui est une instance d'un concept primitif `P` possède les caractéristiques de `P` (comme c'est le cas pour les instances d'une classe dans un langages à objets, voir *chapitre 10*). Les caractéristiques associées à un concept défini `D` sont *nécessaires et suffisantes* : un individu `x` qui est une instance d'un concept

| | syntaxe lispienne | syntaxe allemande |
|---|---|---|
| $\mathtt{C,D} \longrightarrow$ | `A \|` | |
| | `Top \|` | $\top$ |
| | `Bottom \|` | $\bot$ |
| | `(and C D) \|` | $\mathtt{C} \sqcap \mathtt{D}$ |
| | `(not A) \|` | $\neg\mathtt{A}$ |
| | `(all r C) \|` | $\forall\mathtt{r.C}$ |
| | `(some r)` | $\exists\mathtt{r}$ |

FIG. 11.2: *La grammaire du langage de description de concepts $\mathcal{AL}$, avec les syntaxes lispienne et allemande.* `C` *et* `D` *sont des noms de concepts,* `A` *un nom de concept primitif et* `r` *un nom de rôle primitif.*

défini `D` possède les caractéristiques de `D`, et inversement, le fait qu'un individu `y` possède l'ensemble des caractéristiques associées à `D` suffit pour inférer que `y` est une instance de `D`. Cette distinction est à la base du processus de classification : les concepts sont définis de façon déclarative et la mise en place des concepts définis dans la hiérarchie des concepts est effectuée sous le contrôle du processus de classification.

Le caractère nécessaire et suffisant des propriétés associées à une classes et le processus de classification sont également discutés dans le *chapitre 10* pour les systèmes de représentation de connaissances par objets — ou systèmes de RCO — et dans le *chapitre 12* pour les systèmes classificatoires.

## 11.2.2 La description des concepts et des rôles : syntaxe

Il existe plusieurs langages de description de concepts et de rôles. Dans ce qui suit, nous introduisons d'abord un langage minimal appelé $\mathcal{AL}$, qui est enrichi progressivement de nouveaux constructeurs. Le langage $\mathcal{AL}$ s'appuie sur les langages $\mathcal{FL}$ et $\mathcal{FL}$-, qui sont les premières logiques de descriptions [4] à l'aide desquelles ont été établis des résultats théoriques sur la complexité de la subsomption [Brachman et Levesque, 1984] [Levesque et Brachman, 1987].

La grammaire de $\mathcal{AL}$ est donnée à la figure 11.2. Les expressions construites grâce à une telle grammaire sont aussi appelées *expressions conceptuelles*. À la syntaxe *lispienne*, où le nom des constructeurs est donné en toutes lettres et où la notation est préfixée, correspond une syntaxe *allemande*, qui est utilisée dans la plupart des articles théoriques traitant des logiques de descriptions. Dans la suite, les deux syntaxes seront utilisées indifféremment.

Le concept `Top` ($\top$) dénote le concept le plus général et le concept `Bottom` ($\bot$) le concept le plus spécifique. Intuitivement, l'extension de `Top` inclut tous les individus possibles tandis que celle de `Bottom` est vide. Le constructeur `and` ($\sqcap$) permet

4. Pour la petite histoire, dans [Brachman et Levesque, 1984], les logiques de descriptions sont qualifiées de *frame-based description languages*, d'où les initiales $\mathcal{FL}$.

de définir une conjonction d'expressions conceptuelles. Le constructeur `not` ($\neg$) correspond à la négation et ne porte que sur les concepts primitifs. La *quantification universelle* `all` ($\forall$`r.C`) précise le co-domaine du rôle `r` ; la *quantification existentielle* non typée `some` ($\exists$ `r`) introduit le rôle `r` et affirme l'existence d'(au moins) un couple d'individus en relation par l'intermédiaire de `r`.

Le langage $\mathcal{AL} = \{\top, \bot, \neg\mathtt{A}, \mathtt{C}\sqcap\mathtt{D}, \forall\mathtt{r.C}, \exists\,\mathtt{r}\}$ peut être enrichi des constructeurs suivants :

- La négation de concepts primitifs ou définis, qui est notée (`not C`) ou $\neg\mathtt{C}$. L'extension correspondante de $\mathcal{AL}$ est $\mathcal{ALC} = \mathcal{AL} \cup \{\neg\mathtt{C}\}$.
- La disjonction de concepts, qui est notée (`or C D`) ou $\mathtt{C} \sqcup \mathtt{D}$. L'extension correspondante de $\mathcal{AL}$ est $\mathcal{ALU} = \mathcal{AL} \cup \{\mathtt{C} \sqcup \mathtt{D}\}$.

  Notons les équivalences $\bot \equiv \mathtt{C} \sqcup \neg\mathtt{C}$ et $\mathtt{C} \sqcup \mathtt{D} \equiv \neg(\neg\mathtt{C} \sqcap \neg\mathtt{D})$.
- La quantification existentielle typée, qui est notée (`c-some r C`) ou $\exists\mathtt{r.C}$. L'extension correspondante de $\mathcal{AL}$ est $\mathcal{ALE} = \mathcal{AL} \cup \{\exists\mathtt{r.C}\}$.

  La quantification existentielle typée $\exists\mathtt{r.C}$ introduit un rôle `r` de co-domaine `C` et impose l'existence d'(au moins) un couple d'individus `(x,y)` en relation par l'intermédiaire du rôle `r`, où `C` est le type de `y`.

  Notons les équivalences $\exists\,\mathtt{r} \equiv \exists\mathtt{r}.\top$ et $\exists\mathtt{r.C} \equiv \neg(\forall\mathtt{r}.\neg\mathtt{C})$.
- La cardinalité sur les rôles, notée (`atleast n r`) ou $\geq\mathtt{n\,r}$, et (`atmost n r`) ou $\leq\mathtt{n\,r}$. L'extension correspondante de $\mathcal{AL}$ est $\mathcal{ALN} = \mathcal{AL} \cup \{\geq\mathtt{n\,r}, \leq\mathtt{n\,r}\}$.

  Les constructeurs $\geq\mathtt{n\,r}$ et $\leq\mathtt{n\,r}$ fixent la cardinalité — nombre de valeurs élémentaires — minimale et maximale du rôle auquel ils sont associés. En particulier, la construction ($\exists\,\mathtt{r}$) est équivalente à la construction ($\geq\mathtt{1\,r}$).
- La conjonction de rôles, qui est notée (`and` $\mathtt{r_1}$ $\mathtt{r_2}$) ou $\mathtt{r_1} \sqcap \mathtt{r_2}$, les rôles $\mathtt{r_1}$ et $\mathtt{r_2}$ étant primitifs. L'extension correspondante de $\mathcal{AL}$ est $\mathcal{ALR} = \mathcal{AL} \cup \{\mathtt{r_1} \sqcap \mathtt{r_2}\}$.

  Si $\mathtt{r} = \mathtt{r_1} \sqcap \mathtt{r_2}$, alors `r` est un sous-rôle de $\mathtt{r_1}$ et de $\mathtt{r_2}$. Par extension, il est possible de considérer une hiérarchie de rôles comme un ensemble de conjonctions de rôles et de se passer du rôle `toprole` — ce qui est d'ailleurs le cas pour $\mathcal{ALCNR}$— comme expliqué dans [Nebel, 1990a, page 54]. Ainsi, le rôle `chef` peut être défini par `membre` $\sqcap$ `chef-prim`, où `chef-prim` est un rôle primitif au même titre que `membre`.

Comme le montrent les équivalences $\mathtt{C} \sqcup \mathtt{D} \equiv \neg(\neg\mathtt{C} \sqcap \neg\mathtt{D})$ et $\exists\mathtt{r.C} \equiv \neg(\forall\mathtt{r}.\neg\mathtt{C})$, la disjonction et la quantification existentielle typée sont disponibles dans un langage $\mathcal{L}$ de la famille $\mathcal{AL}$ dès lors que la négation de concepts (définis) l'est, et réciproquement. Par convention, un langage $\mathcal{L}$ de la famille $\mathcal{AL}$ est supposé contenir tous les constructeurs qui peuvent être obtenus à partir de ses constructeurs de base. Ainsi, le langage $\mathcal{ALCNR} = \{\top, \bot, \neg\mathtt{C}, \mathtt{C}\sqcap\mathtt{D}, \mathtt{C}\sqcup\mathtt{D}, \forall\mathtt{r.C}, \exists\mathtt{r.C}, \geq\mathtt{n\,r}, \leq\mathtt{n\,r}, \mathtt{r_1} \sqcap \mathtt{r_2}\}$ contient l'ensemble des constructeurs qui viennent d'être présentés (il est aussi d'usage d'employer la lettre $\mathcal{C}$ plutôt que les lettres $\mathcal{UE}$ dans le nom du langage). D'autres équivalences de langages sont discutées par exemple dans [Donini *et al.*, 1997] (simulation de la quantification existentielle typée avec la conjonction de rôles) et dans [Baader, 1996] (simulation de la négation de concepts primitifs avec les restrictions de cardinalité sur les rôles).

Le langage $\mathcal{FL}$, quant à lui, se définit par l'ensemble de constructeurs $\{\mathtt{C} \sqcap \mathtt{D}, \forall \mathtt{r.C}, \exists\,\mathtt{r}, \mathtt{r}\,|\mathtt{C}\}$, où $\mathtt{r}\,|\mathtt{C}$, qui se note aussi $(\mathtt{restrict\ r\ C})$, introduit une contrainte sur le co-domaine du rôle $\mathtt{r}$. Le langage $\mathcal{FL}$- est une simplification de $\mathcal{FL}$ dans laquelle le constructeur `restrict` n'est pas utilisé.

La construction $(\mathtt{restrict\ r\ C})$ peut être rapprochée de la quantification existentielle typée $(\mathtt{c\text{-}some\ r\ C})$, en remarquant toutefois que : d'une part, $(\mathtt{c\text{-}some\ r\ C})$ est un constructeur s'appliquant aux concepts tandis que $(\mathtt{restrict\ r\ C})$ s'applique aux rôles, et d'autre part, $(\mathtt{restrict\ r\ C})$ ne fait que restreindre le co-domaine du rôle `r` au concept `C`, sans aucune contrainte d'existence comme celle qui est associée à la quantification existentielle $(\mathtt{c\text{-}some\ r\ C})$.

Tous les constructeurs envisageables pour une logique de descriptions n'ont pas été présentés ici et il est toujours possible d'enrichir le langage $\mathcal{ALCNR}$. Des listes plus complètes de constructeurs sont données dans [Woods et Schmolze, 1992], [Nebel, 1990a], [Schaerf, 1994] et enfin dans [Patel-Schneider et Swartout, 1993] qui étend et enrichit [Baader *et al.*, 1990].

### 11.2.3 La description des concepts et des rôles : sémantique

À l'instar de la logique classique, une sémantique est associée aux descriptions de concepts et de rôles : les concepts sont interprétés comme des sous-ensembles d'un domaine d'interprétation $\Delta_\mathcal{I}$ et les rôles comme des sous-ensembles du produit $\Delta_\mathcal{I} \times \Delta_\mathcal{I}$. Pour un concept $\mathtt{C}$, $\mathtt{C}^\mathcal{I}$ correspond au sous-ensemble des éléments du domaine $\Delta_\mathcal{I}$ qui appartiennent à l'extension de $\mathtt{C}$, et pour un rôle $\mathtt{r}$, $\mathtt{r}^\mathcal{I}$ correspond au sous-ensemble des couples d'éléments du produit $\Delta_\mathcal{I} \times \Delta_\mathcal{I}$ qui appartiennent à l'extension de $\mathtt{r}$. La définition suivante est donnée dans le cadre du langage $\mathcal{ALCNR}$ et introduit la notion d'*interprétation* dans les logiques de descriptions.

**Définition 1** ***(Interprétation)***

*Une* interprétation $\mathcal{I} = (\Delta_\mathcal{I}, .^\mathcal{I})$ *est la donnée d'un ensemble* $\Delta_\mathcal{I}$ *appelé* domaine de l'interprétation *et d'une fonction d'interprétation* $.^\mathcal{I}$ *qui fait correspondre à un concept un sous-ensemble de* $\Delta_\mathcal{I}$ *et à un rôle un sous-ensemble de* $\Delta_\mathcal{I} \times \Delta_\mathcal{I}$*, de telle sorte que les équations suivantes soient satisfaites :*

$$
\begin{aligned}
\top^\mathcal{I} &= \Delta_\mathcal{I} \\
\bot^\mathcal{I} &= \emptyset \\
(\mathtt{C} \sqcap \mathtt{D})^\mathcal{I} &= \mathtt{C}^\mathcal{I} \cap \mathtt{D}^\mathcal{I} \\
(\mathtt{C} \sqcup \mathtt{D})^\mathcal{I} &= \mathtt{C}^\mathcal{I} \cup \mathtt{D}^\mathcal{I} \\
(\neg\mathtt{C})^\mathcal{I} &= \Delta_\mathcal{I} - \mathtt{C}^\mathcal{I} \\
(\forall\mathtt{r.C})^\mathcal{I} &= \{\mathtt{x} \in \Delta_\mathcal{I} / \forall \mathtt{y} : (\mathtt{x}, \mathtt{y}) \in \mathtt{r}^\mathcal{I} \rightarrow \mathtt{y} \in \mathtt{C}^\mathcal{I}\} \\
(\exists\mathtt{r.C})^\mathcal{I} &= \{\mathtt{x} \in \Delta_\mathcal{I} / \exists \mathtt{y} : (\mathtt{x}, \mathtt{y}) \in \mathtt{r}^\mathcal{I} \wedge \mathtt{y} \in \mathtt{C}^\mathcal{I}\}
\end{aligned}
$$

$$
\begin{aligned}
(\geq \mathtt{n\,r})^{\mathcal{I}} &= \{\mathtt{x} \in \Delta_{\mathcal{I}} / |\{\mathtt{y} \in \Delta_{\mathcal{I}} / (\mathtt{x},\mathtt{y}) \in \mathtt{r}^{\mathcal{I}}\}| \geq n\} \\
(\leq \mathtt{n\,r})^{\mathcal{I}} &= \{\mathtt{x} \in \Delta_{\mathcal{I}} / |\{\mathtt{y} \in \Delta_{\mathcal{I}} / (\mathtt{x},\mathtt{y}) \in \mathtt{r}^{\mathcal{I}}\}| \leq n\} \\
(\mathtt{r_1} \sqcap \cdots \sqcap \mathtt{r_n})^{\mathcal{I}} &= \mathtt{r_1}^{\mathcal{I}} \cap \cdots \cap \mathtt{r_n}^{\mathcal{I}}
\end{aligned}
$$

Intuitivement, les équations précédentes se comprennent comme suit. L'interprétation de $\top$ est le domaine $\Delta_{\mathcal{I}}$ tout entier tandis que celle de $\perp$ se réduit à l'ensemble vide. L'interprétation d'une conjonction (respectivement d'une disjonction) de concepts se ramène à l'intersection (respectivement la réunion) des interprétations des concepts. L'interprétation de la négation d'un concept $\mathtt{C}$ se ramène au complémentaire de l'interprétation de $\mathtt{C}$. L'interprétation de $(\forall \mathtt{r.C})$ précise le type du co-domaine du rôle $\mathtt{r}$, tandis que celle de $(\exists \mathtt{r.C})$ affirme l'existence d'un couple d'éléments $(\mathtt{x},\mathtt{y})$ en relation par l'intermédiaire du rôle $\mathtt{r}$, où $\mathtt{C}$ est le type de $\mathtt{y}$. Les interprétations de $(\geq \mathtt{n\,r})$ et $(\leq \mathtt{n\,r})$ précisent le nombre d'éléments de l'ensemble de valeurs élémentaires associées au rôle $\mathtt{r}$. Enfin, l'interprétation d'une conjonction de rôles se ramène à l'intersection des interprétations des rôles.

L'interprétation de la construction $(\exists\, \mathtt{r})$ est un cas particulier de celle de $(\exists \mathtt{r.C})$, où $\mathtt{C} \equiv \top$ : $(\exists\, \mathtt{r})^{\mathcal{I}} = \{\mathtt{x} \in \Delta_{\mathcal{I}} / \exists \mathtt{y} : (\mathtt{x},\mathtt{y}) \in \mathtt{r}^{\mathcal{I}}\}$ ; il existe au moins un couple $(\mathtt{x},\mathtt{y})$ tel que $\mathtt{x}$ et $\mathtt{y}$ soient en relation par l'intermédiaire de $\mathtt{r}$ (le type de $\mathtt{y}$ est ici quelconque).

**Définition 2** ***(Satisfiabilité d'un concept, équivalence, incompatibilité)***

- *Un concept* $\mathtt{C}$ *est* satisfiable *ou* cohérent *si et seulement s'il existe une interprétation* $\mathcal{I}$ *telle que* $\mathtt{C}^{\mathcal{I}} \neq \emptyset$ *;* $\mathtt{C}$ *est* non satisfiable *ou* incohérent *sinon.*
- *Deux concepts* $\mathtt{C}$ *et* $\mathtt{D}$ *sont dits* équivalents*, ce qui se note* $\mathtt{C} \equiv \mathtt{D}$*, si et seulement si* $\mathtt{C}^{\mathcal{I}} = \mathtt{D}^{\mathcal{I}}$ *pour toute interprétation* $\mathcal{I}$*.*
- *Deux concepts* $\mathtt{C}$ *et* $\mathtt{D}$ *sont* incompatibles *ou* disjoints *si et seulement si* $\mathtt{C}^{\mathcal{I}} \cap \mathtt{D}^{\mathcal{I}} = \emptyset$ *pour toute interprétation* $\mathcal{I}$*.*

Considérons par exemple les expressions conceptuelles suivantes, où les concepts et les rôles employés sont supposés primitifs :

(1) `(and Homme (some enfant Musicien))`
(2) `(and Femme (all enfant Homme))`
(3) `(and Femme (all enfant (and Musicien Homme)))`
(4) `(and (atmost 0 r) (atleast 1 r))`
(5) `(some r (and A (not A)))`

Les trois premières expressions conceptuelles sont satisfiables : (1) représente l'ensemble des individus qui sont des hommes et qui ont un enfant musicien ; (2) représente l'ensemble des individus qui sont des femmes et dont tous les enfants sont des garçons ; (3) représente l'ensemble des individus qui sont des femmes et dont tous les enfants sont des garçons et des musiciens. En revanche, les quatrième et cinquième expressions conceptuelles sont non satisfiables : quel que soit le rôle $\mathtt{r}$, une instance de l'expression conceptuelle (4) devrait être munie simultanément d'au moins une valeur et d'au plus zéro valeur pour r ; l'expression conceptuelle (5) suppose quant à elle l'existence d'un élément dans l'ensemble vide.

Une des particularités de $\mathcal{FL}$-, qui est un langage pour lequel aucune forme de négation n'est disponible, est que tout concept défini en $\mathcal{FL}$- est satisfiable ; un résultat plus fort est démontré dans [Schmidt-Schauß et Smolka, 1991] : tout concept de $\mathcal{FL}$ est satisfiable.

Les notions de satisfiabilité et de cohérence sont également discutées dans le *chapitre 12*, dans le cadre des systèmes classificatoires.

### 11.2.4 La relation de subsomption

Intuitivement, un concept C *subsume* un concept D si et seulement si l'extension de C contient nécessairement — quelle que soit l'interprétation choisie — l'extension de D. Plus formellement, nous avons :

**Définition 3** ***(Subsomption)***

*Un concept* D *est* subsumé *par un concept* C *(respectivement* C subsume D*), ce qui se note* $\mathtt{D} \sqsubseteq \mathtt{C}$ *(respectivement* $\mathtt{C} \sqsupseteq \mathtt{D}$*) si et seulement si* $\mathtt{D}^{\mathcal{I}} \subseteq \mathtt{C}^{\mathcal{I}}$ *pour toute interprétation* $\mathcal{I}$*. Le concept* C *est appelé le* subsumant *et* D *le* subsumé.

Par exemple, un concept est subsumé par tous les concepts qui ont été déclarés comme ses subsumants directs : Personne est subsumé par Top, Homme par Personne, etc. (voir figure 11.1). De plus, le concept Petite-équipe subsume le concept Équipe-moderne car toute instance du second concept vérifie nécessairement toutes les caractéristiques du premier concept, essentiellement le fait qu'une petite équipe compte au plus 5 membres. Par suite, le concept Équipe-moderne est un subsumé du concept Petite-équipe.

La relation de subsomption est *réflexive* (un concept est subsumé par lui-même) et *transitive* (si E est subsumé par D et D est subsumé par C alors E est subsumé par C). Elle est aussi *antisymétrique* (si D est subsumé par C et C est subsumé par D, alors C = D) si deux concepts ayant la même extension sont identifiés, ce que nous supposerons. La relation de subsomption est donc une relation d'ordre partiel, qui organise les concepts en une hiérarchie, où tout concept se compose d'une description propre définie par des propriétés *locales* et d'une description *partagée* avec ses subsumants (comme c'est le cas entre une sous-classe et ses super-classes, voir le *chapitre 1*). La hiérarchie obtenue possède un élément maximal, Top, qui est le concept qui subsume tous les autres concepts, et un élément minimal, Bottom, qui est subsumé par tous les autres concepts.

Nous pouvons reformuler la définition 2 en termes de subsomption :

- Un concept C est non satisfiable si et seulement si $\mathtt{C} \sqsubseteq \bot$.
- Deux concepts C et D sont équivalents si et seulement si $\mathtt{D} \sqsubseteq \mathtt{C}$ et $\mathtt{C} \sqsubseteq \mathtt{D}$. En particulier, $\mathtt{D} \equiv \mathtt{C} \sqcap \mathtt{D}$ si et seulement si $\mathtt{D} \sqsubseteq \mathtt{C}$.
- Deux concepts C et D sont incompatibles si et seulement si $\mathtt{D} \sqcap \mathtt{C} \sqsubseteq \bot$. Cette vision alternative de l'incompatibilité permet d'exprimer que les concepts Homme et Femme sont disjoints de la façon suivante : $\mathtt{Homme} \sqcap \mathtt{Femme} \sqsubseteq \bot$.

### 11.2.5 Le niveau terminologique et les méthodes de tests de subsomption

Classiquement, deux types de déclarations se rencontrent au *niveau terminologique* : les introductions de concepts primitifs (symbole $\dot{\leq}$), et les définitions de concepts (symbole $\doteq$). Chacune de ces déclarations s'interprète comme une *équation terminologique*, qui se compose d'un membre gauche où apparaît le nom du nouveau concept et d'un membre droit où apparaît l'expression subsumant le nouveau concept en cas d'introduction de concept primitif, ou la définition du nouveau concept défini. Du côté des rôles, seules les équations terminologiques introduisant des rôles primitifs sont considérées ici.

En général, ces déclarations s'appuient sur les deux principes suivants : d'une part, un nom de concept n'est utilisé qu'une seule fois en partie gauche d'une équation terminologique, d'autre part, il n'existe pas de *circuit terminologique*. Un circuit terminologique apparaît lorsqu'un concept fait référence directement ou indirectement à lui-même lors de son introduction (le concept apparaît donc simultanément en partie gauche et droite de l'équation terminologique, éventuellement par transitivité). Dans la suite, nous supposons qu'il n'existe pas de circuit terminologique dans les définitions de concepts et de rôles[5]. Si l'absence de circuit terminologique est vérifiée, alors il est possible de substituer tout nom de concept défini par sa définition dans n'importe quelle expression conceptuelle. En particulier, cette opération « de développement des définitions » est à la base du processus de normalisation des expressions conceptuelles qui est présenté ci-après.

La subsomption est la relation fondamentale existant entre les descriptions. Pour tester les relations de subsomption entre les descriptions, deux voies ont été principalement explorées jusqu'à présent : les algorithmes de type normalisation–comparaison, abrégés en *algorithmes NC*, et une méthode dérivée de la méthode des tableaux sémantiques en logique classique [Gochet et Gribomont, 1991], que nous appelons plus simplement dans la suite « méthode des tableaux sémantiques ».

Un algorithme NC s'appuie sur un processus de *normalisation* pendant lequel tous les composants d'une description sont développés et factorisés : toute l'information partagée est recopiée dans le concept traité ; les concepts définis sont remplacés par leur définition, et ainsi tous les symboles d'une définition développée dénotent des concepts ou des rôles primitifs. Le processus de normalisation produit les *formes normales* des descriptions, qui sont ensuite effectivement comparées. Toutefois, seules les formes normales des concepts définis doivent être comparées, car les relations de subsomption sont systématiquement spécifiées pour les concepts primitifs. Il existe une certaine dualité entre normalisation et comparaison : plus il y a de travail fait lors de la normalisation, moins il y en aura à faire lors de la comparaison.

Dans la méthode des tableaux sémantiques, la question « est-ce que `C` subsume `D` » est transformée en la question « est-ce que $\mathtt{D} \sqcap \neg\mathtt{C}$ est non satisfiable » — ce qui suppose que le langage de description des concepts est muni de la négation

5. Un certain nombre d'études sur les circuits terminologiques existent, et nous nous renvoyons le lecteur intéressé à [Nebel, 1991], [Buchheit *et al.*, 1993] et [De Giacomo et Lenzerini, 1997].

des concepts définis — puis la méthode des tableaux sémantiques est utilisée pour répondre à la dernière question (ce qui est présenté plus en détail au § 11.3).

S'il est facile de vérifier la *correction* des algorithmes NC, (la réponse fournie par l'algorithme est en accord avec la sémantique, voir par exemple [Nebel, 1990a, page 77]), il est beaucoup plus difficile de prouver qu'un tel algorithme est *complet*, autrement dit que toutes les relations de subsomption valides sont détectées par l'algorithme. Il s'avère qu'en dehors des langages simples comme $\mathcal{FL}$-, la plupart des algorithmes NC sont incomplets, ce qui se montre en exhibant des contre-exemples (comme il en est fourni dans [Heinsohn *et al.*, 1994] par exemple). En revanche, il est plus facile de démontrer la correction et la complétude des algorithmes qui s'appuient sur la méthode des tableaux sémantiques, ainsi que d'étudier leur complexité et la décidabilité de la logique associée.

### 11.2.6 Algorithmes NC pour tester la subsomption

**La subsomption dans $\mathcal{FL}$-**

Nous présentons dans ce paragraphe l'algorithme (de test) de subsomption associé à $\mathcal{FL}$-, qui, malgré sa simplicité, illustre bien les processus de normalisation et de comparaison des formes normales (rappelons que le langage $\mathcal{FL}$- est défini par l'ensemble de constructeurs $\{\mathtt{C} \sqcap \mathtt{D}, \forall \mathtt{r.C}, \exists\, \mathtt{r}\}$).

Le but de la normalisation est de mettre les concepts définis $\mathtt{C}$ et $\mathtt{D}$ qui sont comparés sous la forme de conjonctions $\mathtt{C} = (\mathtt{and}\ \mathtt{C_1}\ \mathtt{C_2} ... \mathtt{C_n})$ et $\mathtt{D} = (\mathtt{and}\ \mathtt{D_1}\ \mathtt{D_2} ... \mathtt{D_m})$. Dans la présentation de l'algorithme, le symbole $\rightarrow$ doit se lire « se réécrit en » :

- *Normalisation* : factorisation des conjonctions.
  $(\mathtt{and}\ \mathtt{C_1}\ (\mathtt{and}\ \mathtt{C_2}\ \mathtt{C_3})\ \mathtt{C_4}) \rightarrow (\mathtt{and}\ \mathtt{C_1}\ \mathtt{C_2}\ \mathtt{C_3}\ \mathtt{C_4})$.
- *Normalisation* : factorisation des co-domaines.
  $(\mathtt{and}\ (\mathtt{all}\ \mathtt{r}\ (\mathtt{and}\ \mathtt{C_1}\ \mathtt{C_2}))\ (\mathtt{all}\ \mathtt{r}\ (\mathtt{and}\ \mathtt{C_3}\ \mathtt{C_4}))) \rightarrow$
  $(\mathtt{and}\ (\mathtt{all}\ \mathtt{r}\ (\mathtt{and}\ \mathtt{C_1}\ \mathtt{C_2}\ \mathtt{C_3}\ \mathtt{C_4})))$.
- *Comparaison* :
  après normalisation, $\mathtt{C} = (\mathtt{and}\ \mathtt{C_1}\ \mathtt{C_2} ... \mathtt{C_n})$ et $\mathtt{D} = (\mathtt{and}\ \mathtt{D_1}\ \mathtt{D_2} ... \mathtt{D_m})$.
  $\mathtt{D} \sqsubseteq \mathtt{C}$ est $\mathtt{vrai}$ si et seulement si, pour chaque $\mathtt{C_i}$, $\mathtt{i} = 1, .., \mathtt{n}$ :
  - $[\mathtt{concept\ primitif}]$ : si $\mathtt{C_i}$ est un concept primitif, alors il existe $\mathtt{D_j}$ dans l'ensemble $\{\mathtt{D_1}, \mathtt{D_2}, ..., \mathtt{D_m}\}$ tel que $\mathtt{D_j} = \mathtt{C_i}$ ou $\mathtt{D_j} \dot{\leq} \mathtt{C_i}$ (éventuellement par transitivité).
  - $[\mathtt{some}]$ : si $\mathtt{C_i} = (\mathtt{some}\ \mathtt{r})$, alors il existe $\mathtt{D_j}$ dans $\{\mathtt{D_1}, \mathtt{D_2}, ..., \mathtt{D_m}\}$ tel que $\mathtt{D_j} = \mathtt{C_i}$.
  - $[\mathtt{all}]$ : si $\mathtt{C_i} = (\mathtt{all}\ \mathtt{r}\ \mathtt{X})$, alors il existe $\mathtt{D_j}$ dans $\{\mathtt{D_1}, \mathtt{D_2}, ..., \mathtt{D_m}\}$ tel que $\mathtt{D_j} = (\mathtt{all}\ \mathtt{r}\ \mathtt{Y})$ avec $\mathtt{Y} \sqsubseteq \mathtt{X}$.

Un exemple illustrant cet algorithme est donné à la figure 11.3. Le concept $\mathtt{D}$, qui dénote « un homme ayant des enfants et dont tous les amis sont des docteurs ayant une spécialité », est subsumé par le concept $\mathtt{C}$, qui dénote « une personne dont tous les amis sont docteurs ». Les deux concepts $\mathtt{D} = (\mathtt{and}\ \mathtt{D_1}\ \mathtt{D_2}\ \mathtt{D_3})$ et $\mathtt{C}$

```
D ≐ (and Homme                                        D1
        (some enfant)                                 D2
        (all ami (and Docteur                         D3
                      (some spécialité))))

C ≐ (and Personne                                     C1
        (all ami Docteur))                            C2
```

FIG. 11.3: *Le concept* D *est subsumé par le concept* C.

= (and $C_1$ $C_2$) sont sous forme normale, et, $D_1 \sqsubseteq C_1$ car Homme est subsumé par Personne ; $D_3 \sqsubseteq C_2$ car (and Docteur X) est subsumé par Docteur, où X est une description quelconque (ici, X = (some spécialité)).

Deux remarques peuvent être faites : d'une part, la relation de subsomption (and C D) $\sqsubseteq$ C est toujours vérifiée, quelle que soit la description D, d'autre part, le nombre d'éléments de la conjonction D = (and $D_1$ $D_2$...$D_m$) n'est pas forcément plus grand que celui de la conjonction C = (and $C_1$ $C_2$...$C_n$), comme le montre la notion d'équivalence dans la définition 2 : D $\sqsubseteq$ C si et seulement si D $\equiv$ (and D C) ; dans ce cas, (and D C) $\sqsubseteq$ D, mais aussi D $\sqsubseteq$ (and D C).

Dans [Nebel, 1990a, page 54], et comme cela a déjà été discuté pour les rôles au paragraphe 11.2.2, les introductions de concepts primitifs sont transformées en définition de concepts de la façon suivante : Homme $\dot{\leq}$ Personne se transforme en Homme $\doteq$ (and Personne Homme-prim), où Homme-prim est un concept primitif. En s'appuyant sur cette transformation, si, lors de la normalisation, tous les concepts sont remplacés par leur « définition », la clause [concept primitif] de l'algorithme ci-dessus peut se réécrire comme suit : si $C_i$ est un concept primitif, alors il existe $D_j$ dans l'ensemble $\{D_1, D_2, ..., D_m\}$ tel que $D_j = C_i$.

### La subsomption dans $\mathcal{ALNR}$

Ce paragraphe complète le précédent et présente les principales règles de subsomption associées à $\mathcal{ALNR}$.

[concept ou rôle primitif] **:** si C et D dénotent des concepts ou des rôles primitifs, alors D $\sqsubseteq$ C si D = C ou D $\dot{\leq}$ E avec E $\sqsubseteq$ C (règle identique à celle de l'algorithme de $\mathcal{FL}$-).

[not] **:** si C = $\neg$C′ et D = $\neg$D′ sont des négations de concepts primitifs, alors D $\sqsubseteq$ C si C′ $\sqsubseteq$ D′. Ainsi, la négation du concept Docteur subsume la négation du concept Bachelier, Bachelier subsumant Docteur (figure 11.4, (1)). Les cas où les négations sont *croisées*, qui consisteraient à comparer C = $\neg$C′ et D, ou bien C et D = $\neg$D′, ne sont pas traités ici.

[all] **:** si $r_c \sqsubseteq r_d$ et D $\sqsubseteq$ C, alors (all $r_d$ D) $\sqsubseteq$ (all $r_c$ C).

```
(1) (not Bachelier) ⊑ (not Docteur)
(2) (all enfant Docteur) ⊑ (all fils Bachelier)
(3) (all ami (not Bachelier)) ⊑ (all ami (not Docteur))
(4) (atleast 2 fille) ⊑ (atleast 1 enfant)
(5) (atmost 1 enfant) ⊑ (atmost 2 fille)
```

FIG. 11.4: *Relations de subsomption en $\mathcal{ALNR}$.*

Par exemple, le concept « personne dont tous les fils sont bacheliers » subsume le concept « personne dont tous les enfants sont docteurs », où le rôle `fils` est subsumé par le rôle `enfant` (figure 11.4, `(2)`). De façon duale, le concept « personne dont tous les amis ne sont pas docteurs » subsume le concept « personne dont tous les amis ne sont pas bacheliers » (figure 11.4, `(3)`).

`[atleast]` : si $\mathtt{n_c} \leq \mathtt{n_d}$ et $\mathtt{r_d} \sqsubseteq \mathtt{r_c}$, alors $(\mathtt{atleast}\ \mathtt{n_d}\ \mathtt{r_d}) \sqsubseteq (\mathtt{atleast}\ \mathtt{n_c}\ \mathtt{r_c})$.

Par exemple, le concept « personne qui a au moins un enfant » subsume le concept « personne qui a au moins deux filles », où le rôle `fille` est subsumé par le rôle `enfant` (figure 11.4, `(4)`).

`[atmost]` : si $\mathtt{n_d} \leq \mathtt{n_c}$ et $\mathtt{r_c} \sqsubseteq \mathtt{r_d}$, alors $(\mathtt{atmost}\ \mathtt{n_d}\ \mathtt{r_d}) \sqsubseteq (\mathtt{atmost}\ \mathtt{n_c}\ \mathtt{r_c})$.

Par exemple, le concept « personne qui a au plus deux filles » subsume le concept « personne qui a au plus un enfant » (figure 11.4, `(5)`).

### 11.2.7 La notion de base de connaissances terminologique

#### Introduction générale et exemples

D'une façon générale, et dans les logiques de descriptions en particulier, la représentation de connaissances s'articule autour de deux niveaux :

- Le *niveau terminologique* où sont introduites les définitions de concepts et de rôles : ce niveau est relatif à l'*intension* des concepts et des rôles.
- Le niveau *factuel* ou niveau des *assertions*, où sont introduits les individus et les faits dans lesquels ces individus interviennent : ce niveau, appelé aussi *ABox*, est relatif à l'extension des concepts.

Au niveau factuel figurent essentiellement des spécifications d'instances et des relations entre instances. Ainsi, deux types d'assertions se rencontrent : les premières sont de la forme `C(a)` où `C` est un nom de concept et `a` un nom d'individu ; les secondes sont de la forme `r(a,b)` où `r` est un nom de rôle, et `a` et `b` celui de deux individus. Intuitivement, l'assertion `C(a)` stipule que `a` dénote une instance du concept `C` (`a` est dans l'extension de `C`) ; l'assertion `r(a,b)` stipule que les individus `a` et `b` sont en relation par l'intermédiaire de `r` (le couple `(a,b)` est dans l'extension de `r`).

```
Équipe-moderne(TRIO-54)
Homme(ANTOINE)
Personne(COLETTE)
membre(TRIO-54,ANTOINE)
membre(TRIO-54,PATRICK)
chef(TRIO-54,COLETTE)
(atmost 3 membre)(TRIO-54)
```

FIG. 11.5: *Un ensemble d'assertions. La dernière assertion signifie en particulier que* `TRIO-54` *est un instance de l'expression conceptuelle* `(atmost 3 membre)`.

Un ensemble d'assertions est donné en exemple à la figure 11.5. L'étude de ces assertions a pour but de montrer le degré de subtilité que peut recouvrir l'*instanciation*, qui est une des formes du raisonnement terminologique. L'instanciation consiste à déterminer qu'un objet `o` est instance d'un concept `C`, ou bien encore que l'assertion `C(o)` est vérifiée. La définition d'instances repose sur l'*hypothèse du nom unique* — les noms `COLETTE`, `ANTOINE` et `PATRICK` font référence à des individus différents — et sur l'*hypothèse du monde ouvert* [6] : une instanciation est partielle et non définitive, et elle peut être complétée le cas échéant. Par exemple, il est suffisant de savoir que `TRIO-54` est instance du concept `(atmost 3 membre)`, sans avoir à nommer explicitement les membres de `TRIO-54`, du moins, tant que cette donnée n'est pas indispensable.

Dans la pratique, plusieurs cas d'instanciation se présentent, dont les plus courants sont expliqués ci-dessous (voir aussi [Nebel, 1990a] et comparer avec les règles générales d'inférence par classification données dans le *chapitre 12*) :

- $[\mathtt{I}_1]$ : si l'assertion `D(o)` est connue et si `D` est subsumé par `C`, alors l'assertion `C(o)` en découle. Par exemple, `TRIO-54`, qui est une instance du concept `Équipe-moderne`, est aussi une instance du concept `Petite-équipe`, qui lui-même subsume `Équipe-moderne` (voir figure 11.1).
- $[\mathtt{I}_2]$ : si les assertions $\mathtt{D}_1(\mathtt{o})$ et $\mathtt{D}_2(\mathtt{o})$ sont déclarées, et si `C` ne subsume ni $\mathtt{D}_1$ ni $\mathtt{D}_2$, mais subsume $(\mathtt{and}\ \mathtt{D}_1\ \mathtt{D}_2)$, alors l'assertion `C(o)` en découle. Par exemple, supposons que $\mathtt{D}_1$ = `Docteur`, $\mathtt{D}_2$ = `Musicien`, `C` = `(and Bachelier Artiste)`, et que `COLETTE` est une instance de $\mathtt{D}_1$ et de $\mathtt{D}_2$. Dans ce cas, `C` ne subsume ni $\mathtt{D}_1$ ni $\mathtt{D}_2$, mais `C` subsume $(\mathtt{and}\ \mathtt{D}_1\ \mathtt{D}_2)$, et `COLETTE` est une instance de `C`.
- $[\mathtt{I}_3]$ : il est possible de profiter du fait qu'un objet $\mathtt{o}_1$ est en relation avec un objet $\mathtt{o}_2$ pour déterminer une instanciation. Ainsi, alors que ce n'est pas explicitement spécifié, `PATRICK` et `ANTOINE`, qui sont membres de `TRIO-54`, sont nécessairement des instances de `Personne`, car tous les membres d'une

6. Un fait n'est faux que si cela a été effectivement établi. L'hypothèse du monde ouvert s'oppose à l'*hypothèse du monde clos*, qui stipule que tout fait non déclaré, donc non connu, est considéré comme faux.

équipe sont des personnes. Cette instanciation repose sur le principe suivant : si une assertion de la forme $\mathtt{r}(\mathtt{o}_1, \mathtt{o}_2)$ est connue, et si $\mathtt{o}_2$ est une instance d'un concept de la forme `(all r C)` ou encore `(all r D)` avec `D` subsumé de `C`, alors $\mathtt{C}(\mathtt{o}_1)$ en découle. Par exemple, `COLETTE`, qui est déclarée comme `chef` de `TRIO-54`, est forcément une instance du concept `Femme`. Ici, $\mathtt{o}_1$ correspond à `COLETTE`, $\mathtt{o}_2$ à `TRIO-54` et `r` à `chef`. Puisque `TRIO-54` est instance de l'expression `(all chef Femme)`, qui subsume le concept `Équipe-moderne`, l'instanciation `Femme(COLETTE)` en découle.

Un traitement plus systématique de l'instanciation est donné au paragraphe 11.3, où est étudiée une série de règles qui traitent toutes les constructions possibles de $\mathcal{ALCNR}$.

**Base terminologique et modèle**

En l'absence de circuit terminologique, il est possible de normaliser les descriptions de façon à ne plus avoir que des concepts primitifs dans une description. Tout se passe alors comme s'il n'existait aucun concept défini : dans ce cas, les assertions peuvent devenir les ingrédients principaux d'une base de connaissances et du raisonnement terminologique.

Pour associer une sémantique aux assertions, la fonction d'interprétation $.^{\mathcal{I}}$ est prolongée aux individus, en faisant correspondre à un individu `a` un élément du domaine d'interprétation $\Delta_{\mathcal{I}}$ ($\mathtt{a}^{\mathcal{I}} \in \Delta_{\mathcal{I}}$) de telle façon que si $\mathtt{a} \neq \mathtt{b}$ alors $\mathtt{a}^{\mathcal{I}} \neq \mathtt{b}^{\mathcal{I}}$ ; deux individus différents sont interprétés comme des objets différents de $\Delta_{\mathcal{I}}$.

**Définition 4** ***(Satisfiabilité d'une assertion, base terminologique et modèle)***

- *Si* $\mathcal{I} = (\Delta_{\mathcal{I}}, .^{\mathcal{I}})$ *est une interprétation, l'assertion* $\mathtt{C}(\mathtt{a})$ *est* satisfaite *par* $\mathcal{I}$*, ou* satisfiable*, si et seulement si* $\mathtt{a}^{\mathcal{I}} \in \mathtt{C}^{\mathcal{I}}$ *; l'assertion* $\mathtt{r}(\mathtt{a}, \mathtt{b})$ *est* satisfaite *par* $\mathcal{I}$*, ou* satisfiable*, si et seulement si* $(\mathtt{a}^{\mathcal{I}}, \mathtt{b}^{\mathcal{I}}) \in \mathtt{r}^{\mathcal{I}}$.
- *Une* base (de connaissances) terminologique $\Sigma$ *pour* $\mathcal{ALCNR}$ *est un ensemble d'assertions de la forme* $\mathtt{C}(\mathtt{a})$ *ou* $\mathtt{r}(\mathtt{a}, \mathtt{b})$.
- *Une interprétation* $\mathcal{I}$ *est un* modèle *pour* $\Sigma$ *si et seulement si toute assertion* $\alpha$ *est satisfaite par* $\mathcal{I}$.
- *Une base terminologique est* satisfiable *si elle admet un modèle.*
- *L'assertion* $\alpha$ *est* logiquement impliquée *par la base* $\Sigma$*, ce qui s'écrit* $\Sigma \models \alpha$*, si et seulement si* $\alpha$ *est satisfaite par tout modèle de* $\Sigma$.

Nous utiliserons également dans la suite l'expression *système terminologique* pour désigner un système combinant un niveau terminologique et un niveau factuel, avec un langage de description de concepts et de rôles donné.

Considérons par exemple la base terminologique $\Sigma$ suivante :

```
Σ = {enfant(PIERRE,MARIE)
    (all enfant (not Musicien))(PIERRE)
    (and Femme (some enfant))(MARIE)}
```

Intuitivement, les trois assertions indiquent que MARIE est l'enfant de PIERRE, que tous les enfants de PIERRE sont non musiciens et que MARIE est une femme ayant un enfant. La base $\Sigma$ est satisfiable, puisqu'il existe une interprétation $\mathcal{I}$ donnée ci-dessous, qui satisfait toutes les assertions figurant dans $\Sigma$ et qui est donc un modèle pour $\Sigma$ :

$\Delta_{\mathcal{I}} = \{\texttt{PIERRE,MARIE}\}$
$\texttt{PIERRE}^{\mathcal{I}} = \texttt{PIERRE}$
$\texttt{MARIE}^{\mathcal{I}} = \texttt{MARIE}$
$\texttt{enfant}^{\mathcal{I}} = \{\texttt{(PIERRE,MARIE)}\}$
$\texttt{Femme}^{\mathcal{I}} = \{\texttt{MARIE}\}$
$\texttt{Musicien}^{\mathcal{I}} = \emptyset$

L'adjonction d'une assertion comme Musicien(MARIE) rendrait la base non satisfiable, car $\Sigma \models$ (not Musicien)(MARIE) (il est possible de montrer ce résultat en employant la règle $\texttt{I}_3$ explicitée au paragraphe précédent).

### 11.2.8 Un bilan intermédiaire sur le raisonnement terminologique

**Les opérations associées au raisonnement terminologique**

Les principales opérations liées au raisonnement terminologique — ou raisonnement dans un système terminologique — sont les suivantes [Buchheit *et al.*, 1993] [Donini *et al.*, 1994] :

- Le *test de subsomption* permet de vérifier qu'un concept C subsume un concept D. Le test de subsomption est à la base de l'opération de *classification*, qui consiste à déterminer l'ensemble des ascendants directs d'un concept dans la hiérarchie des concepts, les subsumants les plus spécifiques ou SPS, et l'ensemble des descendants immédiats du concept, les subsumés les plus généraux ou SPG [Baader *et al.*, 1994].

  Vérifier qu'un concept C vérifie une propriété prop se ramène également à un test de subsomption : C vérifie la propriété prop si un des subsumants de C vérifie prop.
- Le *test de satisfiabilité d'un concept* C permet de vérifier qu'un concept C admet des instances (il existe au moins une interprétation $\mathcal{I}$ telle que $\texttt{C}^{\mathcal{I}} \neq \emptyset$).
- Le *test de satisfiabilité d'une base terminologique* $\Sigma$ permet de vérifier que $\Sigma$ admet un modèle $\mathcal{I}$ : tout concept et toute assertion sont satisfaits par $\mathcal{I}$.
- Le *test d'instanciation* permet de vérifier qu'un individu a est instance d'un concept C pour une base terminologique $\Sigma$ : $\Sigma \models \texttt{C(a)}$. Plus précisément, le test d'instanciation consiste à retrouver les concepts les plus spécifiques dont un individu a est instance.

Les quatre opérations de tests ne sont pas indépendantes les unes des autres :

(1) C est satisfiable $\Longleftrightarrow$ $\texttt{C} \not\sqsubseteq \bot$,

(2) $\mathtt{C}$ est satisfiable $\Longleftrightarrow \{\mathtt{C(a)}\}$ est satisfiable,
(3) $\mathtt{D} \sqsubseteq \mathtt{C} \Longleftrightarrow \mathtt{D} \sqcap \neg\mathtt{C}$ n'est pas satisfiable,
(4) $\mathtt{D} \sqsubseteq \mathtt{C} \Longleftrightarrow \{\mathtt{C(a)}\} \models \mathtt{D(a)}$,
(5) $\Sigma$ est satisfiable $\Longleftrightarrow \Sigma \not\models \bot\mathtt{(a)}$,
(6) $\Sigma \models \mathtt{C(a)} \Longleftrightarrow \Sigma \cup \{\neg\mathtt{C(a)}\}$ non satisfiable.

Le test de satisfiabilité d'un concept se ramène au problème complémentaire du test de subsomption (1) ou à un test d'instanciation (2). Le test de subsomption se ramène au test de satisfiabilité d'une conjonction de concepts (3) — qui suppose que le langage de description des concepts inclut la négation de concepts définis — ou à un test d'instanciation (4). Le test de satisfiabilité d'une base terminologique se ramène au problème complémentaire d'un test d'instanciation (5). Et finalement, le test d'instanciation se ramène au problème complémentaire du test de satisfiabilité d'une base terminologique (6).

Il est important de remarquer que les équivalences données ci-dessus ne sont plus valides dès que le langage de description des concepts et des rôles fait intervenir directement des instances dans la définition des concepts. Ce n'est pas le cas pour $\mathcal{ALCNR}$, mais ça l'est pour CLASSIC ou LOOM, où sont admis des constructeurs comme `fills` qui attache une valeur à un rôle, `one-of` qui permet de construire des domaines de valeurs par énumération, ou encore `same-as` qui indique qu'un rôle a les mêmes valeurs élémentaires qu'un autre rôle.

### La complexité du raisonnement terminologique

Depuis le milieu des années 80 jusqu'à aujourd'hui, la complexité du raisonnement terminologique et plus spécialement la complexité de la subsomption [7] ont fait couler beaucoup d'encre et ont été des moteurs de la recherche sur les logiques de descriptions. Ces premiers travaux s'intéressent aux propriétés de correction, de complétude, et à la complexité algorithmique de la subsomption, en tant que procédure d'inférence d'un système terminologique considéré comme un système déductif.

Rappelons qu'un système déductif est *correct* si les inférences produites sont en accord avec la sémantique associée au système, autrement dit, ce qui est vrai sur le plan syntaxique l'est sur le plan sémantique ; un système déductif est *complet* si toutes les formules valides — vraies sur le plan sémantique — peuvent être démontrées sur le plan syntaxique [Loveland, 1978].

Considérons à présent les concepts $\mathtt{C}$ pour « personne ayant au moins deux enfants » et $\mathtt{D}$ pour « personne ayant au moins une fille et un fils », qui se représentent en $\mathcal{ALCNR}$ par :

```
C ≐ (and Personne
         (atleast 2 enfant))

D ≐ (and Personne
```

7. Il faudrait dire plus précisément : « la complexité des algorithmes détectant les relations de subsomption entre descriptions ».

```
        (atleast 1 fille)
        (atleast 1 fils))
```

Si le bon sens permet d'affirmer que toute instance de D est une instance de C, autrement dit que C subsume D, les règles algorithmiques données au paragraphe 11.2.6 ne permettent pas de détecter une telle relation de subsomption. Un système terminologique muni d'une procédure de subsomption s'appuyant sur un tel algorithme est alors *incomplet*, ce qui pose le problème suivant : si la procédure de subsomption retourne `vrai`, alors le résultat est exploitable, tandis qu'il ne l'est plus si la procédure retourne `faux` (une relation de subsomption peut être vraie et ne pas être détectée par la procédure).

Le problème de la non complétude de la subsomption a été soulevé une première fois pour KL-ONE dans [Schmolze et Lipkis, 1983]. Ces travaux ont été repris et formalisés dans [Brachman et Levesque, 1984] [Levesque et Brachman, 1987] et [Nebel, 1990a]. Il est montré dans ces articles que le test de subsomption est correct, complet et de complexité polynômiale pour les langages $\mathcal{FL}\text{-} = \{\mathtt{C} \sqcap \mathtt{D}, \forall \mathtt{r.C}, \exists \mathtt{r}\}$ et $\mathcal{FLN} = \{\mathtt{C} \sqcap \mathtt{D}, \forall \mathtt{r.C}, \geq \mathtt{n\,r}, \leq \mathtt{n\,r}\}$ ; le test de subsomption devient de complexité exponentielle et les algorithmes (polynômiaux) proposés sont incomplets dès que des constructeurs portant sur les rôles comme `and` ou `restrict` sont ajoutés à $\mathcal{FL}$- ou à $\mathcal{FLN}$.

Au début des années 90, de nombreux articles traitent du classement des problèmes liés au raisonnement terminologique : subsomption, instanciation, test de satisfiabilité d'un concept ou d'une base terminologique. Des tables recensant ces résultats sont données dans [Woods et Schmolze, 1992] [Heinsohn *et al.*, 1994], [Donini *et al.*, 1994] et [Donini *et al.*, 1997]. Brièvement, le raisonnement terminologique est de complexité polynômiale pour des langages comme $\mathcal{AL}$, $\mathcal{ALN}$ et $\mathcal{FL}$-, mais aussi pour le langage $\mathcal{PL}_1 = \{\top, \bot, \neg\mathtt{A}, \mathtt{C} \sqcap \mathtt{D}, \forall \mathtt{r.C}, \geq \mathtt{n\,r}, \leq \mathtt{n\,r}, \mathtt{r}^{-1}\}$, où A désigne un concept primitif et $\mathtt{r}^{-1}$ dénote la relation inverse pour un rôle, et pour le langage $\mathcal{PL}_2 = \{\mathtt{C} \sqcap \mathtt{D}, \forall \mathtt{r.C}, \exists \mathtt{r}, \mathtt{r}_1 \sqcap \mathtt{r}_2, \mathtt{r}^{-1}, \mathtt{r}_1 \circ \mathtt{r}_2\}$, où $\mathtt{r}_1 \circ \mathtt{r}_2$ dénote la composition de rôles [Donini *et al.*, 1991b]. Les ensembles de constructeurs associés à $\mathcal{PL}_1$ et $\mathcal{PL}_2$ constituent des ensembles « maximaux » à ne pas étendre au risque de passer à une complexité exponentielle. En dehors de ces cas simples, les problèmes liés au raisonnement terminologique sont de complexité exponentielle[8], selon les cas NP-difficiles, co-NP-difficiles, NP-complets, ou dans la classe PSPACE [Donini *et al.*, 1991a] [Schmidt-Schauß et Smolka, 1991] [Donini *et al.*, 1997] voire indécidables [Patel-Schneider, 1989] [Nebel, 1990b].

Les travaux récents sur la subsomption, comme [Buchheit *et al.*, 1993], [Donini *et al.*, 1994] et [Donini *et al.*, 1997], cherchent à exploiter la méthode des tableaux sémantiques pour mettre en valeur des algorithmes complets, comme cela est expliqué dans le paragraphe suivant. Outre leur intérêt théorique, tous les travaux sur la complexité de la subsomption ont permis de mettre en évidence les rapports qui existent entre la complexité du raisonnement terminologique et la richesse — en termes de nombre et de types de constructeurs — des langages de description de concepts et de rôles. Face à ces résultats, deux attitudes ont guidé les concepteurs

8. Voir [Garey et Johnson, 1979] pour les définitions sur la complexité algorithmique.

de logiques de descriptions : avoir un langage de description plutôt pauvre mais une procédure d'inférence correcte, complète et de complexité polynômiale comme pour CLASSIC [Borgida et Patel-Schneider, 1994], ou au contraire avoir un langage de description riche et une procédure d'inférence de complexité exponentielle, complète comme pour KRIS [Baader et Hollunder, 1991a], ou non complète comme pour BACK [Peltason, 1991] et LOOM [MacGregor, 1991]. Les deux attitudes se justifient et nous renvoyons le lecteur intéressé par ce débat aux articles déjà cités, mais aussi à [Doyle et Patil, 1991].

## 11.3 La méthode des tableaux sémantiques dans les logiques de descriptions

Dans ce paragraphe, nous montrons comment la méthode des *tableaux sémantiques* peut être appliquée dans le cadre des logiques de descriptions. En vertu des correspondances exhibées au paragraphe 11.2.8, les quatre opérations de test se ramènent à l'une d'entre elles, en l'occurrence le test de satisfiabilité d'une base terminologique. La vérification de ce test de satisfiabilité va s'appuyer sur la méthode des tableaux sémantiques, dont les principes généraux pour la logique classique sont donnés dans [Fitting, 1990] ou [Gochet et Gribomont, 1991]. Globalement, la méthode des tableaux sémantiques est une méthode de réfutation analytique : pour prouver une formule $\varphi$, il est supposé que $\varphi$ est fausse et les conséquences de cette hypothèse sur les constituants de $\varphi$ sont examinées. La méthode s'appuie sur un ensemble de *règles de décomposition* qui permettent d'assigner une valeur de vérité aux constituants de la formule $\varphi$, une fois fixée la valeur de vérité de la formule $\varphi$ elle-même.

#### Satisfiabilité d'une base terminologique et système de contraintes

Appliquer la méthode des tableaux sémantiques pour satisfaire une base terminologique $\Sigma$ se fait en trois étapes principales :

- transformation de $\Sigma$ en un *système de contraintes* $\mathtt{S}_\Sigma$,
- application systématique d'un ensemble de règles de décomposition jusqu'à *saturation* du système de contraintes ; plus aucune règle de décomposition n'est alors applicable et le système de contraintes est dit *complet*,
- évaluation : si le système complet obtenu ne contient pas de motif de contradiction, il est satisfiable ou non contradictoire.

Avant toute chose, les concepts de la base terminologique considérée $\Sigma$ sont mis sous *forme simple* [Schmidt-Schauß et Smolka, 1991] [Donini *et al.*, 1997] : les seules négations dans une description sont de la forme $\neg\mathtt{A}$, où $\mathtt{A}$ désigne un concept primitif. C'est après cette première transformation qu'est construit le système de contraintes $\mathtt{S}_\Sigma$ associé à $\Sigma$. Les règles du processus de normalisation sont détaillées ci-après :

$$\begin{aligned}
\neg\top &\longrightarrow \bot \\
\neg\bot &\longrightarrow \top \\
\neg(\mathtt{C} \sqcap \mathtt{D}) &\longrightarrow \neg\mathtt{C} \sqcup \neg\mathtt{D} \\
\neg(\mathtt{C} \sqcup \mathtt{D}) &\longrightarrow \neg\mathtt{C} \sqcap \neg\mathtt{D} \\
\neg\neg\mathtt{C} &\longrightarrow \mathtt{C} \\
\neg(\forall\mathtt{r}.\mathtt{C}) &\longrightarrow \exists\mathtt{r}.\neg\mathtt{C} \\
\neg(\exists\mathtt{r}.\mathtt{C}) &\longrightarrow \forall\mathtt{r}.\neg\mathtt{C} \\
\neg(\geq \mathtt{n}\,\mathtt{r}) &\longrightarrow \forall\mathtt{r}.\bot \ \textit{si}\ \mathtt{n} = 1 \\
\neg(\geq \mathtt{n}\,\mathtt{r}) &\longrightarrow (\leq \mathtt{n} - 1\,\mathtt{r})\ \textit{si}\ \mathtt{n} > 1 \\
\neg(\leq \mathtt{n}\,\mathtt{r}) &\longrightarrow (\geq \mathtt{n} + 1\,\mathtt{r})
\end{aligned}$$

La transformation d'une base terminologique $\Sigma$ en un système de contraintes $\mathtt{S}_\Sigma$ nécessite l'utilisation de *variables* qui permettent de définir les *contraintes*. Une contrainte peut être de trois formes — $\mathtt{x} : \mathtt{C}$, $\mathtt{xry}$, $\mathtt{x} \neq \mathtt{y}$ — où $\mathtt{C}$ dénote un concept, $\mathtt{r}$ un rôle, $\mathtt{x}$ et $\mathtt{y}$ des instances ou des variables. Intuitivement, la contrainte $\mathtt{x} : \mathtt{C}$ exprime une instanciation, en l'occurrence que $\mathtt{x}$ est une instance de $\mathtt{C}$. La contrainte $\mathtt{xry}$ exprime que $\mathtt{x}$ est en relation avec $\mathtt{y}$ par l'intermédiaire du rôle $\mathtt{r}$ ; $\mathtt{y}$ est alors appelé un *r-successeur* de $\mathtt{x}$. La contrainte $\mathtt{x} \neq \mathtt{y}$ exprime que les individus $\mathtt{x}$ et $\mathtt{y}$ ont forcément une interprétation différente ; $\mathtt{x}$ et $\mathtt{y}$ sont alors dits *séparés*.

Un *système de contraintes* est un ensemble fini non vide de contraintes des trois formes $\mathtt{x} : \mathtt{C}$, $\mathtt{xry}$ et $\mathtt{x} \neq \mathtt{y}$. Les variables intervenant dans un système de contraintes sont supposées être ordonnées en fonction de leur ordre d'apparition dans le système [Buchheit *et al.*, 1993]. De plus, deux variables $\mathtt{x}$ et $\mathtt{y}$ sont dites *équivalentes* dans $\mathtt{S}_\Sigma$ si elles vérifient les mêmes instanciations : $\{\mathtt{C}/\mathtt{x} : \mathtt{C} \in \mathtt{S}_\Sigma\} = \{\mathtt{C}/\mathtt{y} : \mathtt{C} \in \mathtt{S}_\Sigma\}$.

Sur le plan pratique, une assertion $\mathtt{C}(\mathtt{a})$ est transformée en une contrainte $\mathtt{a} : \mathtt{C}$, et une assertion $\mathtt{r}(\mathtt{a}, \mathtt{b})$ est transformée en une contrainte $\mathtt{arb}$ ou $\{\mathtt{ar_1b}, \mathtt{ar_2b}\}$ selon que $\mathtt{r}$ est atomique ou que $\mathtt{r} = \mathtt{r_1} \sqcap \mathtt{r_2}$. Étant donné une base terminologique $\Sigma$, un premier système de contraintes $\mathtt{S}_\Sigma$ est construit en transformant les assertions comme indiqué ci-avant. Ensuite, de nouvelles contraintes — et donc de nouveaux systèmes de contraintes — sont produites par l'application de règles de décomposition aux contraintes existantes.

**Les règles de décomposition**

Les six règles de décomposition qui sont couramment utilisées dans le cadre de $\mathcal{ALCNR}$ sont présentées à la figure 11.6 [Buchheit *et al.*, 1993] [Donini *et al.*, 1997]. Le système $\mathtt{S}_\Sigma$ y est noté plus simplement $\mathtt{S}$, et pour chaque règle sont décrites les conditions d'application et les conséquences de l'application de la règle sur le système de contraintes $\mathtt{S}$.

Les règles $\rightarrow_\exists$ et $\rightarrow_\geq$ sont dites *génératrices*, car elles introduisent de nouvelles variables dans le système de contraintes. En particulier, la règle $\rightarrow_\geq$ indique que, si le nombre minimal de r-successeurs de $\mathtt{x}$ n'est pas atteint, alors il peut être augmenté.

- Conjonction de concepts :
  $\mathtt{S} \rightarrow_{\sqcap} \mathtt{S} \cup \{\mathtt{x} : \mathtt{C}_1, \mathtt{x} : \mathtt{C}_2\}$,
  si 1. $\mathtt{x} : \mathtt{C}_1 \sqcap \mathtt{C}_2$ est dans $\mathtt{S}$,
  2. les contraintes $\mathtt{x} : \mathtt{C}_1$ et $\mathtt{x} : \mathtt{C}_2$ ne sont pas toutes deux dans $\mathtt{S}$.
- Disjonction de concepts :
  $\mathtt{S} \rightarrow_{\sqcup} \mathtt{S} \cup \{\mathtt{x} : \mathtt{D}\}$,
  si 1. $\mathtt{x} : \mathtt{C}_1 \sqcup \mathtt{C}_2$ est dans $\mathtt{S}$,
  2. ni $\mathtt{x} : \mathtt{C}_1$ ni $\mathtt{x} : \mathtt{C}_2$ ne sont dans $\mathtt{S}$,
  3. $\mathtt{D} = \mathtt{C}_1$ ou $\mathtt{C}_2$.
- Quantification universelle :
  $\mathtt{S} \rightarrow_{\forall} \mathtt{S} \cup \{\mathtt{z} : \mathtt{C}\}$,
  si 1. $\mathtt{x} : \forall \mathtt{r}.\mathtt{C}$ est dans $\mathtt{S}$,
  2. $\mathtt{xrz}$ est dans $\mathtt{S}$,
  3. $\mathtt{z} : \mathtt{C}$ n'est pas dans $\mathtt{S}$.
- Quantification existentielle :
  $\mathtt{S} \rightarrow_{\exists} \mathtt{S} \cup \{\mathtt{xr_1y}, ..., \mathtt{xr_ky}, \mathtt{y} : \mathtt{C}\}$,
  si 1. $\mathtt{r} = \mathtt{r}_1 \sqcap \mathtt{r}_2 ... \sqcap \mathtt{r}_k$,
  2. $\mathtt{x} : \exists \mathtt{r}.\mathtt{C}$ est dans $\mathtt{S}$,
  3. $\mathtt{y}$ est une nouvelle variable,
  4. il n'existe pas $\mathtt{z}$ tel que les contraintes $\mathtt{xrz}$ et $\mathtt{z} : \mathtt{C}$ soient toutes deux dans $\mathtt{S}$,
  5. si $\mathtt{x}$ est une variable, alors il n'existe aucune variable $\mathtt{z}$ introduite *avant* $\mathtt{x}$ dans $\mathtt{S}$ qui soit équivalente à $\mathtt{x}$ dans $\mathtt{S}$.
- Cardinalité minimale :
  $\mathtt{S} \rightarrow_{\geq} \mathtt{S} \cup \{\mathtt{xr_1y_i}, ..., \mathtt{xr_ky_i} | \mathtt{i} \in \mathtt{1..n}\} \cup \{\mathtt{y_i} \not\doteq \mathtt{y_j} | \mathtt{i}, \mathtt{j} \in \mathtt{1..n}, \mathtt{i} \neq \mathtt{j}\}$,
  si 1. $\mathtt{r} = \mathtt{r}_1 \sqcap \mathtt{r}_2 ... \sqcap \mathtt{r}_k$,
  2. $\mathtt{x} : (\geq \mathtt{n}\,\mathtt{r})$ est dans $\mathtt{S}$,
  3. $\mathtt{y}_1$,...,$\mathtt{y}_n$ sont des nouvelles variables,
  4. il n'existe pas dans $\mathtt{S}$ $\mathtt{n}$ r-successeurs de $\mathtt{x}$ séparés et deux à deux distincts,
  5. si $\mathtt{x}$ est une variable, alors il n'existe aucune variable $\mathtt{z}$ introduite *avant* $\mathtt{x}$ dans $\mathtt{S}$ qui soit équivalente à $\mathtt{x}$ dans $\mathtt{S}$.
- Cardinalité maximale :
  $\mathtt{S} \rightarrow_{\leq} \mathtt{S}[\mathtt{y}/\mathtt{z}]$
  si 1. $\mathtt{x} : (\leq \mathtt{n}\,\mathtt{r})$ est dans $\mathtt{S}$,
  2. $\mathtt{x}$ a plus de $\mathtt{n}$ r-successeurs dans $\mathtt{S}$,
  3. $\mathtt{y}$ et $\mathtt{z}$ sont deux r-successeurs de $\mathtt{x}$ dans $\mathtt{S}$ non séparés.

FIG. 11.6: *Les six règles de décomposition de la méthode des tableaux sémantiques pour $\mathcal{ALCNR}$.*

Les règles $\rightarrow_{\sqcup}$ et $\rightarrow_{\leq}$ sont, quant à elles, *non déterministes*, car elles peuvent être appliquées à un système de contraintes de façons différentes [9]. Dans la règle $\rightarrow_{\leq}$, $\mathtt{S}[\mathtt{y}/\mathtt{z}]$ dénote le système de contraintes obtenu à partir de $\mathtt{S}$ en substituant chaque occurrence de la variable $\mathtt{y}$ par la variable $\mathtt{z}$. Cette règle indique que certains des r-successeurs de $\mathtt{x}$, à condition qu'ils ne soient pas séparés, peuvent être *identifiés* pour que la contrainte sur la cardinalité maximale du rôle $\mathtt{r}$ soit respectée.

### Satisfiabilité et système de contraintes contradictoire

Les règles de décomposition sont appliquées jusqu'à ce que le système de contraintes $\mathtt{S}_{\Sigma}$ devienne complet et plus aucune règle n'est alors applicable. Deux cas peuvent se produire : soit le système est contradictoire et la base terminologique $\Sigma$ associée n'est pas satisfiable, soit le système est non contradictoire ; la base $\Sigma$ est alors satisfiable et un modèle de $\Sigma$ peut être construit. La définition suivante précise les motifs de contradiction.

**Définition 5** ***(Système de contraintes contradictoire et contradictions)***

*Un système de contraintes est* contradictoire *s'il renferme un des motifs de contradiction suivant :*

*(1)* $\{\mathtt{x} : \bot\}$.

*(2)* $\{\mathtt{x} : \mathtt{A}, \mathtt{x} : \neg\mathtt{A}\}$.

*(3)* $\{\mathtt{x} : (\leq \mathtt{m}\,\mathtt{r})\}$ *et le nombre de r-successeurs de* $\mathtt{x}$ *est strictement plus grand que* $\mathtt{m}$.

*(4)* $\{\mathtt{x} : (\geq \mathtt{m}\,\mathtt{r}), \mathtt{x} : (\leq \mathtt{n}\,\mathtt{r})\}$, *où* $\mathtt{n} < \mathtt{m}$.

De façon pratique, une base terminologique $\Sigma$ est satisfiable si et seulement s'il existe un système de contraintes complet et non contradictoire $\mathtt{S}_{\Sigma}$ engendré à partir de $\Sigma$. Le problème est alors décidable dans le cadre de $\mathcal{ALCNR}$ et le résultat théorique est démontré dans [Schmidt-Schauß et Smolka, 1991], [Buchheit *et al.*, 1993] et [Donini *et al.*, 1997]). Lorsque le système complet $\mathtt{S}_{\Sigma}$ est non contradictoire, il est possible de construire un modèle de $\Sigma$ à partir de $\mathtt{S}_{\Sigma}$.

### Traitement d'un exemple

Considérons la base terminologique $\Sigma$ dont il faut tester la satisfiabilité ; cet exemple, où tous les concepts et les rôles employés sont primitifs, est tiré de [Donini *et al.*, 1994] :

```
Σ = {enfant(PIERRE,MARIE),
     (all enfant (not Musicien))(PIERRE),
     (and (some enfant Homme)
          (some ami (or Cycliste Alpiniste)))(MARIE)}
```

9. Ces règles correspondent aux règles de *ramification* des tableaux sémantiques en logique classique, par opposition aux règles de *prolongation* [Gochet et Gribomont, 1991].

Intuitivement, cette base représente trois faits : MARIE est l'enfant de PIERRE, tous les enfants de PIERRE ne sont pas musiciens, MARIE a un garçon — un enfant de sexe masculin — et un ami qui est cycliste ou alpiniste. Cette base est satisfiable, ce qui va être confirmé formellement ci-dessous.

La notation allemande est généralement utilisée pour vérifier la satisfiabilité d'une base terminologique avec la méthode des tableaux sémantiques. La base $\Sigma$ est donc présentée en notation allemande suivie du système de contraintes $S_\Sigma$ associé.

$$\begin{aligned}\Sigma = \{&\mathtt{enfant}(\mathtt{PIERRE}, \mathtt{MARIE}),\\ &\forall\mathtt{enfant}.\neg\mathtt{Musicien}(\mathtt{PIERRE}),\\ &\exists\mathtt{enfant}.\mathtt{Homme} \sqcap \exists\mathtt{ami}.\mathtt{Cycliste} \sqcup \mathtt{Alpiniste}(\mathtt{MARIE})\}\end{aligned}$$

$$\begin{aligned}S_\Sigma = \{&\mathtt{PIERRE}\ \mathtt{enfant}\ \mathtt{MARIE},\\ &\mathtt{PIERRE} : \forall\mathtt{enfant}.\neg\mathtt{Musicien},\\ &\mathtt{MARIE} : \exists\mathtt{enfant}.\mathtt{Homme} \sqcap \exists\mathtt{ami}.\mathtt{Cycliste} \sqcup \mathtt{Alpiniste}\}\end{aligned}$$

Étant donné un système de contraintes $\Sigma$, une stratégie d'application des règles est la suivante :

- Appliquer une règle à une variable lorsqu'aucune règle n'est applicable à une instance.
- Appliquer une règle à une variable x lorsqu'aucune règle n'est applicable à une variable y introduite avant x.
- Appliquer une règle génératrice lorsqu'aucune règle non génératrice n'est applicable.

Voilà une suite possible d'applications de règles de décomposition :

$$\begin{aligned}&\text{Règle} \rightarrow_\forall : S_1 &=&\ S_\Sigma \cup \{\mathtt{MARIE} : \neg\mathtt{Musicien}\}\\ &\text{Règle} \rightarrow_\sqcap : S_2 &=&\ S_1 \cup \{\mathtt{MARIE} : \exists\mathtt{enfant}.\mathtt{Homme},\\ &&&\qquad\quad \mathtt{MARIE} : \exists\mathtt{ami}.\mathtt{Cycliste} \sqcup \mathtt{Alpiniste}\}\\ &\text{Règle} \rightarrow_\exists : S_3 &=&\ S_2 \cup \{\mathtt{MARIE}\ \mathtt{enfant}\ \mathtt{x}, \mathtt{x} : \mathtt{Homme}\}\\ &\text{Règle} \rightarrow_\exists : S_4 &=&\ S_3 \cup \{\mathtt{MARIE}\ \mathtt{ami}\ \mathtt{y}, \mathtt{y} : \mathtt{Cycliste} \sqcup \mathtt{Alpiniste}\}\\ &\text{Règle} \rightarrow_\sqcup : S_5 &=&\ S_4 \cup \{\mathtt{y} : \mathtt{Cycliste}\}\end{aligned}$$

Le système de contraintes $S_5$ est complet et non contradictoire : la base terminologique $\Sigma$ est donc satisfiable. Un modèle de $\Sigma$ est le suivant :

$$\begin{aligned}&\Delta_\mathcal{I} = \{\mathtt{PIERRE}, \mathtt{MARIE}, \mathtt{x}, \mathtt{y}\}\\ &\mathtt{PIERRE}^\mathcal{I} = \mathtt{PIERRE}\\ &\mathtt{MARIE}^\mathcal{I} = \mathtt{MARIE}\\ &\mathtt{Homme}^\mathcal{I} = \{\mathtt{x}\}\\ &\mathtt{Musicien}^\mathcal{I} = \emptyset\\ &\mathtt{Cycliste}^\mathcal{I} = \{\mathtt{y}\}\\ &\mathtt{Alpiniste}^\mathcal{I} = \emptyset\\ &\mathtt{enfant}^\mathcal{I} = \{(\mathtt{PIERRE}, \mathtt{MARIE})\ (\mathtt{MARIE}, \mathtt{x})\}\\ &\mathtt{ami}^\mathcal{I} = \{(\mathtt{MARIE}, \mathtt{y})\}\end{aligned}$$

## 11.4 Discussions sur les logiques de descriptions

Dans ce paragraphe, nous présentons la famille des logiques de descriptions dans sa globalité, et les systèmes les plus connus et les plus originaux. Ensuite, nous évoquons la place des logiques de descriptions en représentation des connaissances et les rapports qu'entretiennent les logiques de descriptions avec des formalismes de représentation comme les logiques classiques et les systèmes de représentation de connaissances par objets (ou systèmes de RCO). Ensuite, nous discutons de l'apport des logiques de descriptions en représentation des connaissances, de quelques limitations, et nous terminons en évoquant certaines perspectives de recherche sur les logiques de descriptions.

### 11.4.1 La diversité des logiques de descriptions

Le formalisme des logiques de descriptions provient essentiellement des travaux entrepris par R.J. Brachman sur la représentation des connaissances, ainsi que sur le système KL-ONE et ses descendants [Brachman, 1979] [Brachman et Schmolze, 1985]. Dans KRYPTON, un descendant de KL-ONE, apparaissent pour la première fois les notions de *TBox*, de *ABox*, et les fonctionnalités que doit recouvrir une *ABox* [Brachman *et al.*, 1983]. Le système NIKL, pour *New Implementation of KL-ONE*, est une version épurée et améliorée de KL-ONE [Schmolze et Mark, 1991], avec laquelle sont mis en valeur les principes du raisonnement par classification et les liens existant entre instanciation et maintien de cohérence (toutefois, la subsomption dans NIKL reste indécidable [Patel-Schneider, 1989]). KANDOR [Patel-Schneider *et al.*, 1984] se présente comme un descendant de KRYPTON pour lequel les possibilités de représentation sont volontairement simplifiées, afin que les processus de classification et d'instanciation soient les plus efficaces possibles. En particulier, le système KANDOR se caractérise par un langage de description de concepts et de rôles limité et un niveau factuel où seules les instanciations de concepts et de rôles sont autorisées (ce qui va devenir le standard présenté au paragraphe 11.2.7).

Plus récemment, CLASSIC [Borgida *et al.*, 1989] [Brachman *et al.*, 1991] [Borgida et Patel-Schneider, 1994], le dernier né de la famille KL-ONE, intègre l'ensemble de l'expérience acquise au cours de la décennie précédente. La simplicité d'utilisation de CLASSIC, la fiabilité du système et la qualité de sa documentation ont valu à CLASSIC d'être utilisé dans de nombreuses applications et d'être devenu une des références en matière de logiques de descriptions.

À l'opposé, LOOM [MacGregor, 1991] s'affiche comme un système de représentation de connaissances universel, malgré les problèmes de complexité et de complétude que pose un tel parti-pris. Comme pour CLASSIC, de nombreuses applications ont été réalisées avec LOOM, qui est également devenu une référence en matière de logiques de descriptions.

Après CLASSIC et LOOM, il faut mentionner BACK, un système qui a été développé à la Technische Universität de Berlin dans la deuxième moitié des années 80 [Peltason, 1991] [Hoppe *et al.*, 1993]. Le système a eu une longue évolution, mais dès l'origine, ses concepteurs ont voulu faire de BACK un système fiable, utilisable dans des applications réalistes, notamment pour mettre en œuvre des systèmes d'information et de gestion de bases de données opérationnels. En dehors des études concernant la complexité de la subsomption, de nombreux problèmes relatifs aux logiques de descriptions ont été abordés par les membres du groupe travaillant sur BACK, comme les extensions temporelles pour une logique de descriptions [Schmiedel, 1990], les relations entre rôles, les différentes possibilités d'implantation d'une *ABox*, avec le retrait d'informations [Kindermann, 1992] et la représentation de règles.

À côté des trois figures de proue que sont CLASSIC, LOOM et BACK, il faut mentionner KRIS [Baader et Hollunder, 1991b] et K-REP [Dionne *et al.*, 1993]. Le système KRIS possède un langage de description de concepts et de rôles qui contient $\mathcal{ALCNR}$ et a servi de support à bon nombre d'études théoriques. Le système K-REP, quant à lui, a servi de support à des études théoriques sur le point de vue intensionnel de la subsomption. En particulier, K-REP est pourvu d'une sémantique algébrique, et un système équationnel fournit les propriétés de la subsomption.

Il n'est bien sûr pas possible de mentionner tous les systèmes existants, et nous renvoyons le lecteur intéressé aux catalogues établis dans [Rich, 1991] et dans [Woods et Schmolze, 1992].

### 11.4.2 Les logiques de descriptions et la représentation de connaissances

**Logiques de descriptions et logique classique**

Les éléments principaux qui caractérisent une logique de descriptions sont le langage de description des concepts et des rôles, l'interprétation associée aux expressions du langage et la relation de subsomption. Le parallèle avec un formalisme logique est alors naturel, puisqu'une logique s'appuie sur un langage de construction de formules (bien formées), une interprétation associée aux formules et des règles d'inférences. Les concepts d'une logique de descriptions peuvent être vus comme des prédicats unaires et les rôles comme des prédicats binaires, la subsomption comme une règle d'inférence ; la subsomption $\mathtt{D} \sqsubseteq \mathtt{C}$ est d'ailleurs quelquefois notée $\mathtt{D} \Rightarrow \mathtt{C}$ [Borgida, 1996]. Toutefois, le parallèle est justifié jusqu'à un certain point : une expression conceptuelle et une assertion peuvent se voir comme des cas particuliers de formules logiques du premier ordre, où ne figurent que des variables instanciées. En dehors de l'absence de variables, la manipulation des formules se fait essentiellement de la même façon en logique classique et dans une logique de descriptions.

Une dernière question concerne le *pouvoir expressif* des logiques de descriptions. Cette question est étudiée en détail dans [Borgida, 1996], qui montre qu'une logique de descriptions comme $\mathcal{ALCNR}$ a le même pouvoir expressif qu'un sous-ensemble de formules de la logique du premier ordre n'ayant que des prédicats unaires et binaires, et n'autorisant que l'usage de trois variables au plus (dont deux variables libres au plus). Nous n'avancerons pas plus dans ces considérations et renvoyons le lecteur intéressé à [Borgida, 1996].

**Logiques de descriptions et systèmes de RCO**

Les logiques de descriptions présentent une certain nombre de similarités avec les systèmes de RCO (qui sont présentés dans le *chapitre 10*). Un concept, considéré comme une conjonction de rôles, est comparable à une classe, considérée comme une conjonction de propriétés. Toutefois, les connecteurs `or` et `not`, qui font partie de $\mathcal{ALCNR}$, ne se rencontrent habituellement pas dans un système de RCO (pas plus que dans CLASSIC d'ailleurs). Cette absence rend les disjonctions et les incompatibilités difficiles à représenter (au moins de façon déclarative), bien que la sémantique d'un domaine de valeurs dans un système de RCO soit celle d'une disjonction de valeurs, les domaines spécialisants étant inclus dans les domaines spécialisés (voir *chapitre 12*), tout comme c'est le cas pour le constructeur `one-of`, existant en CLASSIC et en LOOM par exemple.

Un rôle peut se voir associer des restrictions qui portent sur son co-domaine et sa cardinalité, à l'instar des attributs d'une classe. En revanche, l'absence de caractère procédural, comme les méthodes ou les réflexes, différencie nettement logiques de descriptions et systèmes de RCO. Un système de RCO tire l'essentiel de son pouvoir expressif et déductif du mécanisme d'héritage, des réflexes associés aux attributs et des méthodes associées aux classes. De ce point de vue, l'opposition logiques de descriptions–systèmes de RCO s'apparente à la controverse déclaratif–procédural [Winograd, 1975]. Pourtant, les choses ne sont pas aussi tranchées : dans CLASSIC par exemple, il existe des fonctions test et des règles qui jouent un rôle comparable à celui de certains réflexes d'un système de RCO.

L'héritage est un mécanisme de partage de propriétés qui permet d'associer de nouvelles connaissances aux classes et à leurs instances, généralement en retrouvant une valeur, une méthode ou un réflexe dans la hiérarchie d'héritage. Une nouvelle classe se construit en spécialisant une ou plusieurs classes existantes (par adjonction ou redéfinition de propriétés). La mise en place d'une nouvelle classe dans la hiérarchie est laissée à l'entière liberté du programmeur, tandis que la mise en place d'un nouveau concept défini est à la charge du processus de classification. De plus, la recherche de propriétés ne constitue qu'un aspect contingent du pouvoir déductif lié à la subsomption et à la classification. Plus précisément, les propriétés associées à une classe sont considérées comme des conditions nécessaires et jamais comme des conditions suffisantes : une telle sémantique est dite *descriptive* (voir par exemple [Nebel, 1991] [Buchheit *et al.*, 1994] [De Giacomo et Lenzerini, 1997], et les *chapitres 10 et 12*). Dans les logiques de descriptions, les concepts primitifs ont aussi une sémantique descriptive, mais les concepts définis ont une sémantique *définitionnelle*, sur laquelle repose le mécanisme de classification. Ainsi, dans un système de RCO, si un objet `o` est une instance de la classe `C`, cela signifie que `o` a été défini comme tel et qu'il possède les propriétés de la classe `C` (*conditions nécessaires*). Réciproquement, si un objet `o` possède des propriétés qui pourraient en faire une instance de la classe `C`, il n'existe (généralement) pas de sémantique qui puisse assurer que `o` est une une instance de `C` (*conditions suffisantes*). Par suite, le processus de classification n'est pas directement et entièrement exploitable, comme il serait souhaitable (voir toutefois la façon dont le raisonnement par classification est exploité dans [Napoli et Laurenço, 1993], ainsi que dans les *chapitres 12 et 14*).

Nous terminerons en évoquant le *raisonnement par défaut*. Dans un système de RCO, tout attribut d'une classe C se voit généralement associer une valeur par défaut, qui peut être redéfinie dans toutes les sous-classes de C. Une sous-classe D hérite toutes les propriétés associées à ses super-classes, sauf *masquage* ou *exception* à l'héritage [Ducournau et Habib, 1989]. Dans le premier cas, la valeur d'une propriété héritable, valeur effective, méthode ou réflexe, est redéfinie. Dans le second cas, un ensemble de propriétés héritables est écarté. Le masquage et les exceptions ont été beaucoup étudiés dans les systèmes de RCO [Ducournau *et al.*, 1995], mais ces deux techniques posent de nombreux problèmes dans le cadre des logiques de descriptions, car elles conduisent à un raisonnement non monotone, qui va à l'encontre de la sémantique d'inclusion des extensions associée à la subsomption [Brachman, 1985] : il ne peut y avoir d'exception dans l'inclusion ensembliste $\mathtt{D}^{\mathcal{I}} \subseteq \mathtt{C}^{\mathcal{I}}$ lorsque $\mathtt{D} \sqsubseteq \mathtt{C}$. Toutefois, un certain nombre de travaux traitent d'une possible intégration du raisonnement par défaut dans les logiques de descriptions : [Quantz et Royer, 1992] [Baader et Hollunder, 1993] [Padgham et Zhang, 1993]. Parmi les études d'intégration, celle qui est présentée dans [Coupey et Fouqueré, 1997] se démarque quelque peu : des exceptions et des valeurs par défauts sont directement associées à la structure des concepts dans la logique de descriptions $\mathcal{AL}_{\delta\epsilon}$. Une autre originalité de cette étude est de fournir à $\mathcal{AL}_{\delta\epsilon}$ une sémantique algébrique qui s'appuie sur un système équationnel, à l'image des travaux réalisés autour de K-REP [Dionne *et al.*, 1993].

Il existe des comparaisons entre logiques de descriptions, modèles de données (dans les bases de données) et systèmes de RCO, qui ont été faites à l'initiative des chercheurs travaillant sur les logiques de descriptions [Calvanese *et al.*, 1994] [Borgida, 1995]. Toutefois, ces comparaisons ne tiennent pas compte de l'aspect procédural existant dans un système de RCO et peuvent donc apparaître comme incomplètes. À l'inverse, l'ensemble des études théoriques sur les logiques de descriptions a inspiré certains chercheurs travaillant sur les systèmes de RCO et les a amenés a mettre en place une théorie des systèmes classificatoires (ce qui est d'ailleurs l'objet du *chapitre 12*).

Soulignons encore qu'un des intérêts principaux des travaux réalisés sur les logiques de descriptions est de fournir aux concepteurs de systèmes à bases de connaissances de bonnes indications sur ce qu'ils peuvent — et ne peuvent pas — représenter et traiter comme connaissances avec une logique de descriptions, à l'image de [Brachman *et al.*, 1991], où sont examinées « les tâches pour lesquelles CLASSIC est et n'est pas approprié », et de [Baader, 1996], où le pouvoir d'expression des logiques de descriptions est analysé formellement.

#### Quelques limitations des logiques de descriptions

Nous avons vu au paragraphe 11.2.8 que les travaux sur la complexité du raisonnement terminologique ont mis en évidence que plus un langage de description de concepts et de rôles est riche, plus la complexité du raisonnement est élevée. Le dilemme est alors le suivant : faut-il préférer un langage limité et concis, à l'image de CLASSIC, qui ne permet pas de tout représenter, mais qui a un comportement déductif contrôlable, ou bien faut-il préférer un langage riche, à l'image de BACK ou de LOOM, aux possibilités multiples et variées, mais au comportement déductif imprévisible ? La dualité « langage pauvre » ou « langage riche » a fait l'objet de l'article [Doyle et Patil, 1991], où est critiquée la tendance minimaliste associée à CLASSIC et l'« enfermement conceptuel » auquel aurait donné naissance les résultats des travaux sur la complexité de la subsomption [10]. Dans [Doyle et Patil, 1991] sont également discutées les limitations des logiques de descriptions dans le cadre de la conception de systèmes d'intelligence artificielle réalistes. Ces limitations sont de plusieurs sortes, parmi lesquelles figurent :

- la difficulté de représenter des relations entre rôles, comme dans les exemples suivants : *deux personnes qui ont pour résidence des pays différents*, ou bien *deux enfants d'une même famille qui vont dans deux écoles différentes qui sont éloignées l'une de l'autre de 1 kilomètre*,
- la représentation de quantifications sur les relations comme dans *tous les élèves assistent à tous les cours* [Woods, 1991] ;
- les traitements numériques en général ;
- les définitions récursives ;
- les relations $n$-aires.

Il faut également mentionner le fait qu'une logique de descriptions ne peut pas être employée seule dans une application réaliste — comme tout formalisme de RCO d'ailleurs — et nécessite l'usage parallèle d'un langage de programmation, avec lequel sont écrites les fonctions de contrôle global de l'application.

À l'heure actuelle, les conséquences des travaux sur la complexité de la subsomption semblent avoir été assimilées et les réflexions se portent plutôt sur l'utilisation des logiques de descriptions dans des applications et domaines divers comme les bases de données et les systèmes d'information, la fouille de données, le traitement du langage naturel, etc.

**Perspectives : niveaux de connaissances et systèmes multi-niveaux**

Les apports des logiques de descriptions à la représentation de connaissances sont importants. L'un des principaux est d'avoir mis l'accent sur l'existence de différents niveaux de connaissances [Brachman, 1979] et de proposer des moyens effectifs pour les prendre en compte, comme le font les processus de classification et d'instanciation. À côté des niveaux généraux que sont les niveaux terminologiques et factuels, les règles et implications — comme celles de CLASSIC et de LOOM — constituent un troisième niveau de représentation. D'autres niveaux de connaissances peuvent venir enrichir ces niveaux de base [MacGregor, 1991] et traiter divers besoins de représentation, comme le temps [Schmiedel, 1990], les connaissances d'ordre numérique (*concrete domains*) [Baader et Hanschke, 1991], la relation de composition et les objets composites [Padgham et Lambrix, 1994] et enfin les règles du premier ordre [Levy et Rousset, 1996]. Dans le même genre d'idées, il est proposé dans [Buchheit *et al.*, 1994] de distinguer deux niveaux dans la description des

10. Le comportement déductif d'un sous-système de CLASSIC est examiné minutieusement dans [Borgida et Patel-Schneider, 1994], tandis qu'une étude comparative concernant les langages « pauvres » et « riches » est donnée dans [Heinsohn *et al.*, 1994].

concepts et des rôles : le niveau des *schémas*, qui se rapporte aux concepts et aux rôles primitifs, et le niveau des *vues*, qui se rapportent aux concepts définis (schéma et vue sont deux termes qui ont été choisis par analogie avec la terminologie employée dans les bases de données). Il est alors possible de travailler avec des circuits terminologiques au niveau des schémas et d'introduire des constructions soit au niveau des schémas soit au niveau des vues, ce qui conduit à une appréhension différente de la complexité du raisonnement terminologique.

Toutefois, différencier ces niveaux de connaissances est quelquefois difficile, certaines connaissances se décrivant aussi bien par des règles, des implications ou encore des expressions conceptuelles. Ce problème met en lumière l'absence de véritable méthodologie de représentation et de conception d'applications dans le cadre des logiques de descriptions [Brachman *et al.*, 1991]. C'est également vrai pour les systèmes de RCO [Chouvet *et al.*, 1996], en dehors de tous les travaux sur l'acquisition de connaissances.

En s'appuyant sur des travaux qui s'intéressent à l'intégration de plusieurs formalismes de représentation dans un même système [Carré *et al.*, 1995], la mise au point de systèmes de représentation *multi-niveaux*, où chaque niveau est relatif à un langage de description de concepts et de rôles qui lui est propre, constitue une des perspectives d'utilisation des logiques de descriptions [MacGregor, 1991]. Un premier niveau peut correspondre à un langage de description de concepts et de rôles simple, comme $\mathcal{FL}$-, $\mathcal{PL}_1$ ou $\mathcal{PL}_2$ [Donini *et al.*, 1991b], pour lequel la subsomption est correcte, complète et de complexité polynômiale. Les descriptions de premier niveau ne sont en relation qu'avec des descriptions de ce niveau. Au second niveau apparaît un langage de description de concepts et de rôles de l'ordre de $\mathcal{ALCNR}$, ce qui remet (forcément) en cause la complétude de la subsomption. Un troisième niveau peut être réservé à la représentation de connaissances procédurales, un autre aux méta-connaissances, et d'autres niveaux encore peuvent prendre en compte le temps, la composition, etc. — comme évoqué ci-dessus — sans oublier un niveau dédié à la programmation. Les niveaux peuvent être classés par ordre (partiel) de complexité croissant, et les descriptions de niveau $n$ ne peuvent être en relation qu'avec des descriptions de niveau inférieur ou égal à $n$. L'utilisateur quant à lui peut choisir son niveau de représentation en toute connaissance de cause et tenir compte du comportement déductif du système au niveau où il travaille.

## 11.5 Conclusion

Dans ce chapitre, nous avons présenté les logiques de descriptions, qui constituent un formalisme de représentation de connaissances caractérisé par les points suivants :

- Un langage permet de construire des descriptions conceptuelles qui sont génériques (concepts primitifs et définis) ou individuelles (instances).
- Une sémantique est associée à chaque construction syntaxique par l'intermédiaire d'une interprétation.

- Une relation de subsomption permet d'organiser les descriptions par niveau de généralité, et de procéder à des inférences ; cette relation est à la base des processus de classification et d'instanciation.

Sur le plan de la théorie de la représentation des connaissances, les logiques de descriptions ont apporté une certaine originalité dans le traitement de problèmes d'intelligence artificielle. Elles se sont avérées être un des premiers formalismes structurels — où sont manipulées des structures — à différencier et étudier explicitement le niveau syntaxique et le niveau sémantique dans une représentation, ainsi que les relations existant entre ces deux niveaux, comme c'est le cas en logique classique.

Les logiques de descriptions ne sont pas seulement un formalisme théorique et réservé aux théoriciens de la représentation des connaissances : la recherche autour des logiques de descriptions est très active et a des visées à la fois pratiques et théoriques. Ainsi, la construction de systèmes traitant de problèmes réels est au centre des préoccupations de nombreux travaux de recherches. Les logiques de descriptions ne sont pas des formalismes figés et sont suffisamment souples pour accepter l'introduction de nouveaux constructeurs, capable de répondre à des besoins particuliers, charge alors au concepteur d'évaluer et de tenir compte de la complexité de la subsomption qui en résulte.

Les études effectuées dans le cadre des logiques de descriptions constituent une base de première importance pour les recherches sur les systèmes de RCO, que ce soit pour les concepteurs ou les utilisateurs de ces systèmes. En particulier, le raisonnement par classification est devenu une méthode d'inférence essentielle, mais qui nécessite d'être intégrée et utilisée de façon correcte. Dans tous les cas, la recherche sur les systèmes de RCO peut se nourrir des recherches sur les logiques de descriptions, mais la réciproque est vraie aussi, et l'avenir des logiques de descriptions passe certainement par une intégration avec les systèmes de RCO.

**Note sur les logiques de descriptions et le Web**

Les lecteurs navigateurs pourront aller visiter l'adresse suivante sur le Web :

`http://dl.kr.org/dl`

où ils trouveront des informations concernant notamment la bibliographie et les séminaires annuels sur les logiques de descriptions, ainsi que les moyens de joindre les chercheurs du domaine et d'obtenir les systèmes disponibles.

# 12

# La logique des objets : application à la classification incertaine

LE SUCCÈS DES OBJETS est justifié habituellement par les qualités que recherche et promeut le génie logiciel : modularité, extensibilité ou réutilisabilité sont des références obligées (voir *chapitre 2* par exemple). Pourtant, une raison moins reconnue est sans doute aussi importante : les objets (informatiques) ont une capacité « naturelle » de représentation (des objets) du « monde » (*chapitre 1*). Cette capacité est à l'origine de la représentation des connaissances par objets (*chapitre 10*) en intelligence artificielle, autant que des méthodes d'analyse et de conception (*chapitre 4*) en génie logiciel ou encore des bases de données d'objets (*chapitre 5*). En fait, on peut conjecturer que ces bonnes qualités de représentation, par leur « déclarativité » [Winograd, 1975 ; Carré *et al.*, 1995], sont aussi à l'origine des qualités revendiquées par le génie logiciel. L'étude de la sémantique de représentation des diverses approches objet et leur comparaison dans un cadre commun est ainsi une étape nécessaire. Or, parallèlement aux développements de l'approche objet dans le courant principal du génie logiciel, un courant important de la représentation des connaissances et plus particulièrement des *réseaux sémantiques* a conduit à la définition d'une famille de langages, les *logiques de descriptions* (*chapitre 11*), qui s'est trouvée confrontée, dès sa naissance, à ces questions de sémantique et les a résolues d'une manière tout à fait satisfaisante. Il était donc tentant d'appliquer cette recette éprouvée aux systèmes d'objets plus usuels. Le cadre des *systèmes classificatoires* est une abstraction de ces diverses approches objet qui constitue une première proposition en ce sens.

Un système classificatoire est constitué, d'une part d'une *hiérarchie* de *classes*, décrites ou définies en *intension* par un ensemble de *propriétés* et munies d'une interprétation *extensionnelle*, d'autre part d'un ensemble d'assertions formant une description, incomplète, du monde [Ducournau, 1996a]. L'ensemble peut être considéré comme une logique, avec une sémantique directe en théorie des modèles, comme celles des logiques de descriptions [Nebel, 1990a ; Woods et Schmolze, 1992] (*chapitre 11*), et un système de déduction constitué d'un ensemble de règles d'inférences ou d'algorithmes spécialisés. La justification de « classificatoire » est double : ces systèmes offrent une « classification des espèces », au sens de la hiérarchie de classes, et ils se prêtent à une « classification des individus », au sens du mécanisme d'inférence qui reconnaît qu'un individu appartient à une classe. Ce chapitre est consacré à ce mécanisme d'inférence et à la sémantique qu'il suppose.

Cette approche classificatoire a été reconnue et mise en œuvre à la charnière des années 70 et 80 dans les langages dits de *frames* [Minsky, 1975 ; Fikes et Kehler, 1985] qui ont donné naissance à deux courants principaux. L'un des critères discriminant ces deux courants est le caractère *définitionnel* ou simplement *descriptif* des classes. Les logiques de descriptions (LD) ont introduit la distinction entre classe[1] *définie* ou *primitive* [Nebel, 1990a] : pour une classe définie, la description de la classe constitue une *condition nécessaire et suffisante* d'appartenance d'un objet à l'extension de la classe. En revanche, pour une classe primitive, la description n'est qu'un ensemble de conditions nécessaires. Des classes définies autorisent la classification automatique des instances que pratiquent les logiques de descriptions sous le nom de *réalisation* ou, plus récemment, d'*instanciation*. Du côté des systèmes de représentation des connaissances par objets, la sémantique des classes est purement descriptive : aussi, pour remédier à l'impossibilité d'une classification automatique, certains de ces systèmes pratiquent, depuis SHIRKA [Rechenmann, 1988] et ses successeurs TROEPS [Mariño, 1993 ; Projet Sherpa, 1995 ; Euzenat et Rechenmann, 1995] ou FROME [Dekker, 1994], une classification que l'on peut qualifier d'*incertaine* puisqu'elle attribue à la relation classe-instance une modalité parmi *sûre*, *possible* et *impossible*.

Chacune de ces approches présente une limitation importante : pour les logiques de descriptions, les classes primitives constituent une barrière quasiment infranchissable, dans la mesure où l'appartenance à une classe primitive ne peut en général pas se déduire, alors que, de son côté, la représentation des connaissances par objets ne peut effectuer presqu'aucune inférence valide de classification automatique.

L'objectif de ce chapitre est double. Il s'agit d'une part de présenter un cadre abstrait, analogue à celui des logiques de description, mais permettant d'appréhender les spécificités des systèmes d'objets usuels, qu'ils viennent de la programmation, des bases de données ou de la représentation des connaissances. Il s'agit d'autre part d'utiliser ce cadre abstrait pour étudier la validité de cette classification incertaine pratiquée par certains systèmes de représentation des connaissances à objets. Le chapitre renverra ainsi en permanence aux *chapitres 10 et 11*.

Les trois premières parties du chapitre introduisent le cadre général des systèmes classificatoires (SC) puis présentent successivement leur sémantique formelle et une abstraction du système de déduction associé. La quatrième partie pose le problème de la classification d'instances et donne trois versions, extensionnelle, intensionnelle et structurelle, de la classification incertaine, avant d'examiner les questions de correction et de complétude qui ne manquent pas de se poser. Le chapitre se termine par une comparaison avec des travaux connexes et ouvre quelques perspectives.

1. Les logiques de descriptions utilisent en fait le terme de *concept* mais nous utiliserons uniformément celui de *classe*.

## 12.1 Les systèmes classificatoires

Le problème de la classification se pose dans un cadre de représentation par objets, au sens large, qui consiste en une *hiérarchie* de *classes* décrites en *intension* et interprétées en *extension*. Cette hiérarchie constitue une première composante, *intensionnelle*, ce que les logiques de descriptions nomment la *TBox* ; il lui correspond une seconde composante, *extensionnelle*, la *ABox* dans la définition restreinte de [Nebel, 1990a] ou de [Buchheit *et al.*, 1993]. L'ensemble constitue un *système classificatoire*.

### 12.1.1 La composante intensionnelle : les classes

La composante intensionnelle des SC est constituée d'une hiérarchie de classes décrites en intension par des *propriétés* qui s'expriment au travers d'*attributs* ; s'y ajoutent diverses relations entre ces classes.

#### La hiérarchie et son interprétation extensionnelle

$\mathcal{X}$ est un ensemble fini d'*entités génériques* ou *classes*, ordonnées par une relation de *spécialisation* $\prec$ qui est un ordre partiel strict (et $\preceq$ est l'ordre partiel associé). Comme c'est l'habitude dans les approches objet, on parlera de *sous-classe* ou de *super-classe* en référence à cette relation.

Cette hiérarchie s'interprète dans un cadre extensionnel dont nous présentons d'abord un aperçu intuitif et relativement informel : $\Omega$ est un ensemble d'*entités individuelles* ou *objets*, qui peuvent être associés aux classes par l'application $Ext : \mathcal{X} \longrightarrow 2^{\Omega}$ telle que

$$\alpha \preceq \beta \Longrightarrow Ext(\alpha) \subseteq Ext(\beta).$$

$Ext(\alpha)$ est appelé l'*extension* de la classe $\alpha$ : c'est l'ensemble des objets qui « tombent sous le concept » $\alpha$. Lorsque $o \in Ext(\alpha)$, $o$ est une *instance de* $\alpha$, et $\alpha$ est *une* (et non pas *la*) *classe de o*. Inversement, l'application $Isa : \Omega \longrightarrow 2^{\mathcal{X}}$ associe à tout objet *o* les classes dont il est instance, *ses* classes. La relation de *subsomption extensionnelle* est un *préordre* sur $\mathcal{X}$, noté $\trianglelefteq$, induit par l'inclusion des extensions :

$$\alpha \trianglelefteq \beta \stackrel{\text{def}}{\Longleftrightarrow} Ext(\alpha) \subseteq Ext(\beta).$$

De plus, deux éléments de $\mathcal{X}$ sont distingués : un plus grand élément, noté $\top$, d'extension $\Omega$, et un plus petit, noté $\bot$, d'extension vide.

Il faut enfin ajouter à ce cadre général une relation d'incompatibilité, notée $\alpha\nabla\beta$, qui peut se définir extensionnellement par $Ext(\alpha) \cap Ext(\beta) = Ext(\bot)$. La relation d'incompatibilité est héritable : si deux classes sont incompatibles, les sous-classes de l'une sont incompatibles avec l'autre.

Ce cadre extensionnel naïf trouvera dans la suite deux réalisations différentes : la première dans la composante extensionnelle qui décrit les instances *effectives*, la seconde dans la sémantique formelle du SC qui considère les instances *potentielles*.

**La description intensionnelle des classes**

Chaque classe est munie d'une « description » *en intension*, formée d'une collection de *propriétés* dont chacune s'exprime par l'appartenance d'un *attribut* à un *domaine*. $\mathcal{P}$ est un ensemble d'attributs et, pour toute classe $\alpha$, $\mathcal{P}_\alpha$ représente l'ensemble des attributs de $\alpha$. L'*héritage de propriétés* consiste d'abord en une inclusion des ensembles d'attributs associés à des classes qui se spécialisent : $\alpha \prec \beta \Longrightarrow \mathcal{P}_\alpha \supseteq \mathcal{P}_\beta$.

Une application $Dom : \mathcal{X} \times \mathcal{P} \longrightarrow 2^{\mathcal{D}}$ (où $\mathcal{D}$ est un domaine quelconque contenant $\Omega$ ou $2^\Omega$) associe à chaque attribut $P$ d'une classe $\alpha$ un *domaine* $Dom(\alpha, P)$. On notera $Dom_P$ (resp. $Dom_\alpha$) la fonction obtenue en fixant $P$ (resp. $\alpha$).

Le cadre intensionnel est relié au cadre extensionnel comme suit. D'abord, chaque objet $o$ possède l'ensemble des attributs décrits dans ses classes :

$$Attr(o) \stackrel{\text{def}}{=} \bigcup_{\alpha \in Isa(o)} \mathcal{P}_\alpha.$$

Dans un objet $o$, chaque attribut $P \in Attr(o)$ désigne une *valeur*, notée $P(o)$, dont $Dom(\alpha, P)$ représente les valeurs admissibles. Si $o \in Ext(\alpha)$ et $P \in \mathcal{P}_\alpha$, alors $P(o) \in Dom(\alpha, P)$. L'*héritage de propriétés* n'est pas seulement l'héritage des attributs : c'est aussi l'héritage de leur domaine et une contrainte d'inclusion en cas de *redéfinition* de ces domaines :

$$\alpha \prec \beta \ \ \& \ \ P \in \mathcal{P}_\beta \Longrightarrow Dom_P(\alpha) \subseteq Dom_P(\beta). \tag{12.1}$$

À ce stade, on peut introduire la notion d'*intension* d'une classe, notée $Int(\alpha)$ et constituée par le couple $(\mathcal{P}_\alpha, Dom_\alpha)$. Les intensions sont munies d'une relation d'ordre $\sqsupseteq$, similaire à l'inclusion, comme suit :

$$Int(\alpha) \sqsupseteq Int(\beta) \stackrel{\text{def}}{\Longleftrightarrow} \mathcal{P}_\alpha \supseteq \mathcal{P}_\beta \ \ \& \ \ \forall P \in \mathcal{P}_\beta : Dom_P(\alpha) \subseteq Dom_P(\beta)$$

$Int$ est donc, comme $Dom_P$, monotone :

$$\alpha \prec \beta \Longrightarrow Int(\alpha) \sqsupseteq Int(\beta).$$

De façon analogue à la subsomption extensionnelle, la relation de *subsomption intensionnelle* est un *préordre* sur $\mathcal{X}$, noté $\trianglerighteq$, induit par l'inclusion des intensions :

$$\alpha \trianglerighteq \beta \stackrel{\text{def}}{\Longleftrightarrow} Int(\alpha) \sqsupseteq Int(\beta).$$

Les opérations ensemblistes usuelles s'étendent aux intensions en introduisant les opérateurs $\sqcap$ et $\sqcup$ :

$$Int(\alpha) \sqcup Int(\beta) \stackrel{\text{def}}{=} (\mathcal{P}_\alpha \cup \mathcal{P}_\beta, Dom_{\alpha\cap\beta}), \text{ avec}$$

$$Dom_{\alpha\cap\beta}(P) = \begin{cases} Dom_\alpha(P) \cap Dom_\beta(P) & \text{si } P \in \mathcal{P}_\alpha \cap \mathcal{P}_\beta \\ Dom_\alpha(P) & \text{si } P \in \mathcal{P}_\alpha - \mathcal{P}_\beta \\ Dom_\beta(P) & \text{si } P \in \mathcal{P}_\beta - \mathcal{P}_\alpha \end{cases}$$

Dans cette définition, si jamais l'un des $Dom_\alpha(P) \cap Dom_\beta(P)$ est vide (si $P$ n'a aucune valeur admissible), on pose alors que $Int(\alpha) \sqcup Int(\beta) = (\mathcal{P}, \emptyset) = Int(\perp)$. $\sqcap$ se définit de manière duale.

*Covariance et contravariance*

La monotonie de $Dom_P$ (12.1) traduit une *covariance* qui est naturelle en représentation des connaissances. Cette covariance est pratiquée aussi, entre autres, par EIFFEL (*chapitres 2 et 3*) et $O_2$ (*chapitre 5*), bien qu'elle soit l'exact opposé de la *contravariance* que la théorie des types impose pour garantir une compilation « sûre du point de vue des types » (*type safe*) [Castagna, 1996 ; Boyland et Castagna, 1996 ; Castagna, 1997] (voir *chapitre 1*).

Cette incompatibilité de la représentation des connaissances et de la théorie des types s'explique simplement : le domaine d'un attribut est implicitement quantifié existentiellement (pour chaque objet, *il existe une* valeur qui value l'attribut), alors que la substituabilité à la base de la théorie des types sous-entend une quantification universelle (l'expression peut être substituée par *toute* valeur de son type). L'incompatibilité n'est gênante qu'à partir du moment où l'on veut écrire des programme statiquement typés qui manipulent des objets de représentation : ce n'est théoriquement pas l'objet de la représentation des connaissances par objets (encore que ce soit un souhait exprimé par [Carré *et al.*, 1995]), mais c'est l'objectif explicite des bases de données objet (voir *chapitre 5*) et, même celui des langages de programmation par objets, dès lors que leurs objets se mettent partiellement à représenter le monde. D'où le choix non orthodoxe de $O_2$ ou d'EIFFEL.

**L'expression de la composante intensionnelle**

En pratique, une classe $\alpha$ est introduite au moyen de ses super-classes directes $Super(\alpha)$ — explicitement désignées ou, par défaut, $\{\top\}$ — et de son *intension propre*, notée $Int^*(\alpha)$, par laquelle sont ajoutés des attributs ou restreints des domaines. $\mathcal{P}^*_\alpha$ désignera le sous-ensemble d'attributs de $\alpha$ figurant dans $Int^*(\alpha)$. Alors, $Int(\alpha) = (\bigsqcup_{\beta \in Super(\alpha)} Int(\beta)) \sqcup Int^*(\alpha)$. On suppose que le domaine d'un attribut *hérité* sans redéfinition s'obtient par intersection des domaines hérités. Il s'agit donc soit d'un simple héritage du domaine, soit de ce que [Ducournau *et al.*, 1995] appelle une résolution correcte de conflit de valeur, sur le domaine de l'attribut.

Pour que l'expression des classes soit complète, il faut aussi exprimer le domaine des attributs. Le problème est que $Dom$ n'a été considéré jusqu'ici qu'en extension, alors qu'il ne peut être exprimé qu'en intension. Or les classes constituent le seul moyen à notre disposition pour définir un ensemble. En première approximation, $Dom(\alpha, P)$ peut donc être donné par le type de l'attribut, et par ses cardinalités admissibles, grâce à deux fonctions, $type : \mathcal{X} \times \mathcal{P} \longrightarrow \mathcal{X}$ et $card : \mathcal{X} \times \mathcal{P} \longrightarrow 2^{\mathbf{N}}$. L'ensemble des valeurs admissibles est alors défini par :

$$Dom(\alpha, P) \stackrel{\text{def}}{=} \{A \subseteq Ext(type(\alpha, P)) \mid |A| \in card(\alpha, P)\}.$$

Outre l'introduction intensionnelle des classes, par l'explicitation de leurs attributs et de la relation de spécialisation, il peut être nécessaire d'expliciter les relations extensionnelles qui ne se déduisent pas de l'intension. Ainsi, si la relation d'incompatibilité possède un analogue intensionnel, défini par $Int(\alpha) \sqcup Int(\beta) = Int(\bot)$, les deux définitions ne sont pas équivalentes : il peut donc être nécessaire de déclarer explicitement certaines relations d'incompatibilité, en plus des déclarations intensionnelles implicites. On notera ainsi $\nabla_a$ la relation d'incompatibilité exprimée explicitement, pour affirmer par exemple que, dans le domaine modélisé, il n'existe pas d'objets de `[forme ronde]` et de `[couleur rouge]`.

**Classes définies ou primitives**

Chaque classe ainsi introduite doit aussi être qualifiée de *primitive* ou *définie*. Dans ce dernier cas, la description (ou l'intension) de la classe est une condition nécessaire et suffisante d'appartenance pour les instances, alors que, dans le premier cas, il ne s'agit que de conditions nécessaires.

Dans les systèmes d'objets habituels (langages de programmation ou de représentation, bases de données) les classes ne sont en général que primitives. Ce sont les logiques de descriptions (*chapitre 11*) qui ont mis au premier plan cette distinction, mais elle est suffisamment naturelle pour qu'un langage de programmation comme LORE [Caseau, 1985 ; Caseau, 1987] éprouve le besoin d'introduire un équivalent des classes définies sous le nom de *définition par sélection*. Les classes définies constituent aussi l'une des premières formes des *vues* dans les bases de données objet (cf. *chapitre 5*).

### 12.1.2 La composante extensionnelle : les instances

La composante extensionnelle est constituée par la description de toutes les entités individuelles, instances *effectives* des classes introduites dans la composante intensionnelle. On supposera donnée, dans la suite, cette dernière.

**Description du monde**

La composante extensionnelle constitue une *description du monde* notée $\mathcal{W}$. C'est une réalisation particulière de l'interprétation extensionnelle du SC qui doit donc respecter toutes les contraintes formulées dans l'exposé de cette interprétation. La description du monde peut être vue comme un ensemble fini d'assertions, analogue à une base de données relationnelle. Ces assertions peuvent être de deux sortes :

- $\alpha(o) \in \mathcal{X} \times \Omega$ affirme l'existence de l'*objet* $o \in Ext(\alpha)$ ;
- $P(o, o') \in \mathcal{P} \times \Omega \times \Omega$ associe à l'attribut $P$ de l'objet $o$ la valeur $o'$.

La seconde catégorie pose le problème des valeurs manquantes : la valeur de l'attribut $P$ de l'objet $o$, notée $P(o)$, est constituée par éventuellement plusieurs assertions élémentaires : $P(o, v_i), 1 \leq i \leq n$, et $v = \{v_i \mid 1 \leq i \leq n\}$. L'ensemble de ces assertions s'interprétera comme une inclusion, $P(o) \supseteq v$ : c'est une hypothèse du *monde ouvert* sur la valeur des attributs.

Les instances s'expriment tout aussi bien dans un style objet — c'est-à-dire en faisant apparaître des entités avec des attributs —, par un ensemble $\mathcal{O}$ d'instances $o$, appartenant à des classes $Isa_{\mathcal{W}}(o)$ et munies d'un ensemble $Attr_{\mathcal{W}}(o)$ d'attributs $P$ de valeur $P_{\mathcal{W}}(o)$ :

(12.2) $$\mathcal{O} \stackrel{\text{def}}{=} \{o \mid \alpha(o) \in \mathcal{W}\}$$

(12.3) $$Isa_{\mathcal{W}}(o) \stackrel{\text{def}}{=} \{\beta \in \mathcal{X} \mid \alpha(o) \in \mathcal{W} \;\;\&\;\; \alpha \preceq \beta\}$$

(12.4) $$Attr_{\mathcal{W}}(o) \stackrel{\text{def}}{=} \{P \mid P(o, o') \in \mathcal{W}\}$$

(12.5) $$P_{\mathcal{W}}(o) \stackrel{\text{def}}{=} \{o' \mid P(o, o') \in \mathcal{W}\}$$

Ces fonctions caractérisent ce qui a été explicitement affirmé de l'objet, sans aucune inférence, à part la fermeture par $\preceq$ pour $Isa_{\mathcal{W}}$. La fonction $Ext_{\mathcal{W}}$, qui désigne les instances connues d'une classe, se définit symétriquement.

**Les objectifs d'un système classificatoire**

La construction de la hiérarchie elle-même mise à part, un système classificatoire peut être utilisé, au niveau extensionnel, de diverses façons :

**l'héritage** permet de connaître *déductivement* les attributs d'un objet — pas leurs valeurs, au moins dans un premier temps (*chapitre 10*), mais leur présence et leur « intension » —, étant donné les classes de l'objet ;

**la classification** permet de reconnaître *inductivement* un objet comme instance d'une classe, compte-tenu de ses attributs connus : c'est une *identification* au sens du *chapitre 15* ; des assertions $\{\alpha(o)\ P(o, o')\}$, on peut déduire que $o' \in Ext(type(\alpha, P))$, mais on peut aussi induire une classe $\beta$, éventuellement sous-classe de $\alpha$, telle que $o' \in Ext(type(\beta, P))$ et $o \in Ext(\beta)$.

**le filtrage** détermine les instances d'une classe ou, plus généralement, les objets qui vérifient certaines propriétés qui peuvent s'exprimer par l'intension d'une classe : dans les logiques de descriptions, comme dans FROME [Dekker, 1993], toute requête est considérée comme une classe dont il ne reste plus qu'à collecter les instances, une fois qu'elle a été correctement insérée dans la hiérarchie.

**la construction** d'instance consiste à valuer les attributs d'un objet, éventuellement en créant d'autres objets qu'il faut récursivement construire et classer [Girard, 1995].

Mais la distinction de ces différents problèmes est artificielle : il s'agit d'abord d'un problème uniforme de déduction. Ce n'est qu'ensuite que l'on peut introduire des spécificités comme une classification ou un filtrage incertains.

## 12.2 Sémantique des systèmes classificatoires

La *théorie des modèles* offre un cadre général pour établir une sémantique formelle ensembliste : le but est en particulier de définir la notion de conséquence *valide*, c'est-à-dire valable pour tous les *modèles*, d'un ensemble de descriptions.

Certains auteurs partent du principe que l'objectif de la représentation des connaissances est de représenter une partie idéalisée du monde réel, et considèrent qu'il existe un *domaine modélisé*, c'est-à-dire un modèle privilégié, qui est censé détenir la sémantique du SC. C'est, par exemple, la position de [Euzenat, 1994]. Les propositions qui sont vraies dans ce modèle particulier sans être valides traduisent alors une incapacité du système de représentation (ou de ses utilisateurs) à représenter fidèlement le domaine modélisé et lui seul. Mais, un SC est un système générique qui ne peut pas s'appuyer sur l'existence d'un domaine modélisé pour la réalisation de ses algorithmes : pour être corrects, ils doivent correspondre à des inférences valides. Aussi la prise en compte du domaine modélisé en tant que modèle privilégié ne paraît possible que dans le choix d'un SC muni de propriétés spécifiques. À cette *incomplétude intensionnelle* s'ajoute une *incomplétude extensionnelle* liée au fait que l'état du domaine modélisé, c'est-à-dire l'*état du monde*, n'est pas complètement connu.

### 12.2.1 Une sémantique en théorie des modèles

Le cadre général et les notations sont empruntés, avec quelques variantes, aux logiques de descriptions, en particulier à [Nebel, 1990a]. Précisons qu'il ne s'agit que d'une sémantique partielle, d'un schéma général où ne figurent que les équations communes à tous.

**Modèle d'un système classificatoire**

Une *structure sémantique*[2] d'un SC est un quadruplet $(\Delta, \mathcal{E}_\mathcal{X}, \mathcal{E}_\mathcal{P}, \mathcal{E}_\mathcal{O})$ formé d'un domaine arbitraire $\Delta$, d'une fonction d'interprétation des classes $\mathcal{E}_\mathcal{X} : \mathcal{X} \longrightarrow 2^\Delta$, d'une fonction d'interprétation des attributs $\mathcal{E}_\mathcal{P} : \mathcal{P} \longrightarrow (\Delta \rightarrow 2^\Delta)$, et d'une fonction d'interprétation des objets, injective, $\mathcal{E}_\mathcal{O} : \mathcal{O} \longrightarrow \Delta$. Pour simplifier les notations, on considérera une fonction unique, notée $\mathcal{I}$ (en exposant), dont chacune des trois fonctions d'interprétation sera la restriction sur son domaine propre. Une structure sémantique se réduit alors à la paire $(\Delta, \mathcal{I})$.

Un *modèle* $\mathcal{M}$ d'un système classificatoire est une structure sémantique $(\Delta, \mathcal{I})$ qui satisfait les propriétés suivantes, d'une part pour la composante intensionnelle :

$$\bot^\mathcal{I} = \emptyset \qquad \top^\mathcal{I} = \Delta \tag{12.6}$$

$$\alpha \prec \beta \Longrightarrow \alpha^\mathcal{I} \subseteq \beta^\mathcal{I} \tag{12.7}$$

$$\alpha \nabla_a \beta \Longrightarrow \alpha^\mathcal{I} \cap \beta^\mathcal{I} = \emptyset \tag{12.8}$$

$$o \in \alpha^\mathcal{I}, P \in \mathcal{P}_\alpha \Longrightarrow \begin{cases} P^\mathcal{I}(o) \subseteq type(\alpha, P)^\mathcal{I} \\ |P^\mathcal{I}(o)| \in card(\alpha, P) \end{cases} \tag{12.9}$$

$$o \notin \bigcup_{P \in \mathcal{P}_\alpha} \alpha^\mathcal{I} \Longrightarrow P^\mathcal{I}(o) \text{ pas défini} \tag{12.10}$$

2. Les termes de *structure sémantique*, *modèle* et *interprétation* sont souvent synonymes : nous avons pris le parti de désigner par *structure sémantique* un « type », constitué d'un domaine et des quelques fonctions associées à ce domaine, par *modèle* une structure sémantique vérifiant des propriétés particulières et par *interprétation* les fonctions (ou leurs images) qui associent aux entités du langage des éléments du domaine. Le *chapitre 11* fait un usage un peu différent des mêmes termes.

d'autre part, pour la composante extensionnelle :

(12.11) $$\alpha(o) \in \mathcal{W} \Longrightarrow o^{\mathcal{I}} \in \alpha^{\mathcal{I}}$$

(12.12) $$P(o_1, o_2) \in \mathcal{W} \Longrightarrow {o_2}^{\mathcal{I}} \in P^{\mathcal{I}}({o_1}^{\mathcal{I}})$$

Pour une classe définie, on aura une équation supplémentaire du style[3] de :

(12.13) $$\alpha^{\mathcal{I}} = \bigcap \begin{cases} \bigcap_{\beta \in Super(\alpha)} \beta^{\mathcal{I}} \\ \bigcap_{P \in \mathcal{P}^*_\alpha} \{o \mid P^{\mathcal{I}}(o) \subseteq type(\alpha, P)^{\mathcal{I}} \ \& \ |P^{\mathcal{I}}(o)| \in card(\alpha, P)\} \end{cases}$$

Seule l'inclusion, qui se déduit des formules (12.7-12.12), vaut pour une classe primitive.

L'axiome (12.10) dit que les attributs ne sont définis que dans les classes où ils ont été introduits : c'est une caractéristique des systèmes d'objets usuels qui n'est vérifiée ni par les logiques de description (*chapitre 11*), ni par TROEPS ou certains langages « simples » [Crampé et Euzenat, 1996] (*chapitre 10*). C'est pour cet axiome que le co-domaine de $\mathcal{E}_{\mathcal{P}}$ est un ensemble de fonctions et non un ensemble de relations.

### Subsomption, incompatibilité, incohérence et inconsistance

On peut maintenant définir formellement diverses notions extensionnelles dont la présentation n'a été qu'intuitive :

- les relations de subsomption et d'incompatibilité extensionnelle $\trianglelefteq$ et $\nabla$ se définissent en symétrisant les formules (12.7) et (12.8) et en les quantifiant universellement sur tous les modèles :

  (12.14) $$\alpha \trianglelefteq \beta \overset{\text{def}}{\Longleftrightarrow} \forall \mathcal{M} : \alpha^{\mathcal{I}} \subseteq \beta^{\mathcal{I}}$$

  (12.15) $$\alpha \nabla \beta \overset{\text{def}}{\Longleftrightarrow} \forall \mathcal{M} : \alpha^{\mathcal{I}} \cap \beta^{\mathcal{I}} = \emptyset$$

- une classe est *incohérente* si son extension est vide dans tous les modèles[4] : c'est donc une classe subsumée extensionnellement par $\perp$ ;
- une description du monde est *inconsistante* si elle n'a aucun modèle.

Les notions d'incohérence et d'inconsistance, qui sont relativement synonymes dans le langage commun, sont donc clairement distinguées : leur portée est respectivement locale ou globale et elles sont reliées par le fait que toute instance d'une classe incohérente rend la description du monde inconsistante[5]. Dans la suite du chapitre, nous utiliserons toujours ces termes dans ce sens formel.

3. Il y a plusieurs manières de définir une classe : celle-ci est relative aux super-classes et à l'intension propre, mais il est possible de les définir relativement à la seule intension de la classe.

4. Dans la terminologie du *chapitre 11*, ce serait une classe *insatisfiable* : comme nous utilisons plus loin ce dernier terme dans un autre sens, nous reprenons ici le terme d'incohérence utilisé aussi par [Nebel, 1990a].

5. La notion de (in)consistance utilisée dans les CSP (voir *chapitre 9*) est voisine de la nôtre : l'inconsistance réside dans une absence de modèle (logique) ou une absence de solution (CSP). En revanche, il n'y a pas, dans les CSP de notion distincte d'incohérence, et les deux termes sont employés sans distinction. Il est pourtant possible d'introduire la notion de *contrainte incohérente*, car applicable à aucun objet.

**La notion de conséquence**

On arrive alors classiquement à la notion de conséquence valide d'une description du monde :

- le modèle $\mathcal{M}$ satisfait la description $d$, noté $\models_{\mathcal{M}} d$, qui correspond aux versions symétriques des clauses (12.11) et (12.12) ci-dessus ;

(12.16) $$\models_{\mathcal{M}} \alpha(o) \iff o^{\mathcal{I}} \in \alpha^{\mathcal{I}}$$

(12.17) $$\models_{\mathcal{M}} P(o_1, o_2) \iff {o_2}^{\mathcal{I}} \in P^{\mathcal{I}}({o_1}^{\mathcal{I}})$$

- si tous les modèles de $\mathcal{W}$ satisfont $d$, noté $\mathcal{W} \models d$, $d$ est une *conséquence valide* de $\mathcal{W}$ : $\mathcal{W} \models d \iff \forall \mathcal{M} : \quad \models_{\mathcal{M}} d$. Si $\mathcal{W}$ est *inconsistant*, toute description en est une conséquence valide, comme en logique classique.

Une description du monde $\mathcal{W}$ est *fermée* par $\models$ ssi $\mathcal{W} = \{d \mid \mathcal{W} \models d\}$.

La sémantique qui précède a vidé de toute signification formelle les notations intuitives comme $Isa(o)$, $Ext(\alpha)$, $Attr(o)$ ou $P(o)$. Il faut en fait choisir, entre trois notions différentes, suivant que l'on considère la description du monde $\mathcal{W}$, un modèle $\mathcal{M}$ donné ou enfin tous les modèles de $\mathcal{W}$.

### 12.2.2 Propriétés structurelles spécifiques

Un catalogue de propriétés structurelles des systèmes classificatoires est proposé dans [Euzenat, 1993b]. Ces propriétés sont examinées plus en détail dans [Ducournau, 1996a] et nous les reprenons ici pour notre argumentation future. L'*exhaustivité* d'un SC se caractérise par le fait que tout objet est instance d'un (de) *puits* (c'est-à-dire un sommet sans successeur, $\perp$ mis à part) de la relation $\preceq$. C'est le cas des *espèces biologiques* : il n'existe pas de pur `mammifère`, qui ne soit pas membre d'une sous-catégorie, `chien` ou `chat` (*chapitre 15*).

Dans les systèmes de représentation des connaissances par objets comme SHIRKA ou TROEPS (*chapitre 10*), ainsi que dans la quasi-totalité des langages de programmation par objets, la relation d'incompatibilité se déduit de deux autres propriétés spécifiques, l'*exclusivité* (deux classes incomparables sont incompatibles) ou l'*univocité* (c'est la mono-instanciation — tout objet est instance d'une unique classe minimale — habituelle des systèmes à objets, en opposition à la multi-instanciation implicite jusqu'ici dans ce chapitre). Ces trois propriétés sont à la base de toutes les classifications des espèces naturelles. La quasi-totalité des langages de programmation sont univoques : lorsqu'ils sont en *héritage simple*, ils sont de plus exclusifs. Parmi les langages qui nous intéressent, SHIRKA est univoque, mais FROME et TROEPS ne le sont pas, pas plus que les logiques de descriptions.

Notons juste que ces propriétés peuvent s'adresser à l'interprétation du SC (à un ou à tous les modèles), aussi bien qu'à une description du monde ou encore au processus de classification lui-même. Nous prendrons le premier point de vue, contrairement à [Euzenat, 1993b] qui s'en tient au dernier.

### 12.2.3 L'interprétation de l'incomplétude extensionnelle

La sémantique des systèmes d'objets proposée ici fait, au niveau extensionnel, l'hypothèse du *monde ouvert*, comme le font les logiques de descriptions, mais à l'opposé de l'hypothèse du *monde clos* habituelle en programmation logique ou dans les bases de données (*chapitre 5*). Non seulement tous les objets ne sont pas connus, mais les objets connus ne le sont que par certaines classes et certains attributs dont les valeurs sont elles-mêmes incomplètes : c'est une incomplétude extensionnelle.

Le problème est que le SC est utilisé, à un moment donné, pour représenter, non pas n'importe quel modèle, mais un certain *état du monde* (du domaine modélisé) qui n'est connu qu'incomplètement au travers de la description du monde $\mathcal{W}$. On peut considérer une suite croissante (au sens ensembliste) de descriptions consistantes du monde $\mathcal{W}_i, 1 \leq i \leq n$ : le fait que les descriptions soient consistantes et la suite croissante permet de supposer qu'elles décrivent toutes le même monde. Il est probable que pour toute théorie à peu près bien formée, les modèles de $\mathcal{W}_i$ sont des modèles de $\mathcal{W}_j$, pour tout $j < i$. C'est en fait une définition de monotonie. Si l'on assimile cet état du monde à un (sous-ensemble de)[6] modèle(s) de $\mathcal{W}$, toute suite croissante de descriptions $\{\mathcal{W}_i\}_{i\geq 0}$, avec $\mathcal{W} = \mathcal{W}_0$, n'est pas réalisable : seules celles qui conservent ceux que l'on appellera les *modèles du problème* sont possibles.

La difficulté réside dans le fait que ces modèles du problème ne sont pas connus, pas plus du système que de l'utilisateur : c'est d'ailleurs le problème à résoudre. Le raisonnement valide que le système peut mettre en œuvre ne suffit bien sûr pas pour les atteindre. Or, que peut-il faire de plus ? Si l'on exclut un raisonnement hypothétique ou non monotone, rien, si ce n'est déléguer au monde extérieur (l'utilisateur) le soin de compléter ses connaissances pour de nouvelles inférences. Dans ce cadre, l'un des rôles du SC serait de présenter à l'utilisateur des hypothèses plus plausibles que d'autres.

## 12.3 Déduction dans les SC : présentation abstraite

Pendant longtemps, l'étude de la déduction dans les SC (en fait, essentiellement dans les logiques de descriptions) s'est cantonnée aux propriétés calculatoires (décidabilité et complexité) des problèmes, réduisant la déduction elle-même à un algorithme dont on discute, au mieux, la complexité et la complétude. Les approches plus récentes se sont tournées vers des techniques relevant plus de la théorie de la déduction.

De façon abstraite, la déduction dans un SC va consister à définir un connecteur de dérivation $\vdash$, analogue méta-syntaxique de $\models$ mais opérant au plan syntaxique et non plus sémantique : « $\mathcal{H}; \mathcal{W} \vdash d$ » se lit « de $\mathcal{H}$ et $\mathcal{W}$ on peut dériver $d$ », où

6. Le nombre de modèles du problème dépend du genre de problème traité : pour un diagnostic, il n'y a vraisemblablement qu'un modèle, à moins de considérer des hypothèses alternatives. En revanche, pour un problème de conception, il peut y avoir plusieurs solutions alternatives. Enfin, s'il y a plusieurs modèles, on peut chercher à les calculer tous ou se contenter d'un seul.

$\mathcal{H}$ et $\mathcal{W}$ désignent respectivement les composantes intensionnelle et extensionnelle. Ce connecteur peut être défini aussi bien implicitement — au travers d'un algorithme [Nebel, 1990a ; Borgida et Patel-Schneider, 1994], de systèmes de contraintes [Schmidt-Schauß et Smolka, 1991 ; Buchheit *et al.*, 1993 ; Donini *et al.*, 1994] ou de recherche systématique d'un modèle fini [Paramasivam et Plaisted, 1996] —, qu'explicitement, par des règles d'inférence [Borgida, 1992] (voir *chapitre 11*, section 2 et 3).

Les rapports entre déduction et sémantique s'appréhendent classiquement en termes de *correction* — $\mathcal{H};\mathcal{W} \vdash d \Longrightarrow \mathcal{H};\mathcal{W} \models d$ — et de *complétude* (sémantique) — $\mathcal{H};\mathcal{W} \models d \Longrightarrow \mathcal{H};\mathcal{W} \vdash d$.

La littérature sur les logiques de descriptions (*chapitre 11*) abonde en résultats portant sur la décidabilité ou semi-décidabilité de la subsomption dans ces systèmes [Patel-Schneider, 1989], ou sur sa complexité lorsqu'elle est décidable, ce qui amène à envisager un compromis *expressivité-complétude-complexité* [Levesque et Brachman, 1987 ; Attardi, 1991 ; Doyle et Patil, 1991 ; MacGregor, 1991]. Deux conséquences générales peuvent être tirées de ces études :

- un même système classificatoire peut présenter une certaine variabilité de la déduction, ce qui fait perdre toute unicité aux notions dont la définition est basée sur la déduction ;
- une sémantique opérationnelle peut être nécessaire pour expliciter ce que calcule réellement le SC lorsqu'il n'est pas complet : c'est l'objectif affiché des règles d'inférence de [Borgida, 1992].

Dans la suite, nous conserverons un caractère purement abstrait à la déduction, laissant au lecteur le soin de se reporter à [Ducournau, 1996a] et aux quelques références qui précèdent pour la concrétiser. Nous supposerons néanmoins la déduction correcte et monotone, nous penchant exclusivement sur la question de sa complétude et sur les conséquences d'une incomplétude sur la classification.

## 12.4 La classification d'instances

L'un des objectifs des SC est de reconnaître un objet, l'*objet à classer*, comme instance d'une classe et c'est à cet objectif que ce chapitre s'intéresse.

### 12.4.1 La classification dans le cas général

**Inférences de classification**

En suivant l'approche logique à la Gentzen de [Borgida, 1992], le connecteur de dérivation est défini par un ensemble de règles d'inférence que nous illustrerons par deux exemples liés à la classification :

[$\trianglelefteq$-ferm] $$\frac{\mathcal{H};\mathcal{W} \vdash \alpha(o) \qquad \mathcal{H} \vdash \alpha \trianglelefteq \beta}{\mathcal{H};\mathcal{W} \vdash \beta(o)}$$

$$[\mathit{type}\text{-infér}] \qquad \frac{\mathcal{H};\mathcal{W} \vdash \alpha(o) \qquad \mathcal{H};\mathcal{W} \vdash P(o,o')}{\mathcal{H};\mathcal{W} \vdash \beta(o')} \qquad \beta = \mathit{type}_P(\alpha)$$

La première de ces deux règles dit que si $o$ est instance d'une classe $\alpha$ subsumée extensionnellement par $\beta$, il est aussi instance de $\beta$ : c'est la définition de la subsomption extensionnelle et la règle classique de subsomption en théorie des types [Castagna, 1996]. La seconde dit que si $o'$ est valeur d'un attribut $P$ d'un objet $o$ instance de $\alpha$, alors $o'$ est instance de $\mathit{type}_P(\alpha)$ : c'est la version active de la contrainte exprimée par le type des attributs. Incidemment, ce sont les deux seules règles générales de classification dans un SC quelconque.

Parmi les axiomes (12.6-12.12), seul l'axiome (12.9) permet de conclure à l'appartenance d'un objet à l'extension d'une classe : allié à l'axiome (12.12), on obtient la deuxième règle. Toute valeur d'un attribut d'un objet *est* instance des types de l'attribut dans les classes de l'objet. C'est une règle d'inférence de type qui pourrait avantageusement remplacer la vérification de type constituée par l'axiome (12.9) que pratiquent tous les systèmes d'objets : ce n'est qu'une question d'interprétation de la copule (*est*) dans la phrase précédente, par la forme passive *doit être* ou par la forme active *devient*. C'est, au sens strict, une coercition. Cette règle d'inférence a la particularité d'être la seule règle non triviale qui permette le franchissement de la barrière des classes primitives, en permettant de rattacher un objet à une classe primitive, dans la mesure où $\mathit{type}(\alpha, P)$ est une classe qui peut être aussi bien primitive que définie.

Cela vaut bien sûr autant pour les logiques de descriptions que pour la représentation des connaissances par objets. Tout SC, aussi général soit-il, permet (ou devrait permettre) de faire de la classification d'instances automatique, au moins sur les valeurs, c'est-à-dire une inférence de type et pas seulement une vérification de type : la seule condition technique est qu'il permette une migration des instances. Si l'inconsistance logique est assimilée à une erreur de type, cette inférence de type n'est qu'une généralisation, finalement assez élémentaire et naturelle, de la vérification de type.

Si C++, EIFFEL ou la totalité des langages de programmation par objets ne pratiquent évidemment pas cette deuxième inférence (cf. *chapitres 1, 2 et 3*), il faut bien constater que beaucoup de systèmes de représentation des connaissances par objets, dont SHIRKA, ne la pratiquent pas non plus, malgré la simplicité de l'inférence. Les raisons tiennent autant à l'architecture logicielle du système — la vérification de type lors de l'affectation fait partie du noyau du langage sur lequel la classification se greffe de façon assez extérieure — qu'à une philosophie implicite pour laquelle une affectation ne peut pas être la cause d'une classification : si le type est incorrect, ce doit être une erreur de l'utilisateur. Pourtant la notion d'erreur se ramène ici à celle d'inconsistance de la description du monde : si la valeur de l'attribut n'est pas du bon type mais est « classable » dans ce type, cette classification est valide et devrait être faite automatiquement.

### Obstacles à la classification

La déduction en général, et la classification en particulier, se heurtent aux difficultés dues à plusieurs catégories d'incomplétude :

- incomplétude de la description des objets représentés : leurs attributs et les valeurs de ces attributs peuvent être partiellement inconnus (incomplétude extensionnelle) ;
- incomplétude du processus de classification par rapport à la sémantique du système : il opère surtout syntaxiquement, par des comparaisons des intensions qui ne sont pas forcément complètes (incomplétude sémantique) ;
- le langage du système ne permet pas d'exprimer exactement le domaine modélisé (incomplétude syntaxique) ;
- la description des classes n'est pas toujours une condition nécessaire et suffisante d'appartenance pour les objets (incomplétude définitionnelle) : en particulier, dans le cas général, les règles d'inférence permettant de classer sont très limitées.

Incidemment, on peut remarquer que l'approche des systèmes classificatoires diffère des bases de données (*chapitre 5*) par la première forme d'incomplétude, et des logiques de descriptions (*chapitre 11*) par la dernière.

Il s'agit donc d'essayer de surmonter ou de contourner les obstacles dressés par ces différentes catégories d'incomplétude.

### 12.4.2 La classification incertaine

Pour surmonter ces obstacles, l'idée est d'une part de distinguer une classification valide et automatique d'une classification incertaine mais interactive : si l'on ne peut pas déterminer automatiquement les classes de l'objet à classer dans tous les modèles, on se retourne vers l'utilisateur pour qu'il choisisse certains modèles. C'est donc une classification interactive. Il va s'agir d'autre part de reproduire, dans la classification, les deux versants sémantique et syntaxique (ou extensionnel et intensionnel), de notre cadre abstrait.

#### Classification incertaine, version extensionnelle

Une approche sémantique de la classification conduit à définir trois catégories de classes relativement à l'objet à classer $o$. Une classe $\beta$ peut être :

**valide** – c'est une classe de $o$ dans tous les modèles de $\mathcal{W}$ : $\mathcal{W} \models \beta(o)$ ;
**satisfiable** – c'est une classe de $o$ dans au moins un modèle de $\mathcal{W}$ : $\exists \mathcal{M},\ \models_{\mathcal{M}} \beta(o)$ ;
**insatisfiable** – ce n'est une classe de $o$ dans aucun modèle de $\mathcal{W}$ : $\forall \mathcal{M},\ \not\models_{\mathcal{M}} \beta(o)$.

Sur l'emploi du terme « (in)satisfiable », cf. note 4, page 359. On introduirait de même des classes *falsifiables*.

Si la description du monde est consistante, les classes valides sont satisfiables (sinon, toutes les classes sont valides et aucune n'est satisfiable). Dans les logiques de descriptions, la classification d'instance consiste à calculer des classes valides, tout ou partie suivant la complétude sémantique de la classification, sans s'intéresser aux classes satisfiables ou insatisfiables. Dans une approche plus interactive qui est celle de la représentation des connaissances par objets, l'objectif est de soumettre à l'utilisateur des classes satisfiables en lui demandant s'il veut y rattacher l'objet : un tel rattachement revient en fait à éliminer tous les modèles dans lesquels l'objet n'est pas instance de la classe.

Pour la notation, on introduit trois prédicats $Val$, $Sat$ et $InSat$ : $Val(\alpha, o)$ signifie que $\alpha$ est une classe valide pour $o$. Cette notation montre bien la symétrie des qualificatifs : $o$ est valide pour $\alpha$ autant que $\alpha$ est valide pour $o$. Les ensembles de classes valides et satisfiables sont des *idéaux*[7] aussi bien de la relation de spécialisation $\preceq$ que de la subsomption extensionnelle $\trianglelefteq$ (si l'on peut encore parler d'idéal pour un préordre). Quant aux classes insatisfiables, elles constituent un idéal de la relation duale.

Comme seuls les modèles du problème intéressent réellement un utilisateur, il faudrait définir les catégories de classes correspondantes, obtenues par une quantification sur ce sous-ensemble de modèles : on obtiendrait des classes *solution valide*, *solution* ou *non solution*. Cette définition est un peu vaine puisqu'il n'y a aucun moyen de la concrétiser, mais c'est un moyen de rappeler les limites de cette approche : le mieux reste de soumettre à l'utilisateur une classe solution et non pas simplement une classe satisfiable. Si l'utilisateur choisit une classe qui n'est pas solution, donc qui ne correspond pas à l'état du monde courant, une contradiction ultérieure est inéluctable. Cette contradiction se traduira par une inconsistance, mais elle n'est pas appréhendable, *au moment du choix*, dans les catégories sémantiques du SC.

### Classification incertaine, version intensionnelle abstraite

Au niveau intensionnel abstrait, on a bien envie de simplement remplacer le connecteur sémantique de conséquence $\models$ par le connecteur syntaxique de dérivation $\vdash$. Cela marche très bien pour les classes valides. Ainsi, une classe $\beta$ pourra être :

**sûre** – c'est une classe dont la validité est effectivement déduite : $\mathcal{H}; \mathcal{W} \vdash \beta(o)$.

Malheureusement, ce remplacement systématique de $\models$ par $\vdash$ n'est pas possible, en l'état, pour les deux autres catégories. Une solution consiste à introduire les prédicats $Val$, $Sat$ et $InSat$ dans le système de déduction. Une classe $\beta$ pourra alors être :

**sûre** – la validité de $\beta$ est effectivement déduite : $\mathcal{H}; \mathcal{W} \vdash Val(\beta, o)$ ;
**possible** – la satisfiabilité de $\beta$ est effectivement déduite : $\mathcal{H}; \mathcal{W} \vdash Sat(\beta, o)$ ;
**impossible** – l'insatisfiabilité de $\beta$ est effectivement déduite : $\mathcal{H}; \mathcal{W} \vdash InSat(\beta, o)$.

On introduirait de même des classes *falsifiées*. Enfin, on dira qu'une classe est *incertaine* si elle n'est ni possible, ni impossible et qu'elle est *envisageable* si elle n'est pas *impossible*. Comme pour la classification extensionnelle, les ensembles de classes sûres, possibles ou impossibles sont des idéaux de la relation de spécialisation (ou de sa duale), mais pas forcément de la subsomption extensionnelle ou intensionnelle.

7. Un *idéal* d'une relation d'ordre partiel $(E, <)$ est une partie $F \subseteq E$ contenant ses majorants : $x < y \ \& \ x \in F \Rightarrow y \in F$.

Le fait de se servir des prédicats $Sat$ et $InSat$ à droite du connecteur de dérivation amène à les utiliser aussi à gauche : le langage de la description du monde s'enrichit donc de ces deux prédicats (c'est déjà implicite pour $Val$). Il est donc possible de faire un *rattachement négatif* en disant, par $InSat(\alpha, o)$, qu'un objet n'est pas instance d'une classe [8] : sémantiquement, cela revient à éliminer tous les modèles dans lesquels l'interprétation de l'objet appartient à l'interprétation de la classe. En revanche, affirmer, par $Sat(\alpha, o)$, qu'un objet « peut être instance d'une classe » n'est pas très clair sémantiquement, puisque cela ne peut pas se traduire par l'élimination de certains modèles : $Sat$ est une propriété momentanée de la description du monde, qui est monotone pour la déduction mais ne sera pas forcément conservée lors de l'évolution monotone de la description du monde. On conviendra donc que $Sat$ et $InSat$ peuvent tous deux figurer à gauche de $\vdash$, mais que seul $InSat$ peut être affirmé. On pourrait introduire des prédicats méta-syntaxiques $Sur$, $Pos$ et $ImPos$.

**Classification incertaine, version structurelle**

Comment concrétiser ces définitions abstraites ? Intuitivement et approximativement, la comparaison des descriptions d'un objet et d'une classe conduit à considérer trois catégories de classes :

- les classes *impossibles*, dont $o$ viole au moins une des contraintes :

$$\exists P \in \mathcal{P}_\beta \cap Attr_\mathcal{W}(o) : P_\mathcal{W}(o) \notin Dom_P(\beta) \tag{12.18}$$

- les classes *possibles*, dont $o$ ne viole aucune contrainte :

$$\forall P \in \mathcal{P}_\beta \cap Attr_\mathcal{W}(o) : P_\mathcal{W}(o) \in Dom_P(\beta) \tag{12.19}$$

- les classes *sûres*, dont $o$ vérifie toutes les contraintes :

$$\forall P \in \mathcal{P}_\beta : P_\mathcal{W}(o) \in Dom_P(\beta) \tag{12.20}$$

Dans tous les cas, il s'agit d'une classification intensionnelle, voire structurelle, au sens où elle consiste en une comparaison de deux descriptions sans aucune influence d'un tiers. Le choix du vocabulaire étant à la fois délicat (comme le lecteur doit déjà s'en douter et comme la suite le confirmera) et limité, pour ne pas introduire de nouveaux adjectifs, on préfixera les trois adjectifs précédents de l'adverbe *structurellement*, en leur associant les trois prédicats $SSur$, $SPos$ et $SImPos$. Comme pour la classification extensionnelle, les ensembles de classes structurellement sûres, possibles ou impossibles sont des idéaux de la relation de spécialisation (ou de sa duale), mais aussi de la subsomption intensionnelle (mais pas de l'extensionnelle).

Dans les définitions (12.18-12.20), la comparaison entre la valeur et le domaine de l'attribut est doublement approximative : d'une part, la relation entre les valeurs

8. Cela revient à disposer de la négation, mais uniquement au niveau extensionnel, un peu comme les règles de CLASSIC [Resnick *et al.*, 1995 ; Borgida et Patel-Schneider, 1994] peuvent être utilisées pour la classification d'instances mais pas pour la subsomption.

de $P$ et $type(\beta, P)$ peuvent elles-mêmes être qualifiées, récursivement, de structurellement sûre, possible ou impossible ; d'autre part, la cardinalité peut n'être que provisoire et toutes les valeurs pas encore connues. De plus, la formulation utilisée laisse entendre qu'il s'agit d'une comparaison effective entre la valeur et le domaine : en réalité, il peut aussi s'agir d'une comparaison entre le domaine de l'attribut pour l'objet, tel qu'il résulte de ses classes connues, avec le domaine de l'attribut dans $\beta$ : il peut en effet exister $\alpha \in Isa_{\mathcal{W}}(o)$ tel que $Dom(\alpha, P) \subseteq Dom(\beta, P)$ ou $Dom(\alpha, P) \cap Dom(\beta, P) = \emptyset$, ce qui permet de conclure malgré l'incomplétude extensionnelle. En outre, les classes de $o$ déjà connues peuvent être incompatibles avec $\beta$ (au sens de $\nabla$), ce qui assure l'impossibilité. Enfin, les comparaisons varient notablement entre (12.19) et (12.18) d'un côté et (12.20) de l'autre : dans un cas, il s'agit d'éliminer ce qui est évidemment impossible alors que, dans l'autre cas, il s'agit de sélectionner ce qui est sûr, les deux modalités n'étant pas complémentaires. Le lecteur est renvoyé à [Ducournau, 1996a] pour des formulations plus exactes mais cette approximation nous suffira ici.

Voyons d'abord où nous souhaitons, et jusqu'où nous pouvons espérer, en arriver : dans l'idéal, nous aimerions bien qu'une classe *structurellement impossible* (resp. *possible*, *sûre*) soit *impossible* (resp. *possible*, *sûre*), c'est-à-dire *insatisfiable* (resp. *satisfiable*, *valide*). Plus précisément, au stade d'abstraction où nous en sommes, y a-t-il une définition structurelle qui vérifie ces souhaits ?

Seul le premier point va combler notre attente : une classe structurellement impossible est effectivement, et évidemment, insatisfiable. Pour les autres, il va falloir se rabattre sur des objectifs moins ambitieux :

- pour une classe *structurellement sûre*, deux options se présentent :
  - la version forte en fait une classe *valide* lorsqu'elle est *définie*

    (12.21) $$SSur(\alpha, o) \ \ \& \ \ \alpha \text{ défini} \Longrightarrow Val(\alpha, o)$$

    ce serait en fait la définition abstraite des classes structurellements sûres : ce sont les classes qui sont valides quand elles sont définies. Cette équation vient utilement compléter la propriété d'idéal de $SSur$ :

    (12.22) $$\alpha \trianglerighteq \beta \Longrightarrow \big(\forall o : SSur(\alpha, o) \Rightarrow SSur(\beta, o)\big)$$
  - la version faible (que l'on qualifiera de structurellement *probable*) en fait une classe privilégiée pour un rattachement interactif : la consistance du rattachement n'est pas garantie, mais elle est « probable » ;
- pour une classe *structurellement possible* : elle n'est qu'*envisageable* et, tant qu'elle n'est pas prouvée *insatisfiable*, elle peut servir, soit pour des inférences très spécifiques, soit comme « second choix » pour une classification interactive.

### 12.4.3 Correction et complétude : les illusions de la classification structurelle

Reste à savoir dans quelle mesure l'interprétation souhaitée de ces qualificatifs de *structurellement sûr* et *possible* est correcte. Une classe structurellement sûre

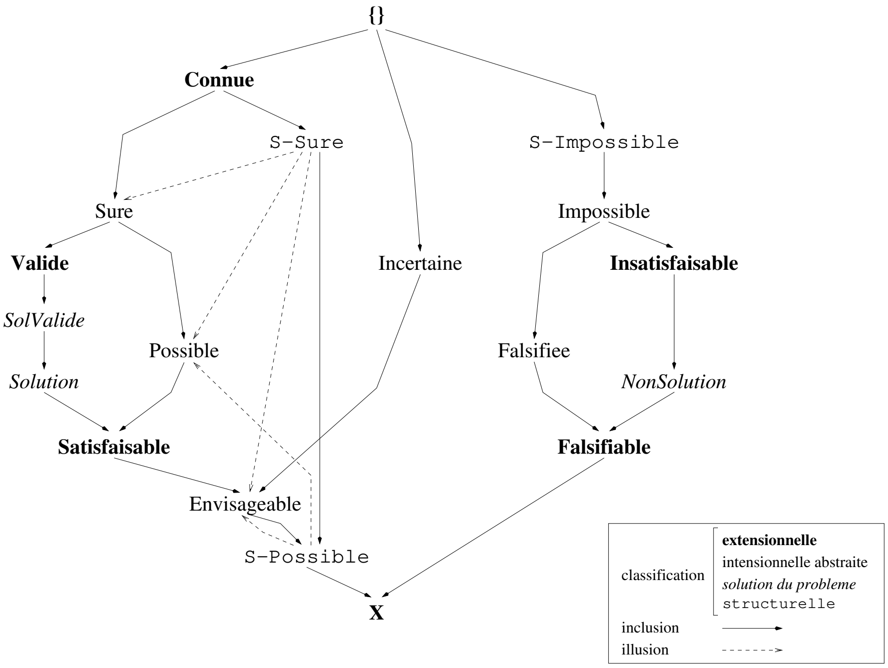



FIG. 12.1: *Inclusion des ensembles de classes résultant des classifications extensionnelle, intensionnelle et structurelle, avec les solutions du problème, inaccessibles, pour une description du monde consistante.*

est-elle valide, satisfiable ou seulement envisageable ? Une classe structurellement possible est-elle satisfiable ? Est-elle même envisageable ?

Notons d'abord que la question de la correction ne présente pas vraiment de difficulté :

- pour la définition abstraite des classes sûres, possibles, impossibles et incertaines, qui est correcte par construction dès lors que le connecteur de dérivation est présumé correct ; rappelons qu'il n'y a pas de définition concrète et unique des classes possibles, sûres et impossibles. Ce ne sont que des cas d'espèces dont chaque SC spécifique donne une définition concrète particulière, éventuellement implicite (c'est-à-dire purement opérationnelle ou algorithmique) par son système de déduction.

- pour les classes *structurellement impossibles* qui sont clairement insatisfiables. En corollaire, une classe satisfiable est forcément structurellement possible, de même qu'une classe incertaine ou structurellement sûre (figure 12.1) ; si jamais la réciproque était vraie, il y aurait alors identité entre les notions structurelles et extensionnelles.

La complétude est en revanche plus difficile à atteindre. En particulier, il n'est pas nécessaire pour une classe insatisfiable d'être structurellement impossible, pas plus que pour une classe valide d'être structurellement sûre : l'incomplétude extensionnelle en dispense.

Qu'elles soient extensionnelles (valide, satisfiable, insatisfiable) ou intensionnelles (sûre, possible, impossible) les notions abstraites se définissent d'une manière entièrement indépendante des spécificités du SC considéré. Lorsque la déduction est complète, ces notions sont deux à deux coextensives. L'un des problèmes soulevés par [Borgida, 1992 ; Borgida et Patel-Schneider, 1994] est que les algorithmes, lorsqu'ils sont incomplets, ne permettent pas toujours de caractériser ce qu'ils calculent. Les notions structurelles peuvent alors devenir intéressantes pour cette caractérisation, mais il n'y pas de raison *a priori* qu'elles coïncident avec les notions abstraites.

Deux démarches opposées sont ainsi à mettre en œuvre :

- formaliser le système de déduction effectif, par exemple avec des règles d'inférence, pour donner une caractérisation concrète, aussi déclarative que possible, des catégories de classes effectivement calculées ;
- étudier la correction de définitions générales comme celles de la version structurelle dont les définitions sont, en réalité, inadéquates, ni correctes ni complètes, ni nécessaires ni suffisantes.

Sans anticiper sur la suite, soulignons que le seul intérêt de la classification structurelle réside dans l'incomplétude sémantique de la déduction ou la trop grande complexité de la classification extensionnelle.

#### Classification non intensionnelle

Pour qu'une classe définie soit valide, il suffit que la classe soit structurellement sûre pour l'objet à classer (12.21). Par ailleurs, plusieurs catégories d'inférence permettent de déduire qu'une classe, aussi bien primitive que définie, est *valide*, même en cas d'incomplétude extensionnelle de l'objet :

- d'abord, l'inférence de type, présentée comme élémentaire, qui permet de classer un objet lorsqu'il est valeur d'un attribut (règle [$type$-infér]) ;
- ensuite, certaines constructions des logiques de descriptions comme la quantification existentielle qualifiée du langage $\mathcal{ALE}$, l'intersection de rôles de $\mathcal{ALR}$ [Donini *et al.*, 1994] ou les *sous-rôles* de [Nebel, 1990a] : Nebel donne en particulier l'exemple de l'`Équipe-moderne` dans lequel un objet est rattaché à une classe pour la raison qu'il est le seul à pouvoir remplir un certain rôle (figure 12.2). Notons que l'inférence due à ces constructions porte sur la valuation d'un attribut, la classification de la valeur s'en déduisant par [$type$-infér].
- enfin, les éventuelles propriétés structurelles spécifiques du SC :
  - si le SC est exhaustif et qu'il n'y a qu'un seul puits envisageable, ce puits est une classe *valide* ;

| $\mathcal{H}$ | $\mathcal{W}$ |
|---|---|
| `Human` $\dot{\leq}$ `Anything`<br>`Man` $\dot{\leq}$ `Human`<br>`Woman` $\dot{\leq}$ `Human`<br>`disjoint(Man,Woman)`<br>`Set` $\dot{\leq}$ `Anything`<br>`member` $\dot{\leq}$ `anyrelation`<br>`Team` $\doteq$ `and(Set, all(member,Human))`<br>`leader` $\dot{\leq}$ `member`<br>`Modern-Team` $\doteq$ `and(Team, atmost(3,member), atleast(1,leader), all(leader,Woman))` | `Modern-team(TEAM-A)`<br>`Man(DICK)`<br>`Man(HARRY)`<br>`member(TEAM-A,DICK)`<br>`member(TEAM-A,HARRY)`<br>`member(TEAM-A,MARY)` |

FIG. 12.2: *L'exemple de l'`Équipe-moderne` de [Nebel, 1990a] (cf.* chapitre *11) : à part `MARY`, tous les `member` de `TEAM-A` sont des `Man`, tous les `member` sont connus, il y a au moins un `leader` et tous les `leader` sont des `Woman`. On a donc $\mathcal{H};\mathcal{W} \models$ `leader(TEAM-A,MARY)`, et par conséquent $\mathcal{H};\mathcal{W} \models$ `Woman(MARY)`.*

– si le SC est univoque et lorsqu'elle existe, la *borne inférieure* de tout ensemble de classes valides est une classe valide ; plus généralement, si cet ensemble de classes a un seul *minorant maximal* envisageable, ce minorant maximal est valide.

Ces deux inférences se généralisent de la même manière : lorsqu'une hypothèse existentielle de validité porte sur un ensemble de classes (dans chaque modèle, l'objet est instance d'une de ces classes[9]), les super-classes communes à toutes celles qui sont envisageables sont valides. Les contrapposées sont analogues : l'absence de classe envisageable dans l'ensemble considéré entraîne une inconsistance. Si une classe valide non puits n'a pas de sous-classe envisageable (exhaustivité), ou si deux classes valides n'ont pas de minorant maximal envisageable (univocité), la description du monde est inconsistante.

Dans ces divers cas, la déduction ne repose plus sur une comparaison « locale » de deux termes mais sur une énumération de tous les candidats qui ressemble assez aux algorithmes de résolution des CSP (*chapitre 9*). En particulier, une classe peut être valide sans qu'il soit nécessaire qu'elle soit structurellement sûre : il suffit qu'elle ne soit pas structurellement impossible (il y aurait sinon une inconsistance).

En corollaire, si une classe structurellement possible mais pas sûre a de grandes chances d'être falsifiable (il suffit d'affecter à un attribut une valeur en dehors du domaine de l'attribut dans la classe), ce n'est pourtant pas certain : la classe pourrait être valide et toute tentative de falsification mènerait à une inconsistance.

9. Ce qui ne signifie pas que l'une au moins est valide. Il faut éviter de commuter les deux quantifications. Il s'agit de $\forall\mathcal{M}, \exists\alpha \in A : \models_{\mathcal{M}} \alpha(o)$ et non pas de $\exists\alpha \in A, \forall\mathcal{M} : \models_{\mathcal{M}} \alpha(o)$ qui équivaut à $\exists\alpha \in A : \mathcal{W} \models \alpha(o)$.

Les classes sûres sont constituées de ces sous-ensembles de classes valides, effectivement calculables (et calculées). Le détail de ces classes sûres est relativement secondaire : ce qui compte est qu'il en existe et qu'elles n'ont même pas besoin d'être structurellement sûres. Il y a toujours moyen de faire un peu de classification automatique, même avec des classes primitives.

### Les classes satisfiables

Dans le cadre d'une classification interactive, il est évidemment essentiel de pouvoir garantir à l'utilisateur que le rattachement d'une instance à une classe est *possible*, au sens où il mène avec certitude à une solution, et pas seulement qu'il est *possible* que ce rattachement mène à une solution.

Une classe *structurellement sûre* est-elle bien *satisfiable* ? Vu l'aspect « structurel » des définitions, il est à craindre que non : il suffit d'introduire un élément non structurel comme l'incompatibilité ou les sous-rôles pour construire un contre-exemple.

Supposons que $o$ ait deux classes $\alpha$ et $\beta$ structurellement possibles (voire sûres) et que ces deux classes soient incompatibles ($\alpha \nabla_a \beta$), le tout sans que la description du monde soit inconsistante. En l'absence d'autre contrainte, ces deux classes sont à la fois satisfiables et falsifiables : chacune est la classe de $o$ dans certains modèles, mais pas dans d'autres à cause de l'incompatibilité. Mais rien n'empêche — c'est-à-dire que cela ne conduit pas forcément à une inconsistance — que $o$ soit la valeur d'un attribut d'un objet dont le type est $\beta$ :

$$P(o', o), \gamma(o') \in \mathcal{W} \text{ et } type(\gamma, P) = \beta,$$

ce qui rendrait $\beta$ valide (par [$type$-infér]) et $\alpha$ insatisfiable. Bien sûr, la validité de $\beta$ peut avoir une cause plus complexe, par exemple se déduire par les inférences spécifiques de l'exhaustivité ou de l'univocité, voire des sous-rôles.

Ajouter des contraintes au SC comme le fait [Gensel, 1995] (voir *chapitre 9*, section 3) conduit au même genre de difficulté : comme calculer l'existence d'une solution (consistance globale) est aussi difficile que le calcul d'une solution, on se limite à une consistance locale (ex. consistance d'arc) qui fait que le possible traduit plus un espoir qu'une réalité !

Les sous-rôles, qui ne sont que des cas particuliers de contraintes, illustrent bien ces difficultés : l'exemple de l'`Équipe-moderne` de [Nebel, 1990a] est une instance parfaite de l'exemple abstrait précédent. `Homme` et `Femme` sont tous deux des classes structurellement possibles (voire sûres) incompatibles, mais l'une est valide, l'autre insatisfiable. Et le calcul de la validité de la classe est trop complexe (NP-complet) pour que Nebel accepte d'incorporer ce genre d'inférence dans ses algorithmes.

### Complétude sémantique

Pour avoir des chances d'obtenir l'interprétation idéale de ces notions structurelles, il faudrait donc les définir,

- en tenant compte de l'incompatibilité ;

- à partir d'une description du monde $\mathcal{W}$ fermée par $\models$, c'est-à-dire contenant toutes ses conséquences valides : $\mathcal{W} = \{d \mid \mathcal{W} \models d\}$.

Alors seulement peut-on espérer que, dans cette description du monde fermée, toute classe structurellement possible ou sûre sans être incompatible est satisfiable, et peut être considérée, arbitrairement, comme une solution.

Mais il faut bien voir ce que cela va impliquer :

- si les classes structurellement possibles sont satisfiables, elles sont donc possibles puisque manifestement calculables, ce qui conduit à l'équivalence d'une part, des classes satisfiables, possibles, envisageables et structurellement possibles, d'autre part, par complément, des classes insatisfiables, impossibles et structurellement impossibles (figure 12.1) ; en corollaire, il n'y a pas de classe incertaine ; cela signifie donc que l'on a obtenu la complétude sémantique pour les prédicats $Sat$ et $InSat$.

- pour que les classes structurellement sûres soient satisfiables, il faut assurer la complétude sémantique, au moins pour les classes valides, mais sans doute aussi sur les classes satisfiables et insatisfiables [10]. Or, si l'on a la complétude sur les classes (in)satisfiables, la correction des classes structurellement possibles et sûres suit trivialement : il suffit d'inclure la satisfiabilité dans leur définition. Plus sérieusement, les définitions structurelles n'ont de réel intérêt qu'en cas d'incomplétude : dès que la déduction est complète pour tous les prédicats $Val$, $Sat$ et $InSat$, il devient inutile de chercher à caractériser ce qui est calculé.

- vouloir faire des classes structurellement sûres des classes valides est encore plus prétentieux : cela signifie que l'on fait de toute classe une classe définie. On verra plus loin qu'il faut s'entourer de quelques précautions avant de pouvoir l'envisager en toute sécurité.

Bien sûr, si la correction s'avère délicate, la complétude n'était même pas revendiquée jusqu'ici, entraînant la coexistence de classes caractérisées au niveau sémantique (valide, satisfiable, insatisfiable) et au niveau syntaxique (sûre, possible, impossible, etc.) Or, en un retournement inattendu, la correction de l'interprétation souhaitée des notions structurelles les plus syntaxiques semble nécessiter finalement la complétude sémantique : on ne garderait donc plus que les classes valides, satisfiables et insatisfiables (identiques respectivement aux classes sûres, possibles et impossibles). Rien n'empêche bien sûr, pour un SC particulier, d'essayer de relever ce défi de la complétude, mais le cas général semble désespéré comme le montre toute la littérature consacrée à la décidabilité et à la complexité de la subsomption [Nebel, 1990a ; Patel-Schneider, 1989] ou de la vérification d'instance [Donini *et al.*, 1994]. De toute façon, cette complétude enlèverait tout intérêt pratique aux définitions structurelles.

10. Considérons deux uniques puits structurellement possibles, dont un insatisfiable : l'autre est donc valide (on suppose ici que la seule façon de prouver la validité de l'un passe par l'insatisfiabilité de l'autre).

**Version dynamique**

Finalement, les classes *structurellement possibles* et *impossibles* représentent la meilleure (ou la plus simple) approximation des classes *envisageables* et *impossibles*, respectivement par excès ou par défaut. Quant au caractère priviligié des classes *structurellement sûres*, il résulte uniquement de la maîtrise de l'incomplétude extensionnelle : vis-à-vis de la classe, l'objet à classer est parfaitement connu. Aucun retournement n'est à craindre d'un attribut dont le dévoilement viendrait contredire l'attachement de l'objet à une classe. Mais le retournement peut très bien venir d'ailleurs !

L'erreur est de chercher une définition statique, indépendante de l'état de la déduction : les définitions statiques sont abstraites et non structurelles, alors que les définitions structurelles ne peuvent être utilisées que dynamiquement.

- pour les classes *structurellement impossibles* ou *sûres*, il s'agit d'une propriété monotone, qui peut donc servir à faire des inférences de façon monotone ;
- pour les classes *structurellement possibles*, il s'agit d'une propriété non monotone — autant pour la déduction que pour l'évolution monotone de la description du monde — qui ne peut servir à faire des inférences que sous des formes bien particulières, comme la forme existentielle présentée plus haut : si l'une au moins des classes d'un ensemble doit être satisfaite, toutes les super-classes communes des classes possibles de cet ensemble sont valides.

Le lecteur trouvera dans [Ducournau, 1996a] comment concrétiser ces définitions avec des règles d'inférence.

### 12.4.4 Comparaisons et historique

La détermination de ces diverses catégories de classes est à la base de plusieurs systèmes de représentation des connaissances par objets comme SHIRKA, TROEPS ou FROME (*chapitre 10*). Mais, depuis son introduction dans SHIRKA [Pivot et Prokop, 1987], la classification incertaine a subi divers avatars dus à des malentendus ou à des variantes. De plus, cette approche n'est pas la seule à essayer d'élargir le champ d'application de la classification.

La génèse de cette idée de classification incertaine n'est pas totalement connue : a-t-elle été déjà exploitée avant SHIRKA ou indépendamment de lui ? Il semble que non si l'on en croit [Euzenat et Rechenmann, 1995], mais on peut lui trouver divers inspirateurs complémentaires : la classification interactive de [Finin, 1986] ou de systèmes *ad hoc* comme CLASSIC [Granger, 1986 ; Granger, 1988], la classification à base de règles de [Fikes et Kehler, 1985] ou d'un autre système *ad hoc*, DIVA [David et Krivine, 1987 ; David et Krivine, 1988], et bien sûr la classification certaine de KL-ONE [Brachman et Schmolze, 1985].

**Sur le sens de sûr**

L'historique des qualificatifs de *sûr* et *possible* et de leur interprétation nécessite quelques éclaircissements (figure 12.3). Le problème vient des deux interprétations

| SC | LMO'96 | TROEPS95 | FROME | TROEPS90 | SHIRKA |
|---|---|---|---|---|---|
| valide | — | | sûre | sûre | |
| satisfiable | — | possible | | | sûre |
| insatisfiable | — | impossible | impossible | impossible | impossible |
| S. sûre & définie | sûre | | sûre | | |
| S. sûre | probable | possible | | sûre | sûre |
| S. probable | | | | | |
| S. possible | possible | inconnue | possible | possible | possible |
| S. impossible | impossible | impossible | impossible | impossible | impossible |

FIG. 12.3: *Significations et caractérisations successives de* sûr *et* possible*. Une case vide traduit le fait que le système ne prend pas en compte la modalité.*

différentes de *sûr* : « il est *sûr* que l'objet est instance de la classe » ou bien « il est *sûr* que l'objet peut être rattaché à la classe ». Suivant l'interprétation, la classe est valide ou satisfiable. Parallèlement à ces deux interprétations, *possible* se comprend comme « il est sûr que l'objet *peut* être rattaché à la classe » ou bien « il est *possible* que l'objet puisse être rattaché à la classe ».

En SHIRKA qui les a introduites, la signification de ces notions était claire : une classe *sûre* est une classe à laquelle il est sûr que l'objet peut être rattaché, une classe *satisfiable* donc [Rechenmann, 1988]. En revanche, une classe *possible* n'est qu'une classe qui n'est pas immédiatement reconnue comme *impossible* : c'est donc une classe *incertaine* ou *possible*, *structurellement possible* en fait.

Par la suite, [Mariño, 1993] interprète le qualificatif de *sûr* par *valide* — comme s'il s'agissait de classification automatique sur des classes définies —, provoquant ainsi une certaine confusion (TROEPS90). De son côté, dans FROME, [Dekker, 1994] utilise *sûr* pour des classes dont les conditions suffisantes sont vérifiées, ce qui en justifie l'interprétation comme des classes valides. Pour éliminer tout risque de confusion, [Euzenat et Rechenmann, 1995] proposent maintenant pour SHIRKA et TROEPS les qualificatifs de *possible* et *inconnu*, à la place de *sûr* et *possible* (TROEPS95) [11].

Enfin, dans une version préliminaire de ce travail [Ducournau, 1996b], les versions extensionnelle et intensionnelle de la classification incertaine étaient distinguées, mais en mélangeant les définitions abstraites et concrètes (LMO'96).

La discussion sur la correction et la complétude de la classification structurelle (cf. paragraphe 12.4.3, page 367) montre en fait que la modalité « sûre », la moins incertaine de la classification incertaine à la SHIRKA, présente beaucoup d'incertitude. Quel que soit son qualificatif, son interprétation par « satisfiable » est correcte en SHIRKA parce qu'il ne fait curieusement aucune inférence automatique — pas plus celle de la règle [$type$-infér] (puisque les valeurs d'un attribut doivent être du bon type lors de l'affectation) que celle de l'univocité (puisque l'objet ne peut jamais être classé que dans *une* sous-classe de son unique classe minimale courante) — tout en imposant un état « correct », la fermeture par $\models$, à tout instant. Mais si l'on imagine le système idéal réalisant la sémantique latente de SHIRKA, et met-

11. On reprochera juste à ce nouveau triptyque le faux-ami provoqué par la non complémentarité de *possible* et *impossible*, ce qui ne nous a pas empêché de les reprendre dans un sens analogue.

tant en œuvre aussi bien la règle [$type$-infér] que l'inférence spécifique de l'univocité (la première nécessite la seconde, à moins d'imposer une structure de *treillis*), il faudrait alors que le calcul des classes valides, satisfiables et insatisfiables soit complet pour assurer la correction des définitions structurelles, ce qui leur retirerait immédiatement tout intérêt.

Plusieurs raisons expliquent les difficultés rencontrées pour connaître la définition exacte de la classification incertaine donnée par les différents auteurs cités (figure 12.3) :

- d'abord, les définitions ne sont pas assez formelles pour être réellement univoques et il est difficile de les distinguer des éventuelles caractérisations ;
- ensuite, deux discours parallèles ont toujours été tenus, à quelques pages ou quelques années d'intervalle, sans être jamais explicitement corrélés : l'un est extensionnel, l'autre intensionnel ou plutôt structurel. Ainsi, dans [Rechenmann, 1988], la définition est implicitement extensionnelle, mais la caractérisation donnée dans les exemples ou ailleurs (par exemple dans [Euzenat et Rechenmann, 1995]) est explicitement structurelle.
- enfin, les successeurs ont persévéré dans l'incertitude, voire l'ont aggravée, en présentant les classes sûres de SHIRKA comme des classes valides, en oscillant entre les définitions structurelles et extensionnelles ou en ne présentant la classification de SHIRKA que sous une forme structurelle ;
- il n'y a en fait aucune notion de modèle sous-jacente, uniquement un *domaine modélisé* : dans tous les cas, les classes possibles sont interprétées comme inconnues et non comme satisfiables.

Ce dernier point fournit une méta-analyse à peu près complète de ces confusions dont la cause principale réside manifestement dans la non distinction des niveaux sémantique et syntaxique.

**Tropisme classificatoire**

Les propriétés spécifiques comme l'univocité ou l'exhaustivité offrent, de même que le caractère défini des classes, un tropisme descendant qui se traduit par le fait que, *dans certaines conditions*, un objet peut être classé automatiquement dans une sous-classe de sa classe courante. C'est ce tropisme descendant qui fait la puissance de la classification, mais il a une contre-partie contraignante : ces « certaines conditions » sont en général plus ou moins structurelles et elles ont une influence intensionnelle non négligeable sur la hiérarchie des classes et sur leur cohérence, c'est-à-dire sur l'existence d'instances dans une description du monde consistante. La subsomption extensionnelle peut ainsi être vue comme une conséquence de la classification :

$$\big(\forall o : Val(\alpha, o) \Rightarrow Val(\beta, o)\big) \Longrightarrow \alpha \trianglelefteq \beta \qquad (12.23)$$

de même que la subsomption intensionnelle se déduit des comparaisons structurelles :

$$\big(\forall o : SSur(\alpha, o) \Rightarrow SSur(\beta, o)\big) \Longrightarrow \alpha \trianglerighteq \beta \qquad (12.24)$$

ce qui symétrise (12.22). Mais ces implications ne peuvent pas être exploitées sous cette forme extensionnelle, puisque la déduction ne peut pas énumérer toutes les instances potentielles d'une classe. (12.24) doit en fait se traduire par des définitions concrètes de $SSur$ et $\trianglerighteq$ assurant leur équivalence. Quant à (12.23), couplée à (12.21) et (12.22), elle amène à constater que le caractère défini des classes est une cause de subsomption :

$$\alpha \trianglerighteq \beta \;\;\&\;\; \beta \text{ définie } \Longrightarrow \alpha \trianglelefteq \beta \qquad (12.25)$$

Ce constat a conduit les logiques de description à la classification automatique des classes dont l'effet consiste à remplacer la subsomption extensionnelle ($\trianglelefteq$) par la spécialisation ($\preceq$) dans la dernière équation.

Cette classification automatique devrait être appliquée *a fortiori* à un système d'objets univoque qui proposerait des classes définies : il faut à tout prix (sous peine d'incohérence) éviter d'avoir des classes incompatibles de même intension. Or, l'univocité conduit inexorablement à l'existence de classes définies : lorsqu'elle existe, la borne inférieure de deux classes est effectivement définie à partir de ces deux classes. Ce tropisme classificatoire peut faire apparaître des objets paradoxaux, dont tous les attributs vérifient scrupuleusement les domaines de leurs classes mais dont la seule existence provoque une inconsistance : c'est le cas par exemple de deux classes définies et incompatibles dont l'union des intensions est non vide, ou bien, dans le cas de l'univocité, si l'intension de la borne inférieure est strictement « plus grande » que la réunion ($\sqcup$) de l'intension des deux classes.

Il semble donc que les systèmes de représentation des connaissances à la SHIRKA ne peuvent pas faire plus longtemps l'économie d'une analyse approfondie de leur tropisme classificatoire et des mécanismes du niveau intensionnel susceptibles d'en limiter les effets pervers : la classification de classes, bien sûr mais pas seulement [12].

### Classification incertaine et logiques de descriptions

À première vue, les logiques de descriptions ne s'intéressent qu'à la classification automatique. Pourtant, la plupart sont très proches de la classification incertaine. Lorsque la logique de descriptions considérée sait calculer la relation d'incompatibilité, par exemple lorsqu'elle possède une négation généralisée comme $\mathcal{ALC}$ [Schmidt-Schauß et Smolka, 1991 ; Donini *et al.*, 1994], c'est en fait implicite : appliqué aux instances cela donne déjà le qualificatif *impossible*. *Sûr* est fourni par la classification et *possible* (en fait *envisageable*) suit, par complément. La complétude est bien sûr une autre histoire.

Le problème est que cela reste potentiel : la classification n'opère que dans la hiérarchie des classes explicitement introduites par ce que [Nebel, 1990a] reconnaît comme une assertion de second ordre, ce qui n'est pas le cas pour la négation de toute les classes. En fait, les logiques de descriptions pourvues de la négation savent calculer, de façon plus ou moins complète, l'insatisfiabilité, c'est-à-dire la validité de la négation d'une classe, mais c'est une vérification d'instance — il faut fournir la classe à nier — alors que la classification repose sur un parcours systématique des

12. En ce sens, l'interprétation de la classification incertaine de Mariño [1993] est compatible avec l'interprétation définitionnelle qu'elle fait des classes de TROEPS. L'erreur ne réside donc pas là mais dans le fait qu'elle ne se donne pas les moyens d'assurer cette interprétation définitionnelle.

classes explicitement introduites pour appliquer à chacune une vérification d'instance.

**Autres généralisations de la classification**

Si la classification incertaine que nous venons de présenter semble trouver son origine dans SHIRKA, d'autres tentatives ont été faites pour étendre le champ d'application de la classification à des connaissances incertaines. De plus, cette approche incertaine a déjà été appliquée à d'autres problèmes, le filtrage en particulier.

*Classification avec défauts et exceptions*

Malgré l'interdit posé par [Brachman, 1985], [Coupey et Fouqueré, 1994 ; Coupey et Fouqueré, 1997] proposent d'introduire dans les logiques de descriptions, plus particulièrement dans le langage $\mathcal{AL}$, deux constructeurs pour les *défaut* et *exception*, avec le langage $\mathcal{AL}_{\delta\epsilon}^{-}$. Cela leur permet notamment de considérer quatre catégories de relation entre classe et instance : sûre, probable, typique et exceptionnelle, la première n'ayant aucun rapport avec la nôtre. Si cette première approche introduit une non monotonie au niveau extensionnel, elle reste complètement monotone au niveau des classes, malgré ces exceptions. [Padgham et Nebel, 1993 ; Padgham et Zhang, 1993] proposent des extensions non monotones du même genre.

*Classification imprécise et incertaine*

L'introduction de données imprécises et incertaines permet d'envisager une classification pondérée par un ou plusieurs coefficients. C'est ce que font [Rossazza, 1990], [Gonzalez-Gomez, 1996] et quelques systèmes plus ou moins *ad hoc* comme CLASSIC [Granger, 1986 ; Granger, 1988]. Dans tous les cas, le précis étant un cas particulier de l'imprécis, et le certain un cas particulier de l'incertain, cette classification imprécise et incertaine doit se ramener, dans le cas particulier de données précises et certaines, à une sémantique aussi fondée que celle qui a été proposée ici.

*Filtrage incertain*

Le filtrage se ramène, indirectement mais doublement, à la classification. Le filtre peut être considéré comme une classe, qu'il faut d'abord classer automatiquement, et par rapport à laquelle il faut ensuite classer toutes les instances de la description du monde. C'est la démarche suivie par les logiques de descriptions et [Dekker, 1993]. La première étape ne donne *a priori* lieu à aucune incertitude, mais la seconde peut se faire par classification incertaine, non pas centrée sur l'*objet à classer*, mais sur le *filtre*, c'est-à-dire la *classe à instancier*. C'est là qu'il faut parler d'instances valides, satisfiables, insatisfiables, possibles, impossibles, sûres, etc.

Dans un deuxième temps, on constate que la classification de la classe (du filtre) elle-même peut donner lieu à un résultat multi-valué. Par rapport au filtre $\phi$, une classe $\alpha$ peut être :

**valide** – toute instance de $\alpha$ satisfait le filtre : $\alpha \trianglelefteq \phi$ ;
**satisfiable** – certaines instances de $\alpha$ peuvent satisfaire le filtre ;

**insatisfiable** – aucune instance de $\alpha$ ne peut satisfaire le filtre : $\alpha \nabla \phi$.

Les modalités sûre, possible et impossible introduites dans YAFOOL par [Chabre, 1988], inspiré par la lecture de [Pivot et Prokop, 1987], sont les analogues intensionnels (pas forcément équivalents, comme nous l'avons vu) de ces trois modalités extensionnelles. Ce filtrage incertain n'apporte donc pas de notion nouvelle mais c'est une étape obligée pour avoir un filtrage efficace : seules les instances des classes satisfiables non valides, ou plutôt des classes possibles non sûres, sont à comparer effectivement au filtre.

Bien sûr, comme le filtrage constitue la base des requêtes dans les bases de données d'objets (*chapitre 5*), il est tentant de leur appliquer aussi ce filtrage incertain. Le filtrage incertain sur les instances est sans objet, à cause de l'*hypothèse du monde clos* faite par les bases de données. Cependant, comme cette hypothèse n'a pas d'effet sur les comparaisons entre classes, le calcul de classes valides, satisfiables et insatisfiables reste une optimisation intéressante, dont il faut espérer qu'elle déjà prise en compte par les SGBDO existants.

## 12.5 Comparaisons, conclusions et perspectives

La notion de *système classificatoire* introduite par [Euzenat, 1993b] a une acception encore plus générale que la nôtre, puisqu'elle repose sur un *critère de catégorisation*, analogue à notre subsomption intensionnelle, mais abstrait dans la mesure ou aucune intension (et inclusion d'intensions, $\sqsupseteq$) n'est spécifiée. Nous en proposons donc ici un cas particulier où l'intension est formée d'un ensemble d'attributs, ce qui offre un cadre plus précis pour intégrer et comparer les logiques de descriptions et la représentation des connaissances par objets [Carré *et al.*, 1995].

[Hautamäki, 1986] propose une formalisation logique des *points de vue* : il passe en fait par l'intermédiaire d'une structure algébrique qui tient lieu de système classificatoire. Cette structure a la particularité de contenir à la fois les composantes intensionnelle et extensionnelle, d'une façon qui rappelle assez les langages de *prototypes*. En revanche, Hautamäki en reste à un niveau intensionnel, sans interprétation extensionnelle, ce qui converge finalement assez bien avec les conclusions du *chapitre 8* qui soulignent les difficultés d'interprétation des prototypes.

Il existe d'autres sémantiques des systèmes d'objets, en particulier dans le domaine de la programmation logique et des bases de données, qui sont comparables à l'approche suivie par les logiques de descriptions et reproduite ici. Citons par exemple F-LOGIC [Kifer *et al.*, 1995] et LAURE [Caseau, 1991]. La classification ne fait en général pas partie de leurs préoccupations.

L'historique de la classification dans les systèmes d'objets montre une phase informelle, celle des langages de *frames* dont [Fikes et Kehler, 1985] est un parfait exemple, dont les dérives ont ouvert la voie, d'une part à une école formelle, celle des logiques de descriptions, d'autre part à ce qu'il faut bien appeler une « école française » de la classification incertaine, rigoureuse mais peu formelle, présentée dans le *chapitre 10*. L'un de nos objectifs était ainsi d'apporter à la seconde la formalisation de la première. La classification incertaine a-t-elle été représentée par d'autres ? Nous n'en avons pas trouvé trace.

Nous avons montré dans ce chapitre comment la classification incertaine à la SHIRKA se comprend comme une approximation intensionnelle d'une classification fondamentalement extensionnelle. Si l'on veut profiter du caractère informatif et de la sécurité des classes structurellement sûres, et non pas en rester à la seule alternative du possible et de l'impossible, il est nécessaire d'en prouver la correction, dont nous avons montré qu'elle risque de reposer sur la complétude sémantique de la classification extensionnelle. Il importe en particulier d'appliquer ces arguments aux systèmes existants (SHIRKA, TROEPS, FROME) pour en vérifier la correction. Il faut ausi étudier le tropisme classificatoire de ces systèmes et mettre au point des mécanismes, automatiques ou interactifs, analogues à la classification de classes, pour assister la construction de la hiérarchie de classes.

Une autre perspective est d'appliquer cette classification incertaine aux logiques de descriptions, pour les affranchir de la barrière des classes primitives : nous avons vu qu'elle est déjà potentielle pour les langages disposant de la négation comme $\mathcal{ALC}$.

# Quatrième partie

# Applications des objets

# 13

# Objets et musique

CE CHAPITRE DÉCRIT QUELQUES APPLICATIONS typiques de la programmation par objets à la réalisation de systèmes mettant en œuvre des connaissances musicales. On montre comment les mécanismes de base de la programmation par objets permettent de représenter tout un ensemble de concepts de base de la musique tonale. Des mécanismes spécifiques sont ensuite utilisés pour résoudre divers problèmes : règles pour l'analyse, satisfaction de contraintes pour l'harmonisation, propagation de contraintes pour l'édition de partitions. Ces applications musicales posent des problèmes de représentation de connaissances difficiles dans un cadre concret, mais dont la portée dépasse largement le cadre de la musique.

## 13.1 Introduction

Il existe une profonde analogie entre musique et calcul (séquence, répétition, branchement conditionnel). Cette analogie explique sans doute pourquoi l'ordinateur a été utilisé pour « faire de la musique » sous toutes ses formes, que ce soit pour la produire, l'analyser ou la simuler. Cette situation fournit de fait un cadre expérimental dans lequel les représentations de connaissances musicales peuvent être directement opératoires, et donc, dans un certain sens, validées, au contraire d'autres champs artistiques.

Les connaissances mises en œuvre en musique sont complexes : à la fois abstraites (intervalles et accords), définies de manière informelle (par exemple les contours flous des *motifs*) et faisant l'objet de typologies sophistiquées et incomplètes (par exemple celle des formes musicales : du rondo à la sonate). La musique offre donc un terrain d'expérimentation privilégié pour l'Intelligence Artificielle, comme l'ont déjà montré les travaux de Hofstadter sur la réflexivité et la musique [Hofstadter, 1985], de Minsky sur le sens commun musical [Minsky et Laske, 1992], ou de Greussay sur l'analyse [Greussay, 1985] (voir [AFIA, 1995] pour un panorama français sur cette question). Ce lien entre musique et intelligence artificielle est particulièrement présent dans le domaine des objets. C'est sur cet aspect que nous nous concentrons dans ce chapitre.

### 13.1.1 Musique et objets

La musique a été pensée naturellement en termes d'objets depuis ses premières formalisations, ceux de Pierre Schaeffer [Schaeffer, 1966] en étant une incarnation particulièrement radicale et concrète. Il n'est donc pas étonnant que les techniques de la programmation par objets aient été appliquées à de nombreux problèmes musicaux. Bien sûr, il n'y a pas toujours concordance entre les objets pensés par les musiciens et les objets de nos langages, mais ce décalage est moteur et donne lieu bien souvent à des modélisations qui s'avèrent être des sources d'inspiration fructueuses pour d'autres domaines.

En France, le système FORMES [Rodet et Cointe, 1991], un précurseur des systèmes d'aide à la composition par les techniques à objets, était basé sur la notion de *processus*, regroupant un objet au sens classique du terme, et un moniteur pour son ordonnancement dans le temps et la synchronisation de ses sous-objets. Ce système fut utilisé principalement pour des compositions musicales, mais il est général et applicable à d'autres contextes comme l'animation ou la synthèse de la parole. Ralph Johnson, un des auteurs des travaux sur les *frameworks* et les *design patterns* [Gamma *et al.*, 1994] (voir chapitre 4), a travaillé dans le groupe de recherche de Carla Scaletti sur le système à objets *Kyma* [Scaletti, 1987], un des premiers environnements interactifs pour la composition et la synthèse sonore, aujourd'hui commercialisé avec une carte d'acquisition spécialisée pour le traitement du signal temps réel. Les travaux de Steven Pope abordent depuis une dizaine d'années divers problèmes musicaux avec les techniques à objets : éditeurs de partitions, environnements de composition algorithmique en temps réel et outils d'assistance à l'exécution musicale. Ces travaux sont incarnés dans le système MODE [Pope, 1991a]. Plus récemment, Bill Walker a développé un système d'improvisation musicale par objets fondé sur l'idée que l'improvisation est une forme de conversation : le système *ImprovisationBuilder* [Walker *et al.*, 1992]. De manière générale, le lecteur pourra consulter [Pope, 1991b] qui décrit quelques applications importantes de la programmation par objets à la musique, ainsi que le numéro spécial du *Computer Music Journal* sur les objets [Pope, 1989].

### 13.1.2 La musique comme prétexte : le système MUSES

Nous décrivons ici quelques problèmes musicaux typiques et des solutions utilisant les techniques à objets : la musique est dans ce cadre considérée comme prétexte pour poser des problèmes de représentation fondamentaux qui dépassent le cadre de la musique, comme des problèmes de représentation du temps, d'ontologies réutilisables ou d'expression de connaissances. Plus précisément, nous prendrons comme exemples un certain nombre de travaux effectués au LIP6 (Laboratoire d'Informatique de Paris 6), autour du système MUSES, qui sert de base à toute une série d'expérimentations en représentation de connaissances musicales. MUSES se présente sous la forme d'une bibliothèque de classes représentant les concepts fondamentaux de la musique tonale [1] : *pitch-classes*, notes, accords, gammes, mélodies.

1. C'est-à-dire la musique basée sur la tonalité comme la musique classique, le jazz, le rock, etc. par opposition à la musique post-tonale comme le dodécaphonisme ou la musique électro-acoustique.

La taille de cette bibliothèque est d'environ 100 classes et 1500 méthodes [Pachet, 1994a]. Plusieurs applications abordant des problèmes musicaux concrets ont été construites autour de MUSES (voir figure 13.1) : un système d'analyse harmonique automatique [Mouton et Pachet, 1995] [Pachet, 1994c], un système d'harmonisation de chorals à quatre voix [Pachet et Roy, 1995a] [Pachet et Roy, 1995b], un système de simulation d'improvisations [Ramalho et Ganascia, 1994] [Ramalho et Pachet, 1994] et un système d'induction automatique de motifs mélodiques [Rolland et Ganascia, 1996].

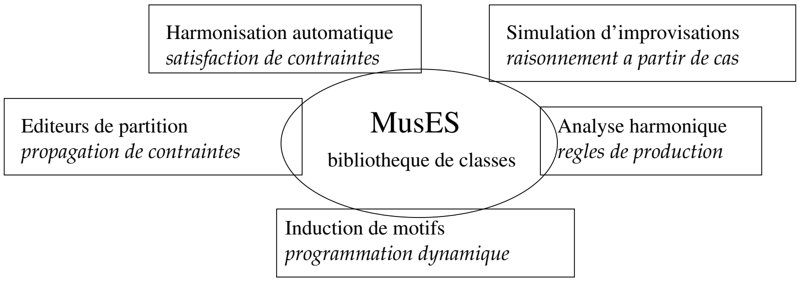



FIG. 13.1: *Le système* MUSES *et ses principales applications.*

Dans ce chapitre, nous décrirons quelques uns des problèmes soulevés par ces systèmes écrits en MUSES : représentation des altérations (dièses et bémols) en utilisant le polymorphisme, représentation de structures temporelles, représentation de règles d'analyse et de contraintes pour l'harmonisation et problèmes d'éditions de partitions.

## 13.2 MUSES : concepts de base et programmation par objets

MUSES est une bibliothèque de classes représentant des connaissances consensuelles sur la musique tonale. Il est bien entendu qu'il n'existe probablement pas de représentation universelle des notions de note, accord, gamme, arpège, etc. Mais il est possible de trouver des représentations raisonnablement réutilisables pour ces concepts de base dans un cadre restreint d'applications musicales. Nous nous sommes essentiellement concentrés sur des applications résolvant des problèmes à caractère analytique, et portant sur des corpus de musique occidentale tonale, à tempérament égal (c'est-à-dire reposant sur le clavier bien tempéré). Dans ce contexte, les premiers concepts à représenter sont ceux de note (indépendante de l'octave ou *pitch-class*) et d'altération (les dièses, bémols et bécarres, signes utilisés pour modifier la hauteur d'une note). Le problème est de représenter non pas tant le signe lui-même, mais bien son intension harmonique. Ainsi, un Ré dièse sonnera-t-il exactement comme un Mi bémol en musique tempérée : c'est la même touche sur le piano. Mais les deux notes portent des intensions musicales bien différentes,

| | |
|---|---|
| `NoteNaturelle<diese`<br>`"rend la variable d'instance diese"`<br>`^diese` | `NoteBemol<diese`<br>`"rend la naturelle correspondante"`<br>`^naturelle` |
| `NoteDiese<diese`<br>`"rend la variable d'instance diese"`<br>`^diese` | `NoteDoubleBemol<diese`<br>`"rend la naturelle bemolisee"`<br>`^naturelle bemol` |

FIG. 13.2: *Les quatre implémentations en* SMALLTALK *de la méthode* `diese`*, représentant une partie de l'algèbre des altérations.*

aussi différentes que le sens des mots « sang », « sans », « s'en », ou « cent »[2]. Ces altérations forment alors une sorte d'algèbre : les dièses et les bémols s'annulent réciproquement. Mais cette algèbre n'est pas simple. Ainsi, une note peut être diésée, voire double-diésée, mais pas plus. Par ailleurs, les notes entretiennent une relation d'équivalence, celle des hauteurs : ré dièse et mi bémol sont des notes distinctes dans la pensée, mais elles *sonnent* pareil, elles représentent la même hauteur.

Une manière particulièrement satisfaisante de représenter ces altérations est d'exploiter le mécanisme de polymorphisme, qui est à la base des langages de programmation par objets. Ceci se fait en considérant une altération comme une méthode et les notes comme des instances de classes différentes, suivant leur altération [Pachet, 1994b]. On définit ainsi six classes de notes : `NoteNaturelle`, ayant 7 instances représentant les notes non altérées, `NoteAlteree`, classe abstraite ayant quatre sous-classes: `NoteDiese`, `NoteBemol`, `NoteDoubleDiese` et `NoteDoubleBemol`, représentant les différents types de notes altérées. L'altération dièse est représentée comme une méthode polymorphe, ayant 4 implémentations différentes (voir figure 13.2). Le type du résultat de cette méthode dépend de la classe dans laquelle elle est implémentée. Ainsi, la méthode `diese` de la classe `NoteNaturelle` rend une instance de `NoteDiese`, calculée statiquement et conservée dans une variable d'instance de même nom ; celle de la classe `NoteDiese` une instance de `NoteDoubleDiese`, aussi conservée dans une variable d'instance. Pour la classe `NoteBemol`, la méthode renvoie la note naturelle correspondante (bémol et dièse s'annulent). La classe `NoteDoubleBemol` renvoie la note naturelle « bémolisée » (double bémol + dièse = bémol). Il faut noter que la méthode `diese` n'est pas implémentée dans la classe `NoteDoubleDiese`. Cette absence provoquera donc une erreur si l'on tente de diéser une note double dièse, ce qui est conforme à notre cahier des charges : les triples dièses n'existent pas ! Le même mécanisme s'applique bien sûr aux bémols.

Enfin, les notes altérées héritent toutes d'une classe abstraite `NoteAlteree`, qui implémente la notion de bécarre (l'anti-altération) par un pointeur vers la note naturelle correspondante (voir figure 13.3).

Une fois la théorie des *pitch-class* et des altérations correctement représentée, l'échafaudage peut être solidement construit. La première notion ajoutée est celle de note effective, dépendante de l'octave (`OctaveDependentNote` : Do6, Do5, etc.). Une solution naturelle serait de considérer les *pitch-class* comme des classes — au sens des langages de classes — et les `OctaveDependentNote` comme des ins-

2. La notion d'altération pose uniquement problème en musique tonale ; la musique post-tonale reléguant l'altération à un signe purement local, dénué de toute intension harmonique.

tances de ces classes. Les différentes classes de *pitch-class* deviendraient alors des métaclasses. Mais ceci nécessite un langage permettant de définir ses propres métaclasses, comme CLOS [Kiczales *et al.*, 1991], ou CLASSTALK [Briot et Cointe, 1989 ; Rivard, 1997]. La solution adoptée en MUSES est l'agrégation : `OctaveDependentNote` est une classe qui agrège une *pitch-class* et un numéro d'octave.

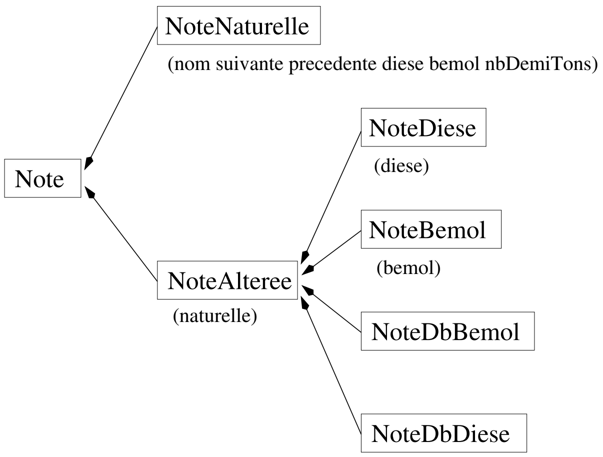



FIG. 13.3: *La hiérarchie des classes de notes et la représentation des altérations en* MUSES. *Les variables d'instance sont entre parenthèses.*

Des notions plus complexes sont ensuite ajoutées, comme les intervalles et leur typologie, qui doivent aussi respecter l'enharmonie (le rapport entre quartes augmentées et quintes diminuées par exemple), les gammes, les accords, etc. Des notions plus abstraites comme celle d'*Analyse* sont aussi représentées. Une fois ces concepts de base correctement définis, des méthodes permettent par exemple de calculer la liste des tonalités possibles pour chaque accord, fournissant ainsi une base solide pour construire des systèmes d'analyse. Ainsi du calcul des tonalités possibles de Do majeur :

```
(Chord fromString: 'C maj') possibleTonalities  -->
([V of F HungarianMinor] [VI of E HungarianMinor]
 [IV of G MelodicMinor] [V of F MelodicMinor] [I of C Major]
 [IV of G Major] [V of F Major] [V of F HarmonicMinor]
 [VI of E HarmonicMinor])
```

## 13.3 Les objets temporels de MUSES

Une fois le cas des objets statiques traité, se pose le problème de la représentation des objets temporels. De nombreux travaux en intelligence artificielle sont consacrés à la représentation du temps. Ils peuvent être regroupés en fonction du type de primitive temporelle sur laquelle ils reposent : intervalle [Allen, 1984], points [McDermott, 1982] ou événements [Kowalski et Sergot, 1986]. Formellement, les trois approches peuvent être reliées les unes aux autres et permettent de représenter les mêmes connaissances [Tsang, 1987]. Cependant, le choix effectué reflète la dimension du temps que l'on considère comme fondamentale : la durée (intervalle) ou l'instant (point, événement). La représentation par intervalle est particulièrement adaptée à l'idée même de note musicale. On peut alors distinguer deux manières principales de représenter les objets temporels, suivant le choix adopté pour représenter ces intervalles : 1) le temps est représenté *en dehors des objets*, dans des collections temporelles, c'est le choix du système MODE, et 2) le temps est *dans les objets* eux-mêmes, c'est le choix adopté pour le système MUSES.

### 13.3.1 Le temps en dehors des objets

Le système MODE [Pope, 1991a] propose une représentation du temps basée sur la notion d'événement (`Event`), classe abstraite représentant des processus temporels. Cette classe possède une variable d'instance `duration`, représentant la durée du processus, dans une unité arbitraire, elle-même réifiée. Par ailleurs, les événements peuvent avoir un certain nombre de propriétés, accessibles par un dictionnaire. Le point important — et astucieux — est que ces événements temporels ne connaissent pas leur temps initial. La justification conceptuelle est qu'un objet temporel n'a de sens qu'au sein d'une collection temporelle déterminée. La durée est ainsi un attribut intrinsèque, alors que le temps de début est vu comme extrinsèque. Ceci permet en pratique de distribuer ces objets dans plusieurs collections temporelles et de faciliter un certain nombre de tâches d'édition (copier/coller). `Event` étant une classe abstraite, une hiérarchie de classes concrètes est construite pour représenter les événements courants comme les notes de musique.

La deuxième notion de base est celle d'`EventList`, qui représente une collection d'événements. Cette collection est constituée de paires (`durée`, `Event`), ou `durée` représente le temps initial de l'événement en question, exprimé comme une durée par rapport au début de la collection temporelle. Cette représentation permet de dissocier le temps initial de l'objet temporel, favorisant ainsi la modularité. En outre, elle permet de représenter, récursivement, les listes temporelles comme des événements, à moindre frais : `EventList` est tout simplement une sous-classe de `Event` ! Ainsi, une `EventList` peut elle-même être considérée comme un objet temporel, à l'intérieur d'une autre collection temporelle, ce qui permet de créer des structures temporelles arborescentes (voir figure 13.4). Noter que ce schéma est une application directe du *design pattern Composite* décrit dans [Gamma *et al.*, 1994].

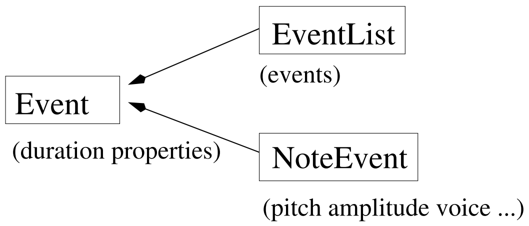



FIG. 13.4: *La hiérarchie des classes d'objets temporels en MODE.*

### 13.3.2 Le temps dans les objets

La deuxième approche, celle suivie dans le système MUSES [Pachet *et al.*, 1996], consiste à créer des structures d'objets temporels explicites. Cette approche est justifiée par le besoin pour certaines applications comme les systèmes d'analyse, de connaître pour chaque note sa position dans une mélodie. Pour ces applications, la représentation précédente est alors inadaptée. Dans cette seconde approche, les objets temporels sont représentés à l'aide de trois notions de base :

- `Lapse`, représentant un intervalle de temps et déterminé par deux variables d'instance ici réconciliées (temps initial, durée),
- `TemporalObject`, représentant un objet temporel, avec une variable d'instance (`lapse`) pointant vers une instance de `Lapse`,
- `TemporalCollection`, représentant une collection d'objets temporels, triés par temps initial croissant.

Comme nous l'avons vu plus haut, MUSES contient des représentations d'un certain nombre de concepts atemporels (pitch-classes, intervalles, accords, etc.). Ces objets pour la plupart ont des correspondants temporels. Une solution pour les représenter serait de définir ces objets temporels en les faisant hériter à la fois de la classe abstraite `TemporalObject` et des classes d'objets intemporels. La solution adoptée en MUSES consiste à utiliser le mécanisme bien connu dans les langages à objets de *délégation* [Lieberman, 1986], ce qui permet, entre autres, d'éviter l'emploi de l'héritage multiple (voir *chapitre 8*). Nous introduisons de plus une notion supplémentaire, permettant de mettre en œuvre cette délégation de manière générique : celle d'« enveloppe temporelle » (`TemporalObjectWrapper`). Cet objet possède une variable d'instance pointant vers un objet non temporel, et lui délègue toutes les opérations non temporelles. Nous faisons la même chose pour les classes de collections (`TemporalCollectionWrapper`). Nous obtenons donc *in fine* un schéma à cinq classes (cf. figure 13.5).

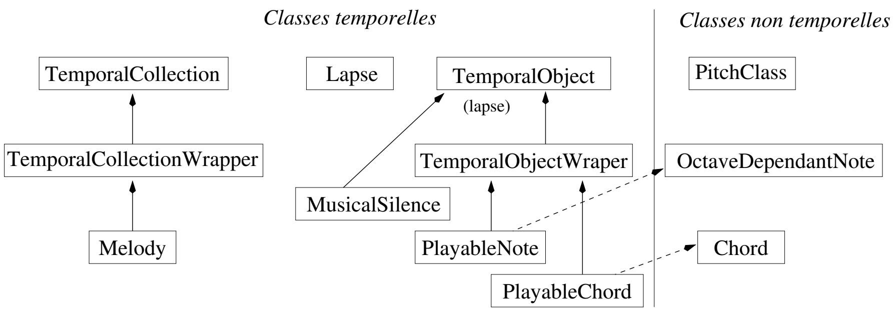



FIG. 13.5: *La hiérarchie des classes temporelles en* MUSES. *Les flèches pleines représentent l'héritage de classe, les traits pointillés la délégation.*

## 13.4 Analyse harmonique de grilles de jazz

Une des activités principales du musicologue est l'*analyse* de pièces musicales, dans le but de mieux comprendre un style, de caractériser un compositeur, ou d'expliciter des structures musicales cachées. Plusieurs théories pour l'analyse ont été développées : stylistique, harmonique, tonale, formelle, Schenkerienne, etc. La plupart de ces théories ont la caractéristique d'être particulièrement bien formalisées, et de nombreuses tentatives d'informatisation de ces techniques ont été proposées [Pachet, 1998]. L'analyse harmonique est un problème particulièrement intéressant en représentation de connaissances car les règles sous-jacentes à cette analyse font l'objet d'un consensus. Nous l'illustrons ici sur l'exemple de l'analyse harmonique de séquences d'accords de jazz.

### 13.4.1 Deux *patterns* de règles pour l'analyse

Étant donné une séquence d'accords, le problème consiste à déterminer la tonalité de chaque accord dans la séquence. Cette tonalité dépend à la fois de la structure de l'accord lui-même (accord majeur, mineur, de septième, etc.), de sa position dans la séquence (temps fort, temps faible) et des accords voisins. Le but de l'analyse est de trouver des tonalités qui soient plausibles musicalement. Cette plausibilité musicale fait appel à des règles harmoniques précises et bien connues qui permettent de décrire des formes musicales de différents niveaux. Par exemple, on peut décrire avec précision une *résolution*, comme un accord de septième suivi d'un accord parfait (du moins dans sa version simple). De même, une structure dite *AABA* peut se décrire par une répétition de quatre formes vérifiant des contraintes d'identité simples. Enfin, une bonne analyse tente de trouver des tonalités qui soient communes au plus grand nombre d'accords possibles (principe de minimisation).

Plusieurs approches ont été explorées pour l'analyse harmonique par ordinateur : procédurales [Ulrich, 1977], à base de grammaires génératives [Steedman, 1984], de grammaires systémiques [Winograd, 1968], de systèmes experts [Maxwell, 1992]. Leur principal inconvénient est de représenter toutes les connaissances mises en jeu à l'aide d'un seul paradigme. Comme toujours, ce paradigme, quel qu'il soit, est souvent bien adapté à un type de connaissances, mais pas à d'autres. On aboutit ainsi à des systèmes complexes, difficiles à comprendre, et limités dans leurs performances. L'approche que nous avons suivie consiste à trouver un modèle de l'analyse qui exploite les structures objets existantes et qui soit généralisable à d'autres problèmes d'analyse, pas forcément musicaux. Cette approche est fondée sur un modèle du processus d'analyse qui relève de la psychologie de la forme (*gestalt*) : on identifie dans un premier temps des formes musicales classiques, comme les II-V, les anatoles, cadences, résolutions ou autres marches harmoniques. Par exemple, un II-V sera reconnu lorsqu'on trouve deux accords voisins tels que la tonique du second soit la quinte de la tonique du premier, que le premier est mineur, le deuxième de septième, etc. (par exemple, la séquence D min 7 / G 7). Dans un deuxième temps, on tente de regrouper les accords isolés à l'aide de règles de regroupement. Par exemple, une règle dit que si un accord isolé est voisin d'une forme analysée et que la tonalité de cette forme analysée fait partie des tonalités possibles de cet accord, alors on peut analyser l'accord dans cette tonalité, et donc grouper l'accord et la forme pour créer une forme englobant les deux (par exemple, un accord isolé de C majeur, suivi d'une longue séquence d'accords analysée en C majeur).

Dans les deux cas, les règles spécifient des manières d'agréger des formes d'ordre $n$ pour créer des formes d'ordre $n + 1$. L'analyse est terminée lorsqu'une forme couvrant l'ensemble de la séquence a été trouvée [Mouton et Pachet, 1995][Pachet, 1994c]. Pour implémenter ces règles, le formalisme des règles de production est particulièrement bien adapté. Nous avons montré [Pachet, 1998] que les règles d'agrégation, que ce soit pour les formes simples (exemple de la règle de II-V) ou pour les règles de regroupement générales (accord isolé suivi de forme analysée), suivent toutes le même schéma :

```
 Soient a1 a2 des instances (d'une sous-classe) de FormeMusicale
SI certaines propriétés sur l'agencement de a1 et a2 sont vraies,
ET certaines propriétés sur leurs caractéristiques locales sont vraies
ALORS créer une instance d'une sous-classe de FormeMusicale
                         agrégeant les deux formes
```

En outre, un des points importants de cette approche, notamment par rapport aux modèles à base de grammaire, est de permettre l'écriture de règles de destruction de formes, appelées *règles d'oubli*, qui permettent d'assurer une convergence rapide du raisonnement, en limitant dynamiquement le nombre de formes créées. Les problémes de convergence (risque de recréer des formes détruites, et donc de boucler) sont résolus en adoptant des stratégies particulières de contrôle, à base d'agendas [Dojat et Sayettat, 1994]. Ces règles suivent, elles, le schéma suivant :

```
 Soient a1 a2 des instances de (une sous-classe de) FormeMusicale
SI certaines propriétés sur l'agencement de a1 et a2 sont vraies,
ET certaines propriétés sur leurs caractéristiques locales sont vraies
ALORS supprimer a2
```

### 13.4.2 Discussion : Patterns et Frameworks

Le modèle élaboré pour rendre compte du processus d'analyse, outre ses intérêts purement musicologiques, présente plusieurs intérêts pour la programmation par objets, et particulièrement pour l'étude des *patterns* et des *frameworks*. D'une part nous avons mis en évidence deux *patterns* de règles, au sens de [Gamma *et al.*, 1994]. Ces deux *patterns* permettent à eux seuls de représenter la totalité des bases de règles nécessaires au raisonnement. Par ailleurs, une comparaison de ce système avec un système développé dans le domaine médical — le système NéoGanesh [Dojat et Sayettat, 1994] — a permis de mettre en évidence une forte analogie entre les deux types de raisonnements (médical et musical), conduisant ainsi à un *framework* permettant la représentation de raisonnements temporels hiérarchiques [Pachet et Dojat, 1995]. Enfin, la mise en évidence d'un problème de circularité dans les grilles de jazz (la fin de la grille précède en fait le début), a été le point de départ d'une recherche plus formelle sur une extension des intervalles de Allen aux intervalles circulaires [Pachet et Carrive, 1996].

## 13.5 Harmonisation automatique

Le problème de l'harmonisation automatique (le plus souvent de chorals à quatre voix comme en a composé Bach), est d'une certaine manière le problème dual de celui de l'analyse. Il consiste, pour une mélodie donnée, à écrire trois autres voix, de manière à respecter les règles de l'harmonie et du contrepoint. Les règles sont du style : « l'intervalle de quarte augmentée est interdit entre deux notes consécutives », ou « la sensible doit monter à la tonique », ou encore « les intervalles de quintes entre deux voix ne doivent pas être parallèles ».

Ce problème a déjà été abordé par diverses techniques, dont celles de la satisfaction de contraintes (CSP, voir *chapitre 9*) : [Ovans et Davison, 1992] par la CSP pure, [Ebcioglu, 1992] utilisant les algorithmes de retour-arrière intelligent, [Tsang et Aitken, 1991] par la programmation logique avec contraintes, [Ballesta, 1994] avec une combinaison de contraintes et d'objets. Aucune de ces approches n'a permis de construire de systèmes aux performances réalistes (une minute environ pour le plus rapide d'entre eux, sur des mélodies d'une dizaine de notes, alors qu'un bon musicien résout ces problèmes en un temps quasiment instantané !).

Dans notre contexte, ce travail constitue un remarquable terrain d'investigation pour les systèmes intégrant satisfaction de contraintes et objets (voir à ce sujet *chapitre 9*). L'exploitation des structures objets complexes comme celles de MUSES permet en effet de poser le problème de manière plus claire et de le résoudre de manière plus satisfaisante. Les connaissances liées aux objets de base (notes, accords, intervalles) sont représentées de manière procédurale avec la bibliothèque MUSES. Les lois des traités d'harmonie et de contrepoint sont exprimées comme des contraintes portant sur ces objets, et sont gérées par un résolveur de contraintes général. On peut montrer alors que l'exploitation des structures objets dans la résolution du problème permet non seulement d'augmenter la lisibilité des contraintes, mais aussi de réduire considérablement leur complexité et donc le temps d'exécution [Pachet et Roy, 1995a] [Pachet et Roy, 1995b]. Un des points-clé de cette approche est de remplacer des contraintes d'arité élevée portant sur des objets atomiques (des valeurs), par des contraintes d'arité faible, portant sur des objets complexes. Typiquement, la fameuse règle interdisant les « quintes parallèles » entre accords consécutifs s'exprime dans l'approche standard par une contrainte (complexe) d'arité huit (quatre notes par accord, chaque note étant considérée ici comme un objet atomique, par exemple une hauteur), et dans la nôtre par une contrainte d'arité deux (deux objets accords). Le système ainsi obtenu est non seulement plus lisible et modifiable, mais aussi environ 10 fois plus rapide que tous les autres.

## 13.6 Éditeurs graphiques

La conception d'éditeurs graphiques constitue un terrain privilégié de la programmation par objets. Le problème de l'édition de partitions est un cas particulièrement difficile. Plusieurs types de logiciels sont proposés sur le marché, répondant à divers besoins, souvent inconciliables. D'une part des logiciels permettent de faire de l'édition de partitions à proprement parler. Ceux-ci offrent à l'utilisateur toute la palette des signes graphiques musicaux, avec laquelle celui-ci peut composer sa partition comme un dessin. D'autres logiciels permettent une édition graphique plus limitée mais interactive, obtenue à partir d'une représentation sémantique riche des notes, comme les logiciels de *sequencers* par exemple. On retrouve alors le même dilemme que pour les traitements de texte : interactivité et automatismes, contre *batch* et flexibilité (par exemple Word contre LaTeX).

La complexité des objets mis en jeu dans ces éditeurs peut être illustrée par un exemple typique : l'affichage des notes et la gestion des syncopes. La représentation graphique d'une note dépend à la fois de sa durée (il y a des blanches, des noires, des croches, etc.) mais aussi de son emplacement, de sa position dans la mesure. Ainsi, une note durant 4 temps, dans une mesure à 4 temps et commençant sur le premier temps doit s'afficher par une *ronde*, graphiquement représentée par un rond sans queue, illustrée figure 13.6 (à gauche)[3]. Lorsque cette même note commence sur le deuxième temps, on ne peut plus l'afficher comme telle, pour deux raisons : d'abord la mesure ne peut contenir que 4 temps, et donc la note « dépasse », et ensuite la règle de syncope stipule qu'une note ne peut franchir un temps plus fort que son temps de départ. Ainsi, la notation correcte est celle d'une noire liée à une blanche (syncope au sein de la mesure), liée encore à une noire à la mesure suivante (cf. figure 13.6, milieu). Si cette même note commence sur le deuxième quart du premier temps, les mêmes contraintes conduisent à la représentation encore plus complexe de la figure 13.6 (à droite). Ces trois représentations doivent être recalculées rapidement, car les notes peuvent être déplacées à la souris. Le problème est encore plus complexe quand les mesures sont de taille variable.


FIG. 13.6: *Trois représentations graphiques pour la même note dans l'éditeur* MUSES.

Trouver une conception par objets d'un tel éditeur qui permette d'assurer un bon compromis entre interactivité et calculs automatiques est un problème ouvert. De nombreuses recherches exploitent les formalismes de *propagation de contraintes* pour proposer des *frameworks* dans lesquels de tels éditeurs sont, en théorie, plus faciles à construire [Vlissides et Linton, 1990] [Brant, 1995] (voir aussi *chapitre 9*). L'application de ces techniques à des éditeurs de partitions n'est pourtant pas facile. Comme nous l'avons vu, la représentation graphique d'une même note peut donner lieu à la création d'un ou de plusieurs objets graphiques, et ce, dynamiquement. Or les formalismes de contraintes ne s'accommodent pas très bien de ces objets « fantômes » ! De plus, les problèmes à résoudre pour atteindre des performances raisonnables nécessitent de représenter des connaissances typographiques complexes : taille et inclinaison des hampes, taille des mesures, affichage des polyphonies, gestion des multiples portées simultanées, problèmes de justification et de rupture de page, etc. Enfin, les partitions réalistes comportent plusieurs centaines

3. Il s'agit plus précisément d'une ellipse légèrement inclinée, dont le trait est plus plein sur les bords que sur les sommets.

de notes, ce qui exclut en pratique l'utilisation de ces techniques. Ici encore, si la structuration par objets permet de réduire considérablement la complexité, celle-ci n'est pas pour autant totalement vaincue et de nouveaux paradigmes de programmation ou de représentation de connaissances doivent être identifiés pour permettre la construction aisée de tels outils.

## 13.7 Autres applications

Bien d'autres activités musicales peuvent ainsi être modélisées par des objets et la place manque pour les décrire toutes. Voici quelques unes d'entre eux, le lecteur est invité à consulter les références pour plus de détails.

### 13.7.1 Classification de sons et programmation de synthétiseurs

Ce problème concerne la spécification de sons pour les synthétiseurs du commerce. Les synthétiseurs commercialisés aujourd'hui proposent des modes de synthèse sonore extrêmement sophistiqués (par modulation de fréquence, soustractive, par modèles physiques), que les musiciens ne maîtrisent généralement pas, ce qui rend leur programmation quasiment impossible. Or les experts en programmation de synthétiseurs utilisent des connaissances souvent explicites, de surface, c'est-à-dire relativement indépendantes des modèles de synthèse effectivement employés par l'appareil. Un exemple typique de transformation de son est « pour rendre un son plus gros, on peut le doubler par une copie conforme et désaccorder très légèrement sa copie ». Reconnaissant que la plupart des règles expertes concernent les transformations applicables aux sons, plutôt que la fabrication de sons à partir de zéro, l'étude décrite dans [Rolland et Pachet, 1996] a conduit à construire une taxonomie de sons fondée sur les transformations applicables. L'idée est alors de classer un son de départ dans cette taxonomie, afin d'en déduire une liste de transformations, associée à chaque type de son trouvé par le classifieur. Cette liste est ensuite proposée à l'utilisateur, qui en sélectionne une. Puis la transformation est appliquée, conduisant alors à un nouveau son, qui est à son tour classé, puis le cycle recommence.

Du point de vue des représentations mises en jeu, ce système présente l'originalité d'être bicéphale : la partie classificatoire est réalisée dans le formalisme des logiques de descriptions (le système BACK, voir *chapitre 11*). Le résultat de cette classification est une liste de types (les classes de son, par exemple `BrassyAble`, `DecayAble`), auxquels sont associés des *noms* de transformations. Le nom de transformation une fois choisi par l'utilisateur, la transformation elle-même est effectuée par une méthode SMALLTALK ayant ce nom comme sélecteur. Les deux parties coopèrent pour réaliser le cycle de base, avec choix de la transformation par l'utilisateur à chaque cycle (voir figure 13.7).

### 13.7.2 Simulation d'improvisation

L'improvisation est une des activités humaines les plus fascinantes qui soit. Des travaux en cours tentent de modéliser cette activité, en l'envisageant comme

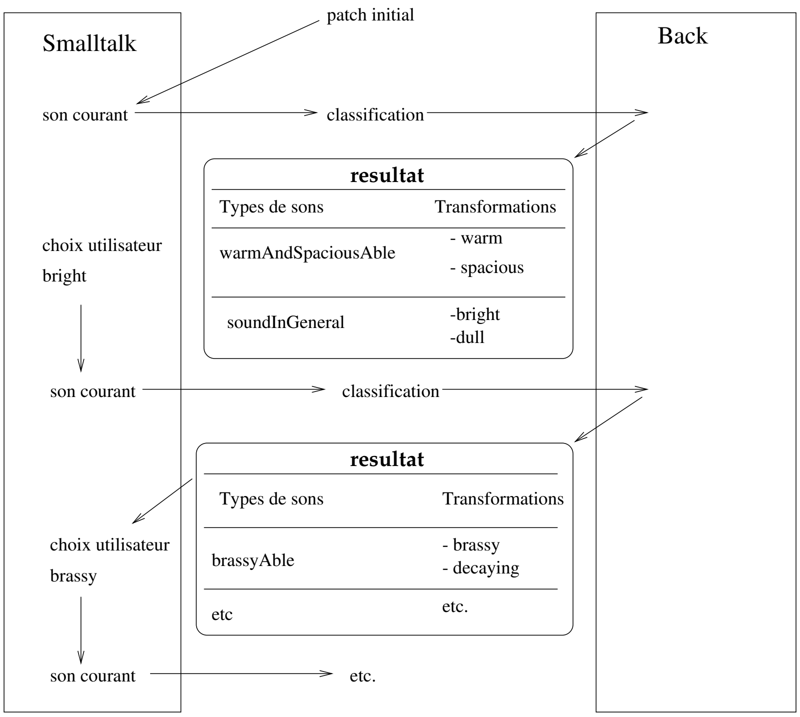



FIG. 13.7: *Le schéma de raisonnement pour la classification de patches de synthétiseurs. La partie classification est réalisée en* BACK, *l'application de la transformation sélectionnée en* SMALLTALK

une forme particulière de résolution de problèmes faisant intervenir une boucle de contrôle dirigée par le « temps restant disponible », et exploitant une représentation des actions musicales par objets [Ramalho et Pachet, 1994], ainsi qu'une mémoire musicale sous forme de cas [Ramalho et Ganascia, 1994]. Ces actions musicales et les cas sont représentés par des objets temporels particuliers et utilisent la représentation des objets temporels vue à la section 3. Ce modèle, qui donne lieu à une mise en œuvre aisée, permet d'étudier précisément ce qui, dans l'activité d'improvisation, relève de la simple juxtaposition de *patterns*, de ce qui relèverait d'une forme de « créativité ».

Un autre travail en cours dans le même domaine concerne le problème de la caractérisation des styles musicaux. Les techniques de programmation dynamique sont utilisées pour détecter des motifs récurrents, dans de larges corpus de mélodies : les improvisations de Charlie Parker [Rolland et Ganascia, 1996], et constituent un autre exemple d'utilisation de la bibliothèque de classes MUSES.

## 13.8 Conclusion

Nous avons montré quelques exemples typiques de problèmes musicaux nécessitant de représenter divers types de connaissances musicales, pour lesquelles la programmation par objets apporte des réponses intéressantes. Les mécanismes de la programmation par objets permettent ainsi de représenter une couche de base substantielle qui sert de point de départ pour l'élaboration de diverses applications. Mais ces mécanismes ne suffisent pas à exprimer bon nombre de connaissances complexes : on fait alors appel à des ontologies spécialisées, aux contraintes, aux règles, à la classification, à la programmation dynamique, etc. pour enrichir le pouvoir d'expression des objets de base, voire construire de nouveaux langages. Certains problèmes restent cependant difficiles à résoudre et mettent en évidence des complexités non encore maîtrisées. Ces problèmes musicaux soulèvent, comme nous l'avons vu, des questions fondamentales de représentation de connaissances, qui mettent à l' épreuve de manière fructueuse les paradigmes actuels de programmation et de représentation.

# 14

# ReSyn : objets, classification et raisonnement distribué en chimie organique

Ce chapitre présente une application, le système ReSyn [Vismara *et al.*, 1992], dans laquelle la représentation par objets et le raisonnement par classification jouent un rôle central. ReSyn est un système d'aide à la conception de plans de synthèse en chimie organique qui est développé dans le cadre du GDR Ticco (Traitement Informatique de la Connaissance en Chimie Organique). Ce GDR [1] interdisciplinaire, dans lequel informaticiens et chimistes collaborent, regroupe une dizaine de laboratoires de recherche du secteur public et différentes sociétés industrielles : Framentec Cognitech, Roussel Uclaf, Sanofi Chimie, Sema Group et Servier.

Si la synthèse de molécules organiques, ayant des propriétés particulières et pouvant être commercialisées, représente un enjeu considérable pour l'industrie chimique, elle pose aussi des problèmes très complexes dont la résolution fait appel à des connaissances chimiques nombreuses, diverses et souvent peu explicites. Depuis une trentaine d'années, beaucoup de travaux, dus essentiellement à des chimistes, ont été menés pour concevoir des systèmes informatiques susceptibles d'aider à la résolution de ces problèmes [Ott et Noordik, 1992]. Bien que certains d'entre eux fassent preuve d'une certaine efficacité, les systèmes réalisés à ce jour n'ont guère dépassé le stade du prototype et, construits à l'aide de techniques informatiques classiques, souvent procédurales, ils suscitent la plupart des critiques faites habituellement aux systèmes experts.

Malgré les acquis, beaucoup reste encore à faire, tant pour mieux comprendre et formaliser la démarche du chimiste dans la résolution des problèmes de synthèse que pour simuler cette démarche sur un ordinateur, en employant les techniques informatiques adéquates. Ceci a guidé notre travail et nous avons entrepris la réalisation

1. Groupe de Recherche du CNRS.

de RESYN, en ayant pour priorité la construction de modèles facilitant l'acquisition des connaissances et garantissant l'évolution du système.

Les connaissances du domaine ont été modélisées selon une approche *orientée objets*, avec l'aide d'experts de la synthèse organique et en s'appuyant sur des résultats antérieurs. La plupart des concepts que manipule RESYN sont décrits en termes de graphes. Ceux-ci sont organisés selon les relations de *subsomption* et la principale méthode de résolution dont dispose RESYN est le *raisonnement par classification* qu'il applique sur ces hiérarchies de graphes. La notion classique de subsomption entre graphes étant insuffisante, nous avons introduit celle de subsomption entre *appariements de graphes*, ce qui nous a permis de développer une méthode originale : la *classification par appariements dirigés*.

Les stratégies employées par les chimistes dans la résolution des problèmes de synthèse forment, pour leur part, un ensemble de connaissances peu explicites que nous nous efforçons de modéliser. Dans RESYN, l'élaboration de directives stratégiques met en œuvre différents points de vue pouvant converger ou se concurrencer. Cette approche est basée sur l'utilisation d'une *représentation distribuée du raisonnement*.

RESYN est développé en YAFOOL [Ducournau, 1991], langage dans lequel les objets sont représentés à l'aide du concept de *frame*.

## 14.1 Le problème de la synthèse en chimie organique

Un problème de synthèse chimique se pose lorsqu'on veut préparer une molécule donnée par des moyens artificiels. Une solution à un tel problème est un chemin de synthèse. Celui-ci doit permettre de passer d'un état initial, constitué par un ensemble de produits de départ, à un état final, contenant la molécule souhaitée dite molécule cible, les opérateurs de changement d'état étant des réactions chimiques. Un chemin de synthèse comporte généralement plusieurs étapes, leur nombre pouvant atteindre plusieurs dizaines dans le cas de molécules très complexes. La résolution se déroule en deux phases successives, l'une pour établir un plan de synthèse (cf. figure 14.1), l'autre pour expérimenter ce plan.

La conception d'un plan de synthèse repose sur une perception de la molécule cible qui s'effectue à partir de la formule structurale de celle-ci. Une molécule étant un objet tridimensionnel composé d'atomes joints par des liaisons, sa formule structurale en est une représentation graphique qui rend compte de sa composition (nombre et type des atomes), de sa constitution (relations de voisinage entre les atomes) et de sa stéréochimie (arrangement relatif des atomes dans l'espace 3D). Par l'analyse de cette information, on reconnaît des sous-structures caractéristiques, telles que des systèmes de cycles, des groupements fonctionnels, des stéréocentres, etc., permettant d'identifier la molécule cible comme un membre de différentes familles chimiques qui possèdent des propriétés particulières ou des modes d'obtention connus. D'autres informations utiles peuvent également être relevées, comme la position relative des sous-structures ou leur place dans la structure cible, les ressemblances avec d'autres molécules, les symétries, etc. Cette perception conduit à une représentation détaillée du problème posé.

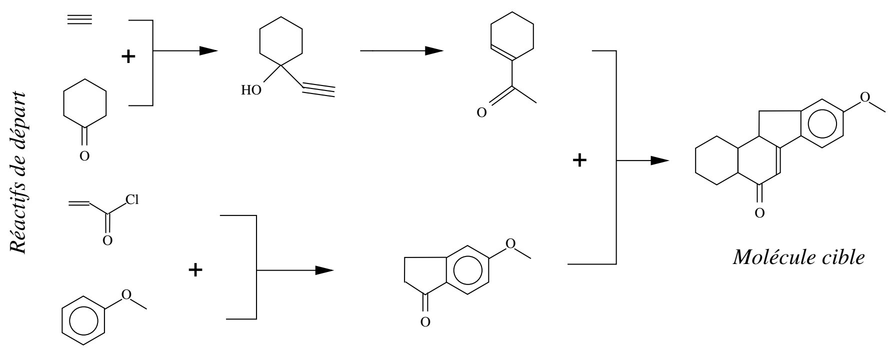



FIG. 14.1: *Exemple de plan de synthèse (les conditions de réaction ne sont pas indiquées).*

Très généralement, dans un tel problème, c'est l'état final contenant la molécule cible qui est à la fois le mieux défini et, chimiquement, le plus complexe. Aussi, la démarche la plus formelle qui ait été proposée jusqu'à présent pour concevoir un plan de synthèse est une approche analytique dans laquelle on recherche, partant de la structure cible et par étapes successives, des précurseurs possibles de celle-ci jusqu'à trouver un ensemble convenable de produits de départ. Pour mener à bien cette rétrosynthèse [Corey et Cheng, 1989], on utilise les opérateurs inverses des réactions chimiques, nommés *transformations*. Appliquée à la représentation d'une molécule, une transformation engendre la représentation de précurseurs de cette molécule. En principe, un précurseur possède une structure plus simple que celle de la molécule d'origine. Lorsqu'un précurseur n'est pas reconnu comme un produit de départ, c'est-à-dire une molécule connue, disponible commercialement ou facile à obtenir, il peut être considéré lui-même comme une nouvelle cible.

À la fin de ce processus, on aura construit un arbre de rétrosynthèse dont la racine est la cible et les feuilles sont des produits de départ. Cet arbre peut contenir un nombre considérable de plans de synthèse s'il est développé systématiquement car des millions de molécules et de réactions sont actuellement connues et sont donc des produits de départ et des opérateurs potentiels. Pour réduire l'espace du problème et contrôler la construction de l'arbre de rétrosynthèse, le chimiste fait appel à des heuristiques appelées stratégies de synthèse. Le plan à expérimenter sera choisi parmi d'autres selon des critères de faisabilité, d'originalité et de coût.

## 14.2 Le système RESYN

RESYN est un système d'aide à l'élaboration de plans de synthèse. Son objectif est de construire un arbre de rétrosynthèse pour une molécule qui lui est soumise. L'utilisateur propose la molécule cible au moyen d'un éditeur permettant de dessiner sa formule structurale, laquelle est interprétée comme un graphe par RESYN. Dans un graphe moléculaire, les sommets et les arêtes figurent respectivement les

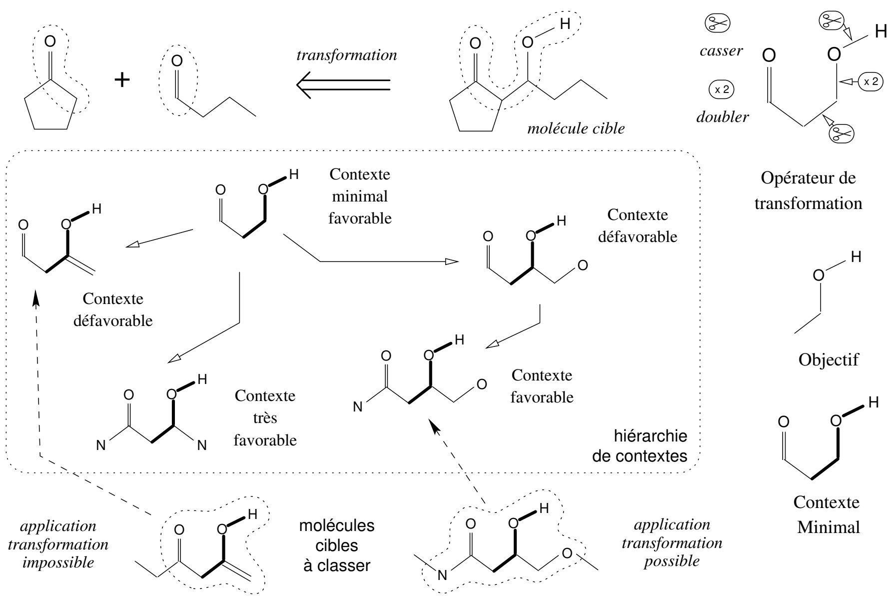



FIG. 14.2: *Transformation, objectif et hiérarchie de contextes*

atomes et les liaisons. Ce graphe est analysé par RESYN afin d'en percevoir les caractéristiques et déterminer des objectifs stratégiques.

Une *transformation* est un opérateur qui modifie un ensemble de sommets et d'arêtes du graphe moléculaire (cf. figure 14.2, en haut). Le graphe induit par cet ensemble constitue l'*objectif* de la transformation (fig. 14.2, à droite). L'application d'une transformation obéit à des contraintes d'ordre chimique qui définissent les environnements favorables ou défavorables à l'utilisation de la transformation. Nous décrivons ces environnements sous la forme de graphes appelés *contextes* qui étendent l'objectif de la transformation aux sommets et arêtes conditionnant son application. Le *contexte minimal* (fig. 14.2, à droite) est le plus général (c.-à-d. le plus simple) de ces contextes.

L'ensembles des contextes, qui peut être très vaste, est représenté sous la forme d'un réseau organisé suivant une relation de subsomption basée sur la notion de sous-graphe partiel (fig. 14.2, au centre). Déterminer si l'on peut appliquer une transformation à la molécule cible correspond à une classification du graphe moléculaire dans la hiérarchie de graphes définie par ce réseau (fig. 14.2, en bas). Si le contexte le plus spécifique contenu dans la molécule est favorable (resp. défavorable) l'application de la transformation sera possible (resp. impossible). Nous présenterons les techniques associées à ce raisonnement par classification dans le paragraphe 14.4.

La figure 14.3 présente un schéma général du fonctionnement actuel du système RESYN. La première étape consiste à « percevoir » les propriétés de la molécule

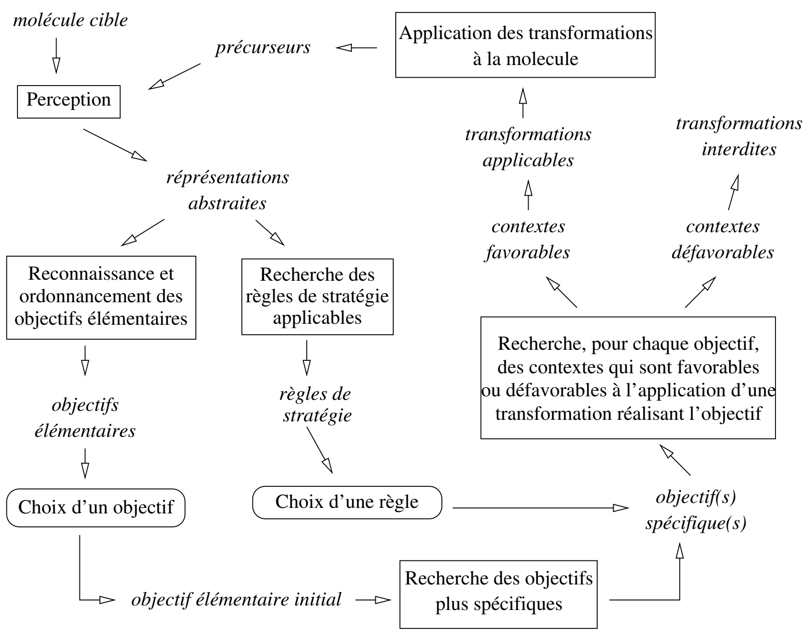



FIG. 14.3: *Schéma général du fonctionnement de* RESYN

cible. Le système est alors capable de déterminer deux types d'*objectifs stratégiques* rudimentaires qui sont proposés à l'utilisateur :

- les *objectifs élémentaires* décrivent une action à appliquer sur une seule liaison de la molécule (par exemple « casser cette liaison »). Lorsqu'un tel objectif est choisi par l'utilisateur, le système essaie d'étendre cet objectif élémentaire de façon à obtenir des objectifs de transformation plus importants. Ces objectifs, dits *spécifiques*, permettent de modifier d'autres liaisons situées autour de celle qui est décrite dans l'objectif élémentaire ;
- les *règles de stratégies* permettent de définir directement un *objectif spécifique*. Elles correspondent généralement à des transformations chimiques importantes dont l'application est systématiquement recherchée.

Pour chacun des *objectifs spécifiques* obtenus, suivant le choix de l'utilisateur, RESYN recherche les contextes les plus spécifiques qui intègrent cet objectif. Ces contextes peuvent être favorables ou défavorables pour l'application de transformations réalisant l'objectif choisi. En effet, il peut exister différentes transformations chimiques qui réalisent le même objectif (en utilisant, par exemple, des catalyseurs différents). De ce fait, un même contexte peut apporter des informations sur l'application de diverses transformations. Afin de factoriser la connaissance, nous avons choisi de regrouper tous les contextes associés à ces transformation dans une seule hiérarchie.

L'étude des transformations réalisant un objectif donné se ramène donc à la recherche des contextes les plus spécifiques parmi ceux qui contiennent l'objectif et sont contenus par la molécule cible. Pour chacun des ces contextes, il suffit de déterminer quelles sont les transformations pour lesquelles le contexte est favorable ou défavorable. Nous verrons dans le paragraphe 14.3.3 que cette détermination fait appel à la notion d'*héritage cumulatif non monotone*.

Les transformations dont le contexte associé est défavorable sont rejetées et le système peut utiliser ce contexte pour justifier sa décision. Si le contexte est favorable, la transformation est appliquée au graphe moléculaire cible, afin d'engendrer un ou plusieurs précurseurs. Le processus général peut alors être réitéré à partir de ces précurseurs. Une base de données de molécules disponibles permet de déterminer si un précurseur constitue un *produit de départ*, c'est-à-dire une feuille de l'arbre de rétrosynthèse.

Toutes les étapes de cette recherche sont regroupées au sein d'un arbre comme celui présenté dans la figure 14.4. La molécule cible initiale constitue la racine de cet arbre et les autres nœuds décrivent les diverses voies de synthèse qui ont été envisagées. L'utilisateur peut accéder à toutes ces informations et choisir le chemin de synthèse dont il désire poursuivre l'étude.

Actuellement, les règles de stratégie sont peu nombreuses et se résument à quelques heuristiques telles que la recherche systématique des objectifs des transformations chimiques les plus importantes (Diels-Alder, Claisen, ...). Les objectifs définis par RESYN sont donc assez élémentaires et souvent de l'ordre de la liaison. Une part de travail consiste à doter RESYN des outils lui permettant d'acquérir ces règles de stratégie. L'intérêt de ce travail est double. D'une part, chercher à construire un modèle du raisonnement de l'expert, c'est le comprendre mieux. D'autre part, définir des objectifs stratégiques dont les graphes sont plus importants que ceux que RESYN manipule à l'heure actuelle, c'est définir des points d'entrée plus spécifiques dans le réseau des transformations et des contextes, et accélérer ainsi le mécanisme de recherche des transformations applicables. Cette démarche sera décrite dans le paragraphe 14.5.

## 14.3 Représentations à objets dans RESYN

Le système RESYN a été écrit en YAFOOL [Ducournau, 1991], un langage intégrant les notions de *frames* et de classes, permettant aussi bien une programmation par objets que par prototypes (voir *chapitre 8*). YAFOOL a été développé par R. Ducournau, avec la collaboration initiale de J. Quinqueton. YAFOOL est écrit en LELISP [Chailloux *et al.*, 1986]. Il intègre une interface graphique YAFEN ainsi qu'un « moteur d'inférence » YAFLOG. L'ensemble est désigné sous le terme d'Y3.

Pour concevoir un système tel que RESYN, on est principalement confronté à un problème de modélisation, aussi bien au niveau des structures que des méthodes. Le recours à un langage à objets comme Y3 s'est avéré bien adapté aux caractéristiques de la connaissance manipulée dans un système d'aide en synthèse organique :

- la connaissance est parcellaire : la confrontation d'expertises multiples ne facilite pas la construction d'une vision globale du problème. Elle encourage au contraire la définition de modèles décrivant une aire d'expertise réduite.

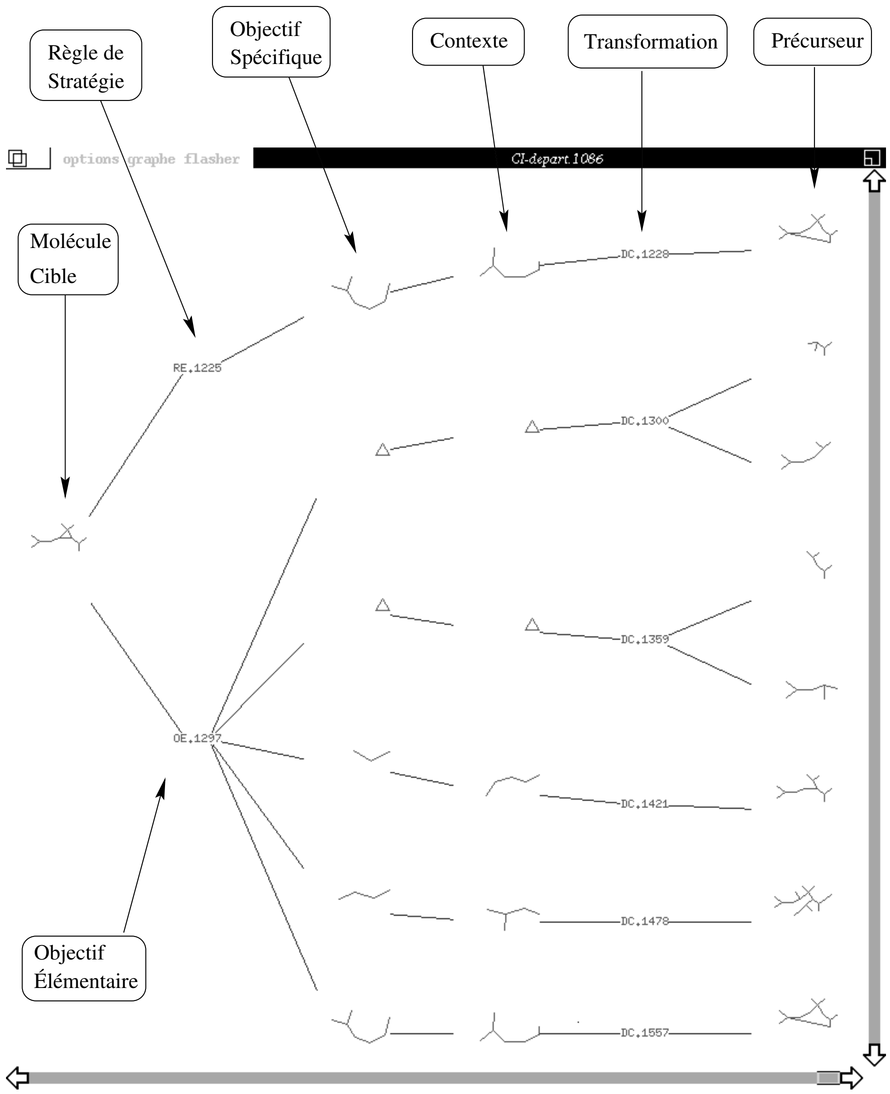



FIG. 14.4: *Extrait d'un arbre de rétrosynthèse construit par* RESYN.

La modularité propre aux langages à objets s'avère ainsi bien adaptée à la représentation de la connaissance. La notion de facette associée aux attributs, et plus particulièrement de réflexe, procure un grand pouvoir d'expression des caractéristiques des concepts et des contraintes qui leurs sont associés ;

- la connaissance est évolutive. Le domaine de la synthèse étant partiellement formalisé, le travail de modélisation n'est jamais parfaitement terminé. RESYN est un prototype qui doit permettre de tester la validité et la pertinence de modèles de connaissance ou de méthodes d'analyse et de résolution. Le principe d'« encapsulation » associé à la programmation par objets facilite l'évolution de la connaissance. Les objets étant relativement autonomes, il est possible de modifier un modèle en limitant les risques d'interaction avec les autres concepts décrits dans le système. L'interaction entre objets est en effet fortement contrainte par la définition de points d'entrées obligatoires au niveau des objets (méthodes ou attributs) ;

- la connaissance peut être hiérarchisée. Le mécanisme d'héritage autorise l'organisation des concepts suivant divers niveaux d'abstraction. Cette représentation hiérarchique de la connaissance simplifie la définition de nouveaux concepts dont la description peut s'appuyer sur des modèles déjà existants qu'il suffit de spécialiser ;

- un système d'aide à la synthèse doit permettre l'accès aux connaissances qu'il manipule. Leur lisibilité est facilitée par la modularité inhérente aux langages à objets. La présence d'interfaces dédiées à la visualisation des connaissances est généralement commune à tous les langages à objets. Dans Y3, l'existence d'un « browser »[2] évolué s'est avérée très utile lors de la conception de RESYN. Par ailleurs, dans la mesure où YAFEN est lui-même défini par un ensemble d'objets, l'élaboration de nouvelles interfaces consiste généralement à spécialiser des objets graphiques préexistants.

### 14.3.1 Quelques notions de base en Y3

Tout objet en Y3 est basé sur le concept de *frame*. Un *frame* comporte un certain nombre d'*attributs*, eux-mêmes divisés en *facettes* qui peuvent être déclaratives ou procédurales.

Une facette déclarative contient des informations définissant l'attribut. La facette `value` est la plus courante car elle décrit la valeur de l'attribut. Mais il existe d'autres facettes déclaratives comme les facettes de typage (`un` ou `des` pour un attribut à valeur unique ou multiple). Par exemple, un objet `ensemble-d'entiers` possédera deux attributs : `les-élements` et `cardinal` ayant respectivement pour facette de typage « `des` = entiers » et « `un` = entier ».

2. Le terme de « browser » (de l'anglais *to browse = feuilleter*) désigne l'interface graphique qui permet de visualiser un objet et de se déplacer dans la hiérarchie d'héritage ou suivant les relations qui existent entre les objets.

Les facettes procédurales décrivent des procédures qui sont déclenchées « par réflexe » lors d'un accès à l'attribut. Il existe quatre facettes procédurales (ou *réflexes*) qui sont pré-définies dans Y3 :

- les facettes `si-ajout` et `si-enlève` sont activées lorsqu'on modifie la valeur de l'attribut (ajout ou retrait du contenu de la facette `value`). Dans le cas de l'objet `ensemble-d'entiers`, l'attribut `les-élements` possédera deux réflexes : « `si-ajout` = incrémenter `cardinal` » et « `si-enlève` = décrémenter `cardinal` » ;
- la facette `si-besoin` permet d'appeler une fonction calculant la valeur de l'attribut lors d'un premier accès en lecture sur cet attribut ;
- la facette `si-possible` sert à vérifier si la valeur qu'on veut ajouter à l'attribut est compatible avec les facettes de typage qui on été définies.

Les réflexes `si-ajout` et `si-enlève` sont généralement utilisés pour assurer la cohérence entre les valeurs[3] des attributs d'un ou plusieurs objets. Quant au réflexe `si-besoin`, il permet de repousser le calcul de la valeur d'un attribut jusqu'au moment où l'on en a réellement besoin. Une fois que cette valeur est calculée, elle est placée dans la facette `value` de l'attribut. Cela suppose que cette valeur n'est plus susceptible d'évoluer. À l'inverse, pour accéder à une caractéristique de l'objet dont la valeur peut constamment évoluer, il faut recourir à une *méthode*. Comme dans tous les langages de classes, Y3 permet d'associer des méthodes aux objets. Une méthode est en fait un attribut particulier dont le contenu est une fonction LELISP.

L'ensemble des objets est organisé de façon hiérarchique suivant une relation d'ordre partiel appelée relation de *spécialisation* (voir *chapitre 10*). La figure 14.5 décrit le graphe de spécialisation des descendants de l'objet `atome`.

## 14.3.2 Objets composites

La notion d'héritage ne permet pas de traduire tous les liens qui peuvent exister entre les objets. Il faut souvent recourir à d'autres relations comme la relation de « composition » qui établit un lien structurel entre des objets. Un objet est dit « composite » s'il est formé par l'agrégation d'un ensemble d'objets qui constituent ses « composants ». Chaque composant décrit une partie de l'objet composite auquel il est rattaché. Un objet `molécule` est ainsi composé d'un ensemble d'atomes et de liaisons comme le montre la figure 14.6. L'attribut `atomes` d'une `molécule` désigne l'ensemble des objets de type `atome` qui composent cette molécule. À l'inverse, l'attribut `est-contenu-par` d'un atome a pour valeur le nom de la molécule dont cet atome est un composant. Les valeurs définies par ces deux attributs étant étroitement liées, il faut mettre en place un mécanisme permettant de maintenir leur cohérence.

Supposons qu'on veuille « retirer » un atome d'une molécule, c'est-à-dire supprimer son nom de la liste d'atomes décrite par l'attribut `atomes` de la molécule. Pour maintenir la cohérence de la base de connaissances, il faut que l'atome concerné

3. Lorsque nous parlons de la « valeur d'un attribut » il s'agit en fait du contenu de sa facette `value`.

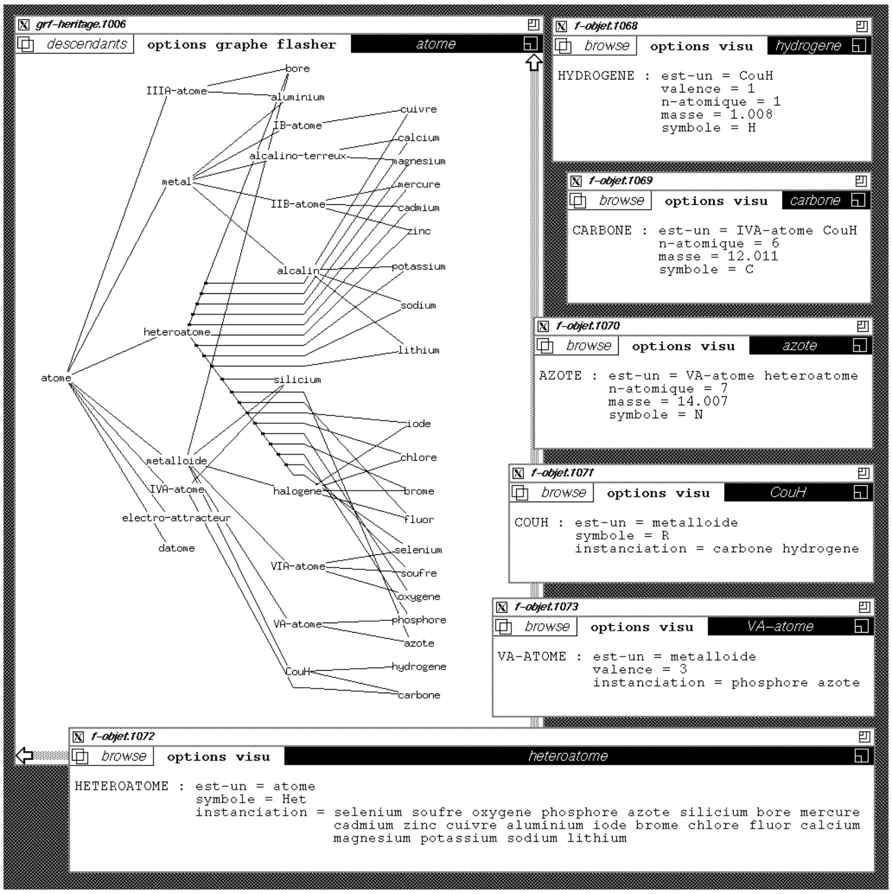



FIG. 14.5: *Le graphe de spécialisation des atomes dans* RESYN

```
(defmodele molecule
          (atomes (des . atome)
                  (lien-inverse . est-contenu-par)
                  (to-kill))
          (liaisons (des . liaison)
                    (lien-inverse . est-contenue-par)
                    (to-kill))
          ... )

(defmodele atome
          (est-contenu-par (un . molecule)
                           (to-remove)
                           (lien-inverse . atomes))
          ... )
```

FIG. 14.6: *Quelques modèles d'objets dans* RESYN.

soit « averti » du fait qu'il n'est plus un composant de la molécule. Le nom de cette dernière est alors enlevé de l'attribut `est-contenu-par` de l'atome retiré.

Pour faciliter la définition de ce type de relations, la notion de « lien inverse » a été introduite dans YAFOOL. Elle est mise en œuvre par la facette `lien-inverse`. Dans le cas de l'attribut `atomes` de l'objet *molécule*, la facette `lien-inverse` désigne l'attribut `est-contenu-par` qui lui correspond dans le modèle `atome`. L'établissement d'un « lien inverse » correspond en fait à la mise en place de réflexes `si-ajout` et `si-enlève` sur l'attribut concerné.

La définition de « liens inverses » entre les objets nécessite la mise en place d'un mécanisme de maintien de la cohérence lors de la destruction des objets. Ce mécanisme se traduit par l'ajout d'une facette `to-remove` indiquant que les valeurs de l'attribut concerné doivent être retirées avant que l'objet ne soit détruit. Ce retrait automatique peut alors entraîner le déclenchement des réflexes associés au lien inverse. Dans le cas d'une relation de composition, la survie des composants n'a généralement pas de sens lorsque l'objet composite est détruit. Une facette `to-kill`, placée dans l'attribut désignant les composants de l'objet composite, permet alors de déclencher la destruction automatique de ces composants.

### 14.3.3 Héritage cumulatif non monotone

Nous avons vu dans le paragraphe 14.2 que la connaissance liée à l'application d'une transformation chimique est représentée sous la forme d'une hiérarchie de contextes. Chaque contexte est modélisé par un objet qui contient — en dehors de la description du graphe qui lui est associé — un attribut décrivant la liste des transformations pour lesquelles ce contexte est favorable. Nous noterons `transformations-possibles` cet attribut et `transformations-interdites` l'attribut équivalent pour les cas où le contexte est défavorable.

La figure 14.7 donne un exemple d'une telle hiérarchie. Les contextes sont reliés entre eux par la relation `est-un` du langage YAFOOL : ce sont ici des *prototypes*, liés par une relation de *délégation* (voir *chapitre 8*). Chaque contexte `C_x` est décrit par un graphe (noté `G_x`). L'objet `C_x` définit la classe des contextes qui « contiennent » le graphe `G_x`. On peut ainsi dire que le contexte `C_b` est un contexte `C_a` puisque le graphe `G_b` « contient » le graphe `G_a`. De ce fait, l'objet `C_b` va pouvoir hériter des propriétés de `C_a`.

La recherche des transformations réalisant un objectif donné repose sur un raisonnement par classification permettant de déterminer les contextes les plus spécifiques contenus dans la molécule cible. Lorsqu'un tel contexte a été trouvé, il faut chercher toutes les transformations applicables dans un tel environnement.

Étudions l'exemple de la figure 14.7. Nous pouvons distinguer plusieurs cas en fonction du contexte le plus spécifique qui est présent dans la molécule cible :

1. contexte `C_a` : c'est le cas le plus simple. Les transformations applicables sont celles contenues dans l'attribut `transformations-possibles` de `C_a`.
2. contexte `C_b` : comme dans le cas précédent, on va retenir la transformation `T_3` présente dans `C_b`. Mais ce contexte est aussi un contexte `C_a`. Il hérite

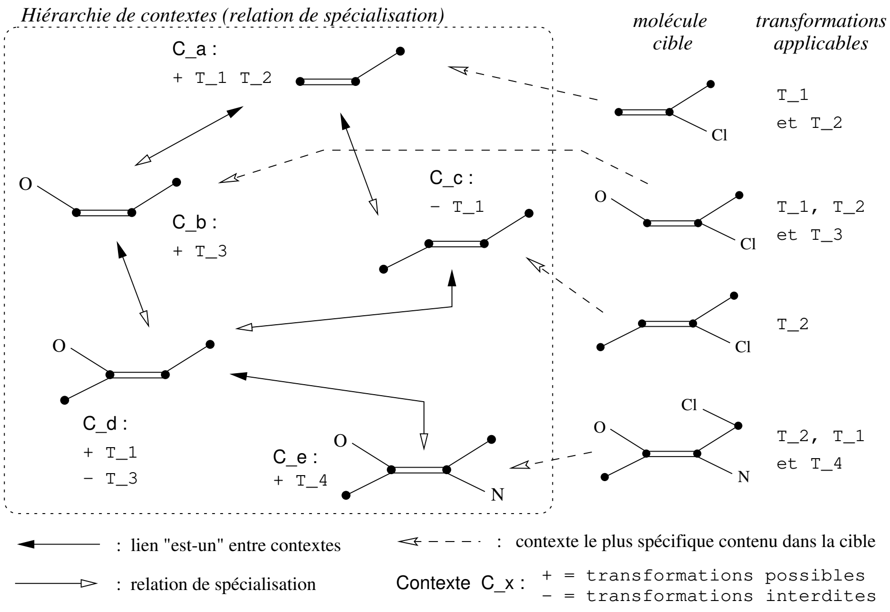



FIG. 14.7: *Héritage cumulatif non monotone*

donc des propriétés de l'environnement chimique associé à C_a. L'ensemble des transformations applicables pour C_b est ainsi composé de l'union des valeurs présentes dans les attributs transformations-possibles de C_b et de tous ses ascendants dans la hiérarchie. On parle alors d'*héritage cumulatif*.

3. contexte C_c : ce contexte hérite *a priori* des transformations possibles T_1 et T_2 de son ascendant C_a. Mais une de ces transformations apparaît dans l'attribut transformations-interdites de C_c. Ce contexte constitue une exception pour l'application de cette transformation dans le cadre de l'environnement associé à C_a. Seule la transformation T_2 sera applicable pour C_c. On retrouve la notion classique d'*héritage non monotone* liée à l'emploi d'exceptions dans les langages à objets.

4. contexte C_e : il s'agit du cas le plus général d'*héritage cumulatif non monotone*. Pour le contexte C_e, on peut appliquer les transformations présentes dans cet objet (T_4) ou celles qui apparaissent dans ses ascendants (T_2) et qui ne sont pas interdites par des ascendants intermédiaires (T_3). On notera le cas particulier de T_1 qui est définie dans C_a, interdite dans C_c et à nouveau autorisée dans C_d.

La notion d'héritage cumulatif non monotone est décrite dans [Ducournau *et al.*, 1995].

On retrouve bien évidemment les problèmes de conflit d'héritage dans le cas d'un héritage multiple. Cependant, la sémantique associée aux contextes permet de résoudre facilement ce type de conflit. Considérons un contexte C_x qui hérite des propriétés de deux contextes `C_y` et `C_z`, incomparables par la relation de spécialisation. Si une transformation `T_i` apparaît à la fois dans le champ `transformations-possibles` de `C_y` et dans le champ `transformations-interdites` de `C_z`, on résoudra le conflit en considérant que `T_i` n'est pas applicable pour le contexte `C_x`. En effet, `C_x` contient les deux environnements associés aux contextes `C_y` et `C_z`. Bien que la présence de `C_y` suggère la possibilité d'appliquer *a priori* la transformation `T_i`, l'environnement associé à `C_z` constitue un exemple connu pour lequel `T_i` n'est pas applicable. Le système peut donc résoudre le conflit en rejetant `T_i`, tout en signalant sa décision afin qu'elle soit confirmée par un expert du domaine et que `T_i` soit explicitement rajoutée dans l'attribut `transformations-interdites` de `C_x`.

## 14.4 Raisonnement par classification

Nous avons vu dans le paragraphe 14.2 que la recherche d'un contexte favorable à l'application d'une transformation est effectuée à l'aide d'un raisonnement par classification sur une hiérarchie de graphes. Les techniques généralement utilisées pour organiser une base de connaissances constituée de graphes sont semblables à celles qui sont employées pour d'autres modes de représentation des concepts : indexation, tables de « hachage » ou hiérarchisation. C'est à ce dernier mode d'organisation que nous nous intéressons ici. Il suppose l'existence d'une relation de *subsomption* entre les graphes de la base de connaissances. Cette relation repose sur la notion de *sous-graphe partiel* définie en théorie des graphes. Le raisonnement par classification apparaît alors comme le mode de traitement naturel d'une telle hiérarchie de graphes.

### 14.4.1 Préliminaires

**Relation de subsomption entre graphes**

Considérons un ensemble de concepts représentés par des graphes *non orientés* et *connexes*. Chaque graphe $G = (X, E)$ est muni d'une fonction d'étiquetage sur les sommets $f_X : X \to \mathcal{T}_X$ et sur les arêtes $f_E : E \to \mathcal{T}_E$. On suppose que les ensembles d'étiquettes $\mathcal{T}_X$ et $\mathcal{T}_E$ sont munis des relations d'ordre partiel $\geq_X$ et $\geq_E$.

Par exemple, considérons une base de connaissances contenant des molécules. Chaque molécule sera décrite par un « graphe moléculaire » dans lequel chaque sommet est étiqueté par un type atomique plus ou moins générique ($\mathcal{T}_X = \{$ *halogène, métal, carbone, azote ...*$\}$) et chaque arête est associée à un élément de l'ensemble $\mathcal{T}_E = \{$ *liaison-simple, liaison-double, liaison-triple ...* $\}$. Quant à la relation d'ordre partiel $\geq_X$, elle peut être définie sur $\mathcal{T}_X$ à partir de la classification périodique des éléments de Mendeleïev.

La subsomption entre deux graphes $G_A = (X_A, E_A)$ et $G_B = (X_B, E_B)$, appelée *co-subsomption* [Napoli, 1992], se définit de la manière suivante (cf. figure 14.8) :

**Définition 5** $G_A$ subsume $G_B$ *(noté $G_A \succeq G_B$) si et seulement si :*

> *il existe un morphisme injectif $\varphi : X_A \longrightarrow X_B$ tel que :*
> - *$\varphi$ préserve l'adjacence du graphe $G_A$ :*
>   $\forall x, y \in X_A,\ \{x, y\} \in E_A \Rightarrow \{\,\varphi(x),\, \varphi(y)\,\} \in E_B$
> - $\forall x \in X_A,\ f_{X_A}(x) \ \geq_X\ f_{X_B}(\varphi(x))$
> - $\forall \{x, y\} \in E_A,\ f_{E_A}(\{x, y\}) \ \geq_E\ f_{E_B}(\{\,\varphi(x),\, \varphi(y)\,\})$

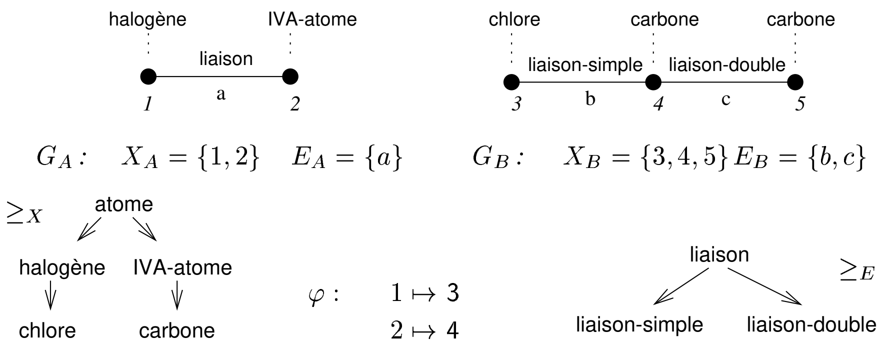



FIG. 14.8: *Relation de subsomption entre deux graphes ( $G_A \succeq G_B$ ).*

Cette relation de subsomption s'apparente à une relation d'isomorphisme de sous-graphe partiel. En effet, si le graphe $G_A$ *subsume* le graphe $G_B$, alors il existe un isomorphisme entre $G_A$ et un sous-graphe partiel de $G_B$ qui respecte les relations d'ordre $\geq_X$ et $\geq_E$. On parle alors d'*appariement* de $G_A$ dans $G_B$.

La relation de subsomption permet d'organiser un ensemble de graphes de façon hiérarchique, allant des graphes les plus généraux (c'est-à-dire les plus « petits ») aux plus spécifiques (c'est-à-dire les plus « grands »). Chaque graphe constitue la définition en *intension* d'une « classe » de graphes. On peut définir la classe associée à un graphe $G_A$ par l'ensemble infini de tous les graphes qui sont subsumés par $G_A$. Un graphe appartiendra à la classe associée à $G_A$ si et seulement si il possède un sous-graphe partiel isomorphe à $G_A$.

### Algorithmes de classification

Le raisonnement par classification est utilisé pour ajouter un nouvel objet dans une hiérarchie ou pour déterminer à quelles classes appartient cet objet afin d'en déduire ses propriétés par un mécanisme d'héritage (voir *chapitre 11*). Étant donné une hiérarchie quelconque d'objets ordonnée par une relation de subsomption $\succeq$ et étant donné un nouvel objet $c$ on doit pouvoir déterminer :

- l'ensemble des *Prédécesseurs Immédiats* de $c$, noté $PredI(c)$, défini par l'ensemble des objets les plus spécifiques parmi ceux qui subsument $c$ (on parle aussi de *Subsumants les Plus Spécifiques*) :
  $PredI(c) = \{\, x$ tel que $x \succeq c$ et $\nexists y \neq x$ vérifiant $x \succeq y \succeq c\,\}$

- l'ensemble des *Successeurs Immédiats* de $c$, noté $SuccI(c)$, défini par l'ensemble des objets les plus généraux parmi ceux qui sont subsumés par $c$ (on parle aussi de *Subsumés les Plus Généraux*):
  $SuccI(c) = \{\, x$ tel que $c \succeq x$ et $\nexists y \neq x$ vérifiant $c \succeq y \succeq x \,\}$

*Calcul des prédécesseurs immédiats*

Les méthodes permettant de déterminer l'ensemble des prédécesseurs immédiats tentent de limiter le nombre d'objets étudiés en se basant sur la transitivité de la relation de subsomption (cf. [Baader *et al.*, 1994]).

Il existe deux méthodes principales pour construire l'ensemble $PredI(c)$. La première repose sur un parcours en profondeur de la hiérarchie. Elle est issue du système KL-ONE [Lipkis, 1982]. Quant à la seconde, proposée par Levinson [Levinson, 1992], elle consiste à étudier les objets de la hiérarchie suivant une extension linéaire de l'ordre partiel défini par la relation de subsomption. Pour ce faire, Levinson utilise un marquage sur les objets qui peut prendre les valeurs *vrai, faux* et *non-marqué*. Au début de l'algorithme, aucun objet n'est marqué et l'ensemble résultat $S$ est vide.

```
Recherche des predecesseurs immediats de c :
  S ← ∅;
  Tantque | il existe un objet x non marqué, tel que PredI(x) = ∅          | Faire :
          | ou ∀z ∈ PredI(x), z est marqué avec la valeur vrai              |
     Si x subsume c Alors
        marquer x avec la valeur vrai;
        S ← ( S \ PredI(x) ) ∪ {x};
     Sinon marquer x avec la valeur faux;
```

Pour améliorer la classification sur une hiérarchie de graphes, Levinson suggère d'y ajouter de nouveaux graphes baptisés « graphes index ». Ces graphes sont de petite taille et permettent de focaliser la recherche des prédécesseurs sur un ensemble réduit de graphes de la hiérarchie qui sont susceptibles de subsumer le graphe $c$ à classer. En effet, si $c$ est subsumé par un certain nombre de graphes index, les prédécesseurs de $c$ seront aussi subsumés par ces mêmes graphes index. Les graphes index sont placés en haut de la hiérarchie dont ils constituent les éléments les plus généraux. La recherche des prédécesseurs immédiats de l'objet $c$ à classer commence par la détermination des graphes index qui subsument $c$. L'algorithme est alors appliqué en ne considérant que les descendants de ces graphes index dans la hiérarchie, réduisant par là même le nombre d'objets étudiés.

Il faut cependant noter que la détermination d'un ensemble optimal de graphes index pour une hiérarchie donnée reste un problème difficile à résoudre. Par ailleurs, il arrive souvent que les hiérarchies manipulées contiennent des graphes de petite taille – décrivant des connaissances élémentaires – qui constituent en fait autant de graphes index. C'est notamment le cas dans le système RESYN.

*Calcul des successeurs immédiats*

La détermination des successeurs immédiats d'un objet $c$ s'appuie sur le calcul des prédécesseurs immédiats. En effet, pour être subsumé par $c$, un objet $x$ doit nécessairement l'être par les prédécesseurs immédiats de $c$. Il faut cependant noter que les successeurs immédiats de $c$ ne sont que rarement des successeurs immédiats des éléments de $PredI(c)$ et peuvent se trouver beaucoup plus bas dans la hiérarchie.

Pour calculer l'ensemble des successeurs immédiats de $c$ on peut utiliser la méthode proposée par Levinson [Levinson, 1992 ; Ellis, 1993 ; Guinaldo, 1996]. On commence par déterminer la sous-hiérarchie contenant tous les objets qui sont subsumés par tous les prédécesseurs immédiats de $c$. Cette sous-hiérarchie, appelée *focus*, correspond à l'intersection de toutes les sous-hiérarchies ayant pour racine un des prédécesseurs immédiats de $c$. On utilise ensuite une extension linéaire de la relation de subsomption pour parcourir cette sous-hiérarchie de façon à toujours étudier un objet $x$ avant les objets qu'il subsume. Si $x$ est subsumé par $c$, il est placé dans l'ensemble $SuccI(c)$ et tous ses descendants sont retirés de la sous-hiérarchie. Le processus s'arrête lorsque la sous-hiérarchie est vide.

### 14.4.2 Subsomption d'appariements

Le processus de classification que nous avons décrit jusqu'ici reste très général. Il s'applique à tout type de graphe étiqueté, comme les graphes conceptuels [Guinaldo, 1996]. Dans le cadre de la chimie, ce mécanisme général doit être complété pour tenir compte de la spécificité du mode de classification inhérent au domaine.

Dans le système RESYN, l'application d'une transformation chimique est conditionnée par l'existence d'un contexte favorable. Nous avons vu que la recherche d'un tel contexte repose sur un processus de classification dans une hiérarchie de graphes (§ 14.2).

Lorsqu'on cherche à appliquer une transformation à un endroit précis de la molécule, on dispose du graphe $G_O$ décrivant l'objectif de la transformation et de son appariement $f_O$ vers le graphe moléculaire $G_{mol}$. Il s'agit d'étudier les appariements de contextes autour de $f_O$.

Dans l'exemple de la figure 14.9, la transformation associée à l'objectif $G_O$ peut être appliquée à deux parties de la molécule qui correspondent aux appariements $f_O$ et $f'_O$.

Pour savoir si la transformation est applicable, on étudie l'ensemble des contextes qui sont à la fois plus spécifiques que l'objectif $G_O$ et plus généraux que le graphe $G_{mol}$. Parmi ces contextes, on ne s'intéresse qu'à ceux qui peuvent « intégrer » l'appariement $f_O$, c'est-à-dire tous les contextes $G_i$ tels que :

- $G_i$ est subsumé par $G_O$,
- et il existe un appariement $f_i$ de $G_i$ dans $G_{mol}$ tel que l'image de $G_O$ dans $G_{mol}$ par $f_O$, notée $f_O(G_O)$, soit un sous-graphe partiel de $f_i(G_i)$ ;

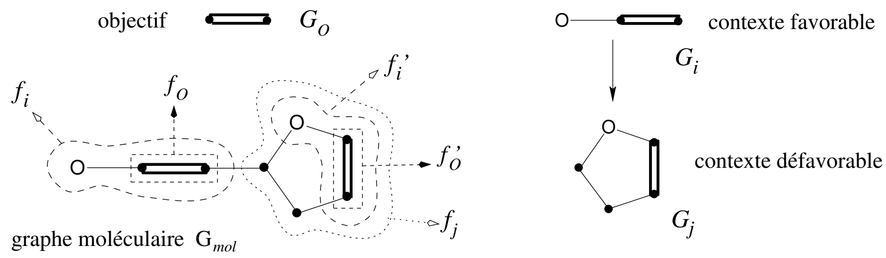



FIG. 14.9: *Appariements d'un objectif et de contextes dans un graphe moléculaire*

Sur la figure 14.9, l'appariement $f_O$ de l'objectif $G_O$ est « recouvert » par l'appariement $f_i$ du contexte *favorable* $G_i$. On en déduit que la transformation est applicable à cet endroit de la molécule. Par contre, le second appariement $f'_O$ de l'objectif est « recouvert » non seulement par un appariement $f'_i$ de $G_i$ mais aussi par l'appariement $f_j$ du contexte *défavorable* $G_j$. Il en résulte que la transformation n'est pas applicable dans la partie de la molécule désignée par l'appariement $f'_O$.

Bien que ces deux exemples concernent un même graphe moléculaire $G_{mol}$ qui est subsumé par le contexte défavorable $G_j$, l'application de la transformation est possible pour l'appariement $f_O$ de l'objectif.

Nous pouvons en déduire que le problème de l'application d'une transformation ne peut pas se résoudre par un simple raisonnement par classification basé sur la relation de subsomption $\succeq$ définie entre graphes. Il ne faut pas rechercher tous les contextes qui subsument $G_{mol}$ mais uniquement ceux qui peuvent s'apparier dans $G_{mol}$ en intégrant l'appariement $f_O$ de l'objectif.

Il paraît ainsi nécessaire d'introduire une nouvelle relation de subsomption $\succeq_{app}$ qui soit définie non pas sur l'ensemble des contextes mais plutôt sur celui de leurs appariements dans le graphe moléculaire.

**Définition 6** *On dit que l'appariement $f_i$ du graphe $G_i$ dans $G_{mol}$* subsume *l'appariement $f_j$ de $G_j$ dans $G_{mol}$, noté $\boldsymbol{f_i} \succeq_{\text{app}} \boldsymbol{f_j}$, si et seulement si $f_i(G_i)$ est un sous-graphe partiel de $f_j(G_j)$ (noté $f_i(G_i) \sqsubseteq f_j(G_j)$).*

Cette nouvelle relation de subsomption est étroitement liée à la relation $\sqsubseteq$ : « *est sous-graphe partiel de* ». La relation $\succeq_{app}$ constitue un *préordre* partiel sur l'ensemble des appariements de contextes possibles dans le graphe $G_{mol}$.

Nous allons utiliser la hiérarchie définie sur les graphes, par la relation $\succeq$, pour déterminer tous les appariements qui sont subsumés par $f_O$ relativement à $\succeq_{app}$. On peut s'appuyer sur la propriété évidente :

**Propriété 1** *Si $f_i$ et $f_j$ sont des appariements respectifs de $G_i$ dans $G_{mol}$ et de $G_j$ dans $G_{mol}$, alors* $f_i \succeq_{\text{app}} f_j \implies G_i \succeq G_j$ .

Pour trouver tous les appariements qui intègrent $f_O$, on peut déterminer parmi les descendants de $G_O$, dans la hiérarchie définie par $\succeq$, ceux qui subsument $G_{mol}$ (relativement à $\succeq$). Pour chaque contexte $G_i$ ainsi trouvé, on ne retient que les appariements $f_i$ tels que $f_O \succeq_{app} f_i$.

Cette méthode n'est cependant pas satisfaisante en pratique. En effet, si le graphe $G_{mol}$ étudié est relativement grand, le graphe associé à un contexte donné peut s'apparier en de nombreux endroits de $G_{mol}$. Or parmi tous ces appariements, nous ne nous intéressons qu'à ceux qui sont subsumés par $f_O$.

Il serait donc intéressant de pouvoir focaliser la recherche des appariements de contextes « autour » du sous-graphe partiel de $G_{mol}$ défini par $f_O(G_O)$. C'est cette idée qui est à la base de la notion d'« *appariements dirigés* » que nous allons exposer maintenant.

### 14.4.3 Appariements dirigés

Pour déterminer un appariement d'un graphe $G_i$ dans le graphe $G_{mol}$, il faut associer chaque sommet de $G_i$ à un sommet de $G_{mol}$ de telle sorte que le morphisme injectif obtenu préserve l'adjacence de $G_i$ et les contraintes sur les étiquettes. Dans le système RESYN, nous avons utilisé un *réseau de contraintes* pour modéliser ce problème en termes de CSP[4] [Régin, 1995 ; Bessière *et al.*, 1995].

Il serait trop long de détailler ici cette méthode, mais on peut dire que chaque sommet de $G_i$ peut *a priori* s'apparier avec tout sommet de $G_{mol}$ ayant une étiquette et un voisinage compatibles. Pourtant, seuls les sommets de $G_{mol}$ qui sont « autour » de $f_0(G_O)$ nous intéressent.

D'après la définition 6, si $f_i$ est un appariement de $G_i$ dans $G_{mol}$ tel que $f_O \succeq_{app} f_i$ alors $f_O(G_O) \sqsubseteq f_i(G_i)$. Cela signifie que certains sommets de $G_i$ doivent être appariés avec des sommets appartenant à $f_O(G_O)$. En effet, puisque $G_O$ subsume $G_i$, tout appariement de $G_i$ dans $G_{mol}$ doit « contenir » un appariement de $G_O$ dans $G_{mol}$.

Le principe des « appariements dirigés » consiste alors à utiliser $f_O$ pour construire $f_i$. En effet, si on détermine les relations structurelles d'inclusion qui existent entre les graphes $G_O$ et $G_i$, on peut considérer $f_O$ comme la « solution partielle » d'un morphisme $f_i$ de $G_i$ dans $G_{mol}$. Ici nous ne décrirons que brièvement cette méthode (cf. [Vismara, 1995 ; Vismara, 1996] pour une présentation plus complète).

Considérons deux graphes $G_i$ et $G_j$ tels que $G_i \succeq G_j$ et supposons qu'on connaisse les appariements possibles de $G_i$ dans le graphe $G_{mol}$ (figure 14.10). La méthode des « appariements dirigés » va nous permettre de calculer les appariements de $G_j$ dans $G_{mol}$ à partir de ceux de $G_i$ dans $G_{mol}$.

Pour ce faire, on dispose de l'ensemble, noté $AppDir_{i\to j}$, des *appariements dirigés de $G_i$ dans $G_j$*. Cet ensemble contient tous les appariements de $G_i$ dans $G_j$ qui ne sont pas équivalents à une symétrie de $G_j$ près[5]. L'ensemble $AppDir_{i\to j}$ peut être calculé à l'avance et stocké dans la hiérarchie de graphes au niveau du lien existant entre $G_i$ et $G_j$. Le calcul des appariements de $G_j$ dans $G_{mol}$ s'effectue alors de la manière suivante : pour chaque appariement $f_i$ de $G_i$ dans $G_{mol}$, on construit

4. *Constraint Satisfaction Problem*, voir *chapitre 9*.

5. On peut montrer que la suppression des appariements symétriques relativement à $G_j$ ne remet pas en cause la validité de la méthode, tout en réduisant la taille des ensembles d'appariements dirigés. Il faut également préciser que d'autres contraintes interviennent dans la définition de ces ensembles [Vismara, 1996].

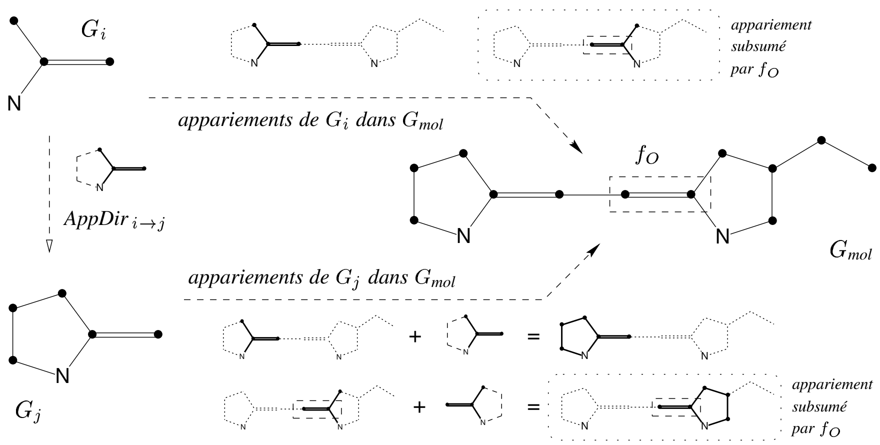



FIG. 14.10: *Principe des appariements dirigés.*

un, ou plusieurs, appariements $f_j$ de $G_j$ dans $G_{mol}$, en utilisant les appariements stockés dans l'ensemble $AppDir_{i \to j}$ pour étendre $f_i$ au graphe $G_j$. Cette méthode est illustrée par la figure 14.10. Pour chacun des deux appariements de $G_i$ dans $G_{mol}$, on obtient un appariement de $G_j$ dans $G_{mol}$ en utilisant l'appariement de $G_i$ dans $G_j$ qui est stocké dans l'ensemble $AppDir_{i \to j}$.

Il est important de noter ici que si parmi les appariements de $G_i$ dans $G_{mol}$, un seul $f_i$ est subsumé par l'appariement $f_O$ de l'objectif, on ne construira qu'un seul appariement $f_j$ de $G_j$ dans $G_{mol}$ à partir de $f_i$.

L'intérêt de la méthode des appariements dirigés est double.

D'un point de vue algorithmique, ils permettent de focaliser la recherche des appariements des graphes de la hiérarchie en se limitant à la partie du graphe cible $G_{mol}$ entourant l'appariement $f_O$ initial. Lorsque le graphe cible est relativement grand, cette méthode permet d'éviter des calculs d'appariements inutiles.

Par ailleurs, la notion d'appariements dirigés offre un grand intérêt pour la représentation de certaines connaissances liées au domaine d'application. Nous avons vu que dans RESYN, les graphes de la hiérarchie permettent de définir des environnements favorables ou défavorables à l'application d'une transformation caractérisée par son graphe objectif. Comme le montre la figure 14.11, certains contextes intègrent plusieurs appariements de l'objectif qui sont favorables ou défavorables, ou pour lesquels on ne dispose d'aucune information. La transmission de ce dernier type d'appariements est alors inutile et alourdit le processus de classification. La notion d'appariements dirigés permet ainsi de ne considérer que les appariements qui apportent une réelle information. Ce mécanisme correspond au raisonnement de l'expert qui élimine directement les appariements entre structures moléculaires n'ayant aucun sens chimique.

De façon générale, la méthode des appariements dirigés peut être utilisée pour tout raisonnement par classification sur des hiérarchies de graphes. Elle se simplifie même lorsque le problème se limite à la recherche des prédécesseurs et des

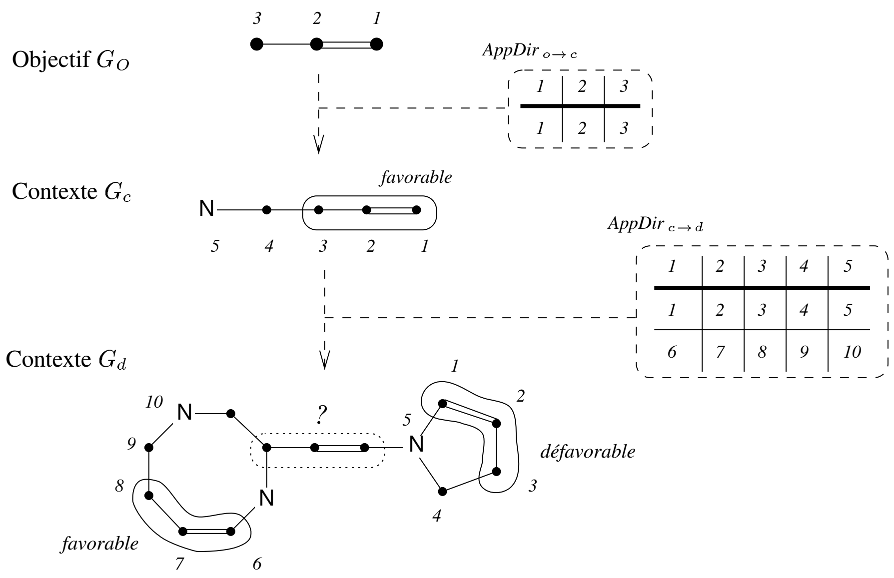



FIG. 14.11: *Les appariements dirigés permettent de ne pas transmettre les appariements pour lesquels on ne dispose d'aucune information.*

successeurs immédiats d'un graphe pour la relation classique de subsomption $\succeq$. La comparaison de deux graphes se réduit alors à la recherche d'un appariement quelconque, sans qu'il faille se focaliser autour d'un appariement initial $f_O$.

Précisons enfin que l'idée de stocker, dans la hiérarchie, des informations explicites sur les relations structurelles existant entre deux graphes a été également suggérée par [Levinson, 1994], dans le cadre des *graphes conceptuels*. La solution qu'il décrit est cependant moins efficace que celle des appariements dirigés. Si un graphe $A$ est le prédécesseur d'un graphe $B$ dans la hiérarchie, il propose de simplement stocker pour chaque sommet $x$ de $A$ l'ensemble des sommets de $B$ avec lesquels $x$ peut être apparié. Levinson justifie son choix de ne pas directement stocker tous les appariements entre $A$ et $B$ par le fait que le nombre de ces appariements peut s'avérer très important. Si ce problème reste théoriquement possible avec les appariements dirigés, il est largement résolu en pratique par l'élimination des appariements symétriques inutiles.

## 14.5 Raisonnement distribué

L'élaboration d'un plan de synthèse passe par la définition *d'objectifs stratégiques*. Un objectif stratégique est une sous-structure de la molécule cible à laquelle une action a été associée. Un tel objectif formalise un sous-problème d'un problème de synthèse : l'accomplissement de l'action associée à chaque sous-structure résout ce sous-problème. En rétrosynthèse des actions de deux types sont appliquées à une cible : celles qui visent à *transformer* une partie de la cible et celles destinées à *préserver* une sous-structure.

Les *stratégies de synthèse* [Corey, 1967 ; Trombini, 1987] sont des collections de règles qui assistent le chimiste dans la définition d'objectifs stratégiques. Ces règles s'appuient sur différentes caractéristiques moléculaires. Certaines prennent en compte des aspects topologiques, d'autres la stéréochimie, etc. Nous représentons chacun des aspects d'un problème de synthèse par une entité autonome, appelée *agent*. C'est la coopération de ces agents qui permet au système, alors qualifié de « multi-agents », de résoudre un problème. Sur la figure 14.12 trois agents participent à la résolution du problème : l'agent *Topologie*, l'agent *Stéréochimie* et l'agent *Chimie*. Chaque agent calcule une *vue locale* du problème à résoudre qui correspond au point de vue que l'agent modélise. Une *vue globale* est reconstituée grâce aux *interactions* que les agents entretiennent entre eux.

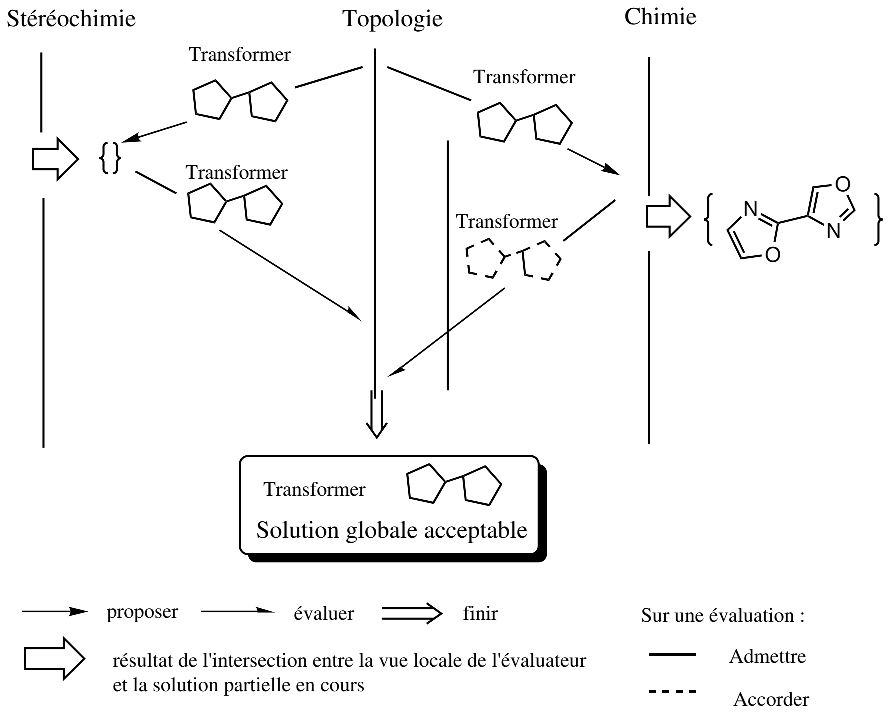



FIG. 14.12: *Passage de vues locales distribuées à une solution globale*

Une vue locale est un ensemble de sous-graphes de la molécule cible. Notamment, la vue locale de l'agent *Topologie* est constituée des cycles, chaînes, liens, systèmes cycliques composant la cible. C'est à partir de ces sous-graphes qu'une vue globale est calculée. *Les agents communiquent donc en échangeant des graphes*. Les interactions des agents ont pour objectif de déterminer si les différents aspects du problème à résoudre convergent ou divergent. Lorsque sur un même sous-graphe de la cible les agents appliquent une action de même type (par exemple transformer), leurs points de vue convergent. Dans le cas contraire, les deux classes d'actions étant *incompatibles*, les connaissances locales des différents agents sont en conflit.

La résolution d'un problème résulte de plusieurs dialogues. Un dialogue se déroule ainsi. L'un des agents soumet un objectif stratégique aux autres agents participant en même temps que lui à la résolution du problème courant (ses *accointances*). À la réception d'un objectif stratégique un agent calcule s'il existe une intersection entre le sous-graphe représentant l'objectif stratégique reçu et les différents sous-graphes composant sa vue locale. Si une telle intersection existe, elle est matérialisée par un ensemble d'arêtes correspondant aux liaisons de la molécule cible appartenant à différentes vues locales. Selon que les actions appliquées à ces liaisons communes sont compatibles ou non, les agents étiquettent les arêtes des sous-graphes reçus avec les valeurs *accorder* ou *réfuter*. Les arêtes n'appartenant pas à cette intersection sont étiquetées avec la valeur *admettre*. Sur la figure 14.12, le résultat de l'intersection entre vue locale et objectif proposé est donné entre accolades. Cette intersection est vide pour l'agent *Stéréochimie* qui étiquette avec *admettre* chacune des arêtes de l'objectif proposé. Cette intersection n'est pas vide en revanche pour l'agent *Chimie*. Les actions appliquées par les deux agents sont compatibles : l'agent *Chimie* étiquette avec *accorder* chacune des arêtes du sous-graphe reçu.

C'est à partir de ces étiquettes que l'agent qui a soumis à ses accointances un objectif stratégique affecte à ce dernier un état global. Si aucune arête n'a été réfutée, l'objectif stratégique proposé devient une solution globale au problème de synthèse courant. Sinon, l'agent fait appel à une base de connaissances pour résoudre ce conflit.

Cette approche distribuée des stratégies de synthèse a été intégrée au système RESYN et est plus amplement discutée dans [Jambaud, 1996]. Un agent est implémenté par un objet YAFOOL. `Boîte-aux-lettre`, `Accointances`, `Vue-locale` sont quelques uns des attributs importants de la classe `agent`. Le dialogue est implémenté à l'aide d'opérateurs qui permettent aux agents de proposer des objectifs stratégiques, de les critiquer et de leur affecter un état global. Une méthode YAFOOL, définie dans la classe `agent`, est associée à chacun de ces opérateurs de dialogue. Par ailleurs, chaque spécialisation de la classe `agent` dispose d'une méthode spécifique de `perception`, qui lui permet de construire sa vue locale selon l'aspect concerné (topologie, stéréochimie ou chimie).

RESYN est aujourd'hui à même d'élaborer des directives stratégiques à partir de bases de connaissance, en confrontant plusieurs points de vue. Cette démarche se distingue des systèmes actuels d'aide à la synthèse qui ne peuvent prendre en compte qu'un seul point de vue. De ce fait, il s'agit là d'un bon exemple d'utilisation du paradigme multi-agents en résolution de problèmes.

## 14.6 Conclusion

La construction d'un système capable d'aider à résoudre des problèmes complexes du monde réel nécessite de modéliser explicitement le domaine d'application et de mettre en œuvre des techniques informatiques permettant la représentation de connaissances multiples — à différents niveaux d'abstraction — et la coopération de plusieurs méthodes de résolution [David *et al.*, 1993]. C'est la démarche que nous avons suivie pour réaliser le système RESYN, lequel a pour but d'aider les chimistes dans la conception de plans de synthèse pour des molécules organiques.

La résolution des problèmes de synthèse fait appel à des connaissances nombreuses et diverses allant de concepts très abstraits de la théorie chimique jusqu'à des faits comme l'existence d'une réaction particulière ou la disponibilité commerciale d'une molécule. Malgré sa diversité, l'ensemble de ces connaissances possède la propriété de pouvoir se décrire par des graphes organisés en hiérarchies. Ainsi, la théorie des graphes offre-t-elle un cadre formel convenable pour représenter, manipuler et rechercher des objets structurés tels que les molécules, leurs transformations ou les contextes d'application de ces transformations. À ces hiérarchies de graphes est associée la classification, puissant mécanisme de raisonnement utilisé tant dans la résolution des problèmes que dans la mise à jour des connaissances. Un langage à objets comme Y3, de par ses fortes capacités de gestion de connaissances à la fois modulaires et évolutives, s'est avéré très efficace dans la mise en œuvre de ce modèle lors de la réalisation du système RESYN.

Cependant, un problème aussi complexe que la synthèse d'une molécule organique ne peut être résolu en appliquant un type unique de raisonnement. Au delà de la classification de la molécule cible par rapport à diverses catégories d'objets chimiques, l'analogie faite par le chimiste entre son problème et des problèmes déjà résolus guide en grande partie la conception d'un plan de synthèse. C'est pourquoi, un mécanisme de raisonnement à partir de cas est en cours d'intégration dans RESYN [Napoli et Lieber, 1994]. Il doit permettre au système d'utiliser une base de données de chemins de synthèse afin d'adapter au problème posé les solutions ou fragments de solutions que ces chemins décrivent. Le chimiste fait également appel à de nombreuses heuristiques, en particulier pour définir et ordonner les objectifs à atteindre. Ces stratégies de synthèse sont encore peu formalisées, ce qui n'autorise pas l'adoption d'une méthodologie de construction de Système à Base de Connaissances (SBC) pour en doter RESYN. Elles forment plusieurs classes qui correspondent chacune à des points de vue distincts sur un même problème, lesquels peuvent être convergents ou divergents sur une action à entreprendre. Nous avons choisi de représenter de manière indépendante la connaissance relative à chaque point de vue, la construction de directives stratégiques s'effectuant à l'aide d'un raisonnement distribué entre des agents disposant de cette connaissance et pourvus de procédures de négociation.

La réalisation d'une application dans un domaine aussi complexe que la synthèse en chimie organique a permis d'approfondir des notions fondamentales en représentation par objets et sur les mécanismes de raisonnement qui peuvent lui être associés. Finalement, grâce aux techniques informatiques utilisées, et bien qu'il ne soit encore qu'un prototype, RESYN est un système d'aide à la conception de plans de synthèse considérablement plus avancé que les systèmes déjà connus.

# 15

# Représentations par objets et classifications biologiques

Il y a deux écueils à éviter en parlant de domaines aussi vastes que l'informatique et la biologie, d'une part se situer à un niveau de généralité telle que le discours perd toute pertinence et, d'autre part, ne s'exprimer que par des analogies convaincantes uniquement pour un public connaissant bien les deux domaines. C'est pourquoi, le propos de ce chapitre se limitera doublement à la fois en informatique et en biologie.

Cette limitation est nécessaire pour l'informatique car de nombreux domaines de cette discipline scientifique concernent la biologie. Parmi ceux-là, en biologie comme ailleurs, la programmation par objets (voir *chapitre 3*) a une grande importance pratique. Par exemple, la complexité croissante des simulations informatiques en génétique des populations [Quesneville, 1996] ou en écologie [Legendre, 1996] ne peut que bénéficier des travaux sur la programmation distribuée (voir *chapitres 6 et 7*) et des avancées des méthodes de conception et de programmation par objets (voir *chapitre 4*). De même, les gigantesques projets de bases de données envisageant d'inventorier toute la biodiversité de la planète, comme *Systematics Agenda 2000*[1], ne peuvent se concevoir de manière réaliste sans considérer attentivement les bases de données objets (voir *chapitre 5*).

Les objets sont aussi utilisables en biologie pour la représentation des connaissances (voir *chapitre 10*), et le lien le plus profond entre le « monde des objets » et la biologie est certainement la *classification*. C'est pourquoi ce chapitre se concentre justement sur le domaine de la biologie qui produit les classifications des êtres vivants : la *systématique* [Matile *et al.*, 1987]. Ainsi, après une présentation des différents aspects que recouvre le terme de classification, ce chapitre donne un aperçu des méthodes relatives à la classification en systématique en les comparant avec les principes de classification utilisés dans les représentations par objets.

---

1. Voir par exemple `http://www.nhm.ac.uk/esf/leidweb.html`.

## 15.1 Les sens de « classification »

Un premier lien entre les « objets » et la biologie saute aux yeux quand on regarde les illustrations des articles sur la représentation des connaissances par objets. Les biologistes seraient surpris de voir autant de classifications d'êtres vivants dans ces publications d'informatique. En effet, qui dans le « monde des objets » n'a pas entendu parler de l'éléphant Clyde ou des autruches, oiseaux qui ont le mauvais goût de ne pas voler ? Bien sûr, cette connexion n'est que superficielle et probablement caricaturale mais elle montre clairement qu'un des domaines privilégiés où l'on trouve de bons exemples de classification est bel et bien la biologie.

Mais il faut faire attention car ce terme de classification ne désigne pas la même chose pour les informaticiens et les biologistes. De plus, même dans le simple contexte de chacun de ces deux domaines, comme tous les termes du langage commun ayant pris dans l'histoire un sens technique, la classification désigne selon les interlocuteurs des notions liées mais plus ou moins différentes et souvent malheureusement confondues.

### 15.1.1 Classification = Structure

Pour un mathématicien, une classification est une structure mathématique particulière dite « structure classificatoire ». C'est un ensemble recouvrant de sous-ensembles non vides d'un ensemble.

Si on appelle *I*, l'ensemble des items à classifier, un ensemble $s \subset 2^I$ est une *structure classificatoire* de $I$ si, et seulement si, ses éléments, les classes, sont non vides et recouvrants, c'est-à-dire si et seulement si :

1. $\emptyset \notin s$
2. $\cup c \in s = I$

Cette structure formalise une notion très générale dont on retrouve des instances variées notamment dans des activités scientifiques de toutes natures. On en reconnaît plusieurs types. Formellement, un ensemble $S$ de structures classificatoires est un *type de structure classificatoire* si et seulement si :

1. $\exists s \in S$ tel que $I \in s$
2. $\exists s \in S$ tel que $\forall i \in I, \{i\} \in s$

Les types de structures classificatoires les plus usuels sont les *partitions* dont les classes sont disjointes, c'est-à-dire dont :

$$\forall (c, c') \in s^2, c \cap c' = \emptyset$$

et les *hiérarchies* qui ont nécessairement $I$ et ses singletons comme éléments et dont toute paire de classes a, soit une intersection vide, soit un de ses éléments inclus dans l'autre. C'est-à-dire formellement une structure classificatoire $s$ est une hiérarchie si et seulement si :

1. $I \in s$

2. $\forall i \in I, \{i\} \in s$
3. $\forall (c, c') \in s^2, c \cap c' \in \{\emptyset, c, c'\}$

Il existe bien d'autres types de structures classificatoires, comme par exemple les *pyramides*, mais qui ne seront pas traitées ici car leurs rapports avec la biologie sont plus lointains.

Le rappel de la définition des hiérarchies est particulièrement utile ici car, d'une part, comme on le détaillera plus loin, les classifications biologiques sont des structures classificatoires fondamentalement hiérarchiques et car, d'autre part, l'usage du terme de hiérarchie est hésitant à la fois en biologie et en informatique. En particulier, on remarquera que les hiérarchies des langages de représentation par d'objets ne correspondent pas exactement à la définition ensembliste formelle précédente : non seulement les classes de ces « hiérarchies » n'ont pas nécessairement qu'une seule classe immédiatement incluante, leur super-classe, mais surtout elles ne se limitent pas à indiquer les relations d'inclusion entre classes car elles cherchent aussi à représenter les propriétés soit nécessaires, soit suffisantes, soit les deux qui rationalisent ces relations (voir *chapitre 12*).

Il est à noter aussi que cette définition d'une hiérarchie rejette totalement le sens usuel de ce mot tel que dans l'expression de « hiérarchie militaire » qui est mieux décrite par une structure de préordre total. En effet les classes de militaires (généraux, lieutenants ... ) sont disjointes et ordonnées. Cette distinction entre les sens courants de hiérarchie est cruciale en biologie car des formes archaïques (mais encore vivaces hors de la systématique) de classification des êtres vivants cherchent à distinguer « dans l'échelle de la Nature » des groupes supérieurs et des groupes inférieurs plutôt que des relations d'inclusion entre groupes. Dans cette optique dépassée, les groupes d'êtres vivants comme poissons, amphibiens, reptiles, oiseaux et mammifères sont, comme dans la hiérarchie militaire, disjoints et ordonnés par un hypothétique degré d'évolution dont le sens concret reste encore à définir.

Ajoutant à la confusion, on découvre souvent en biologie des hiérarchies, cachées aux yeux de leurs utilisateurs, sous la forme d'autres structures mathématiques en bijection avec les hiérarchies comme certains graphes, certains ordres ou certaines mesures de dissimilarités. Ainsi par exemple, tirant parti de l'existence d'une bijection entre les hiérarchies et les arbres enracinés, une classification en biologie prend souvent la forme, sous le nom de phylogramme, phénogramme ou cladogramme, du tracé d'un tel graphe.

Plus loin dans le texte, le terme de hiérarchie fera toujours référence à la définition ensembliste classique rappelée ici.

### 15.1.2 Classification = Algorithme

Le terme de classification en informatique est plutôt réservé à la désignation de traitements particuliers exploitant de manières diverses des structures classificatoires. Trois types de problèmes de classification peuvent être reconnus.

1. On rencontre des problèmes de *classification stricto sensu* où l'on cherche à construire une structure classificatoire d'un type choisi. Un algorithme résolvant ce type de problème peut se formaliser comme : $I$ et $S$ étant donnés,

| Nom | But | Entrées | Sorties |
|---|---|---|---|
| Classification *stricto sensu* | Construction d'une structure classificatoire | $I, S$ | $s \in S$ |
| Discrimination | Construction d'une méthode d'identification | $I, s \in S$ | $R : I \to s$ |
| Identification | Affectation d'un item | $i \in I, s \in S$ | $c \in s$ tel que $i \in c$ |

TAB. 15.1: *Les trois traitements relevant de la classification. $I$ est l'ensemble des items, $i$ est un item, $S$ est un type de structure classificatoire, $s$ est une structure classificatoire, $c$ est une classe, $R$ est une règle d'affectation.*

trouver $s \in S$. C'est à cette catégorie que correspondent en particulier les algorithmes de classification de l'analyse des données et les algorithmes de regroupement conceptuel de l'apprentissage automatique.

2. On trouve des problèmes de *discrimination* où l'on veut construire des méthodes (ou ce qui est équivalent dans ce cas, des règles) associant une (ou plusieurs) classes d'une structure classificatoire à un objet à classer. Ainsi un algorithme qui, étant donné $I$ et $s \in S$, fournit une application $R : I \to s$, est un algorithme de discrimination. Ces algorithmes ont été pour la plupart développés historiquement en statistique, en analyse des données et en reconnaissance des formes. Cela correspond à ce qui est appelé algorithme de classification en apprentissage automatique.

3. Enfin le terme de classification est parfois relatif à des problèmes d'*identification*, caractérisés par l'exploitation d'une méthode qui associe une ou plusieurs classes d'une structure classificatoire à un objet à classer. Dans ce cas, à partir de $i \in I$ et de $s \in S$ qui sont donnés, on cherche $c \in s$ tel que $i \in c$. Il en est ainsi dans le domaine de la représentation des connaissances par objets qui, sous le nom d'algorithmes de classification, étudie des algorithmes d'identification particuliers (voir *chapitre 12*). De manière générale, cette dernière catégorie forme un domaine particulièrement vaste, abordé notamment par des aspects très divers des mathématiques appliquées.

Par delà ces ambiguïtés terminologiques, on remarque que les deux dernières catégories d'algorithmes (discrimination et identification) nécessitent, pour leur exécution, qu'une structure classificatoire ait été construite au préalable, ce qui explique leur rapprochement des algorithmes de classification *stricto sensu*.

Ces trois types de problèmes de classification et les algorithmes correspondants peuvent être considérés comme des tentatives d'automatisation de l'induction d'une des formes possibles que peut prendre la connaissance experte et jouent à ce titre un rôle important dans de nombreux domaines scientifiques. Ainsi, après une structuration des observations faites dans le monde réel (algorithmes de classification *stricto sensu*), on cherche à établir des règles ou des lois (algorithmes de discrimination) dont l'usage (algorithmes d'identification), ensuite, nous donne un pouvoir prédictif. Leurs vastes possibilités d'application expliquent vraisemblablement la variété des approches pour mettre en œuvre ces algorithmes et la diversité des termes pour les désigner.

### 15.1.3 Représentation par objets et classification

En représentation par objets, l'accent est surtout mis sur la représentation et l'utilisation des structures classificatoires plus que sur les algorithmes de construction automatique de ces structures. Pourtant tous ces aspects de la classification semblent difficilement dissociables en pratique. En effet, comment justifier un quelconque traitement appliqué à une structure classificatoire sans savoir ce qu'elle signifie ? Et comment savoir ce qu'elle signifie sans savoir d'où elle vient et comment elle a été construite ?

Pour prendre un exemple concret, il paraît totalement justifié qu'un expert en vis et boulons, à qui on demande de produire une classification de ses objets préférés, pose la question suivante : « Oui, mais laquelle ? ». De même, si l'expert en produit une sans s'être posé cette question, il est fort opportun de lui poser la question suivante : « Bien, mais pourquoi celle-là ? ». Comment répondre à ces deux questions autrement qu'en produisant soit une procédure de construction de la classification, acceptable dans le domaine concerné, soit un critère explicite du choix de la bonne classification ? On retrouve ici les deux grandes catégories d'approches de la classification *stricto sensu* : d'une part l'approche dite « constructive » où l'algorithme découvre une structure classificatoire dont les propriétés n'étaient pas spécifiées à l'avance, et d'autre part une approche « par optimisation » où, à partir d'une mesure objective de la qualité du résultat, déclarée explicitement *a priori*, l'algorithme a pour tâche d'explorer l'espace des structures admises pour tenter d'y découvrir la meilleure. On comparera avec profit ces deux approches avec les méthodes proposées par le génie logiciel pour la conception des hiérarchies de classes (voir *chapitre 4*.

Avoir recours à un algorithme optimal de classification, garantissant à tout coup l'obtention du meilleur résultat et relevant d'une approche « par optimisation », présente comme avantages, non seulement que l'obtention d'un résultat exact est garanti, mais aussi et surtout que la pertinence de l'algorithme pour un problème donné ne dépend que de la définition du critère à optimiser et non pas des détails de la procédure d'optimisation. Du point de vue épistémologique, cette approche aboutit à séparer de manière non ambiguë ce qui relève du domaine d'application (description des données initiales, spécification du type de structure classificatoire, recherche et spécification du critère à optimiser) de ce qui relève de l'informatique (l'exploration de l'espace des solutions possibles et l'étude des propriétés de l'algorithme d'exploration).

L'approche constructive, comme celle qui a présidé à la conception des premières méthodes de classification ascendante hiérarchique [Gordon, 1987] [Celeux *et al.*, 1989], ne doit bien sûr pas être rejetée par principe. Elle peut trouver sa pertinence si l'on dispose d'arguments forts pour considérer la procédure de construction comme une simulation d'un processus naturel. Ces vingt dernières années, durant l'émergence d'une nouvelle méthode en classification des êtres vivants, l'*analyse cladistique* (voir § 15.3.2), de nombreux biologistes ont opposés l'approche constructive et l'approche par optimisation de la classification. Cette opposition a abouti en particulier à un rejet quasi total des algorithmes de classification ascendante hiérarchique, pourtant nés en biologie [Sokal et Sneath, 1963].

La recherche d'une justification sémantique aux représentations des connaissances n'est pas neuve. En effet, un des apports majeurs des travaux sur les logiques de descriptions, initiés par KL-ONE (voir *chapitre 11*), a été la mise en évidence qu'un système de représentation des connaissances doit proposer plus que des structures de données dont la sémantique est seulement définie de manière interne par les traitements réalisables (c'est-à-dire seulement définie du point de vue de la manière de « représenter en machine ») mais doit surtout être munie une interprétation externe (c'est-à-dire du point de vue de ce qui doit être représenté) [Woods et Schmolze, 1992] (voir *chapitre 12*). Cela a en particulier l'avantage d'aboutir à une claire différenciation entre le support des connaissances et les connaissances elles-mêmes. Effectivement, sans faire cette distinction, on peut se demander de quoi la base de connaissances est une représentation. Mais cela ne semble pas encore suffisant car, sans préciser la question dont la base de connaissances est la réponse et notamment ce qui justifie le choix de telle ou telle structure classificatoire, les représentations du monde seront toujours sous-déterminées et les traitements automatisables limités de ce fait, faute de pouvoir tirer parti de toutes les particularités de ce qui est à représenter (voir *chapitre 10*).

Contrairement à ce qui se passe souvent dans les applications concrètes des idées de la représentation des connaissances, la recherche d'une justification à la production de telle ou telle classification précise est une préoccupation majeure des systématiciens depuis plus de deux siècles. Par exemple, fortement impliqué dans ces questions, de Candolle a introduit très tôt le terme de *taxonomie* (actuellement taxinomie) pour désigner la théorie des classifications [Candolle, 1813].

À l'heure actuelle la taxinomie est une partie importante de la systématique. En effet, sans ce garde-fou méthodologique, comment empêcher chaque observateur de la Nature de proposer sa propre organisation du monde vivant. De même, comment prétendre que la systématique est une activité scientifique sans que ce qui est recherché par cette discipline ne soit explicitement spécifié [Tassy, 1986]. L'attitude actuelle en biologie semble être bien décrite par une tendance à ne pas perdre le fil de la pertinence, au moins empiriquement, depuis la collecte des observations, la construction des connaissances à partir de celles-ci et enfin l'emploi rationnel de ces dernières, et ce, même si ces trois étapes sont effectuées en des temps différents par des personnes différentes.

Lors de la réalisation d'un système d'information, un moyen souvent employé pour se libérer de la nécessité de justifier d'où proviennent précisément les connaissances est de se reposer uniquement sur l'expert. Mais cela ne peut être suffisant en biologie car il est exigé par le milieu scientifique que les experts justifient leur vision sur l'organisation de la Nature par des arguments rationnels. L'informatisation en biologie n'a pas pour but de construire une représentation informatique des représentations mentales d'un expert mais une représentation informatique de la Nature grâce à son observation et éventuellement au travers de travaux d'experts l'ayant déjà observée. Le concept d'oiseau vu par un psychologue est très différent de celui vu par un systématicien.

## 15.2 Les pratiques de la classification *stricto sensu*

Il découle de ce qui précède que la construction de structures classificatoires, quel qu'en soit le moyen, est un aspect crucial de la pratique scientifique. Cela justifie l'intérêt tout particulier porté aux algorithmes de classification *stricto sensu*, non seulement parce qu'ils automatisent la construction des classifications, mais aussi parce que leur emploi raisonné exige d'expliciter les propriétés des résultats recherchés.

Bien entendu, comme les autres scientifiques, les biologistes se sont intéressés aux algorithmes de classification mais avec deux particularités :

1. la classification des êtres vivants (en tant que structure) étant un des fondements de toute activité scientifique en biologie, une de ses disciplines s'y consacre entièrement : la systématique ;
2. la classification des êtres vivants ayant d'immédiates implications législatives, pharmacologiques, sociologiques, commerciales, religieuses, médicales ... les débats sur ce qu'elle doit être et sur comment l'obtenir font rage depuis plusieurs siècles.

En effet, depuis l'antiquité, les principes par lesquels les classifications du monde vivant ont été construites ont beaucoup changé et l'on doit reconnaître deux transitions fondamentales dans l'histoire des idées :

1. le passage de la recherche de classifications nommées actuellement « artificielles » ayant pour unique but l'organisation des connaissances, à la recherche de classifications dites « naturelles » devant exprimer l'ordre caché de la Nature ;
2. l'obtention d'un consensus quasi universel sur ce qu'est cet ordre naturel caché : l'unique histoire évolutive commune à tous les êtres vivants. Ainsi, les classifications biologiques se doivent maintenant d'être *phylogénétiques*, c'est-à-dire fondées sur nos connaissances du processus classificatoire naturel[2] qui construit la biodiversité : *l'évolution des êtres vivants*.

Ce raccourci simplifié à l'extrême de ces siècles d'histoire montre que l'on peut faire appel à la classification pour des objectifs très différents ; il se rajoute donc aux problèmes terminologiques évoqués précédemment pour expliquer les difficultés à la compréhension de ce qu'est la classification, une autre source de confusion née de la variété de ses objectifs.

Parfois on est amené à construire une structure classificatoire seulement dans le but d'en décréter l'existence. Dans ce cas, la présence d'un processus classificatoire naturel est indifférent car on cherche à imposer au monde une structuration, le plus souvent pour des raisons utilitaires. Par exemple en biologie, on peut vouloir chercher une hiérarchie sur un ensemble d'espèces minimisant le nombre de questions nécessaires à leur identification.

2. Le terme de naturel est ici utilisé comme antonyme d'artificiel et non pas comme référence à la Nature.

| Buts | Questions | Présupposés |
|---|---|---|
| Décret d'une classification utile | Quelle est la structure classificatoire la plus utile ? | L'existence d'un processus classificatoire naturel est indifférent |
| Analyse exploratoire | Quels sont les arguments à l'existence d'une structure classificatoire | L'existence d'un processus classificatoire naturel est une hypothèse à tester |
| Découverte d'une classification préexistante | Quelle est *la* structure classificatoire | On a la certitude de l'existence d'un processus classificatoire naturel |

TAB. 15.2: *Les trois circonstances aboutissant à la création d'une classification. Si cette présentation des trois buts de la classification couvre exhaustivement l'ensemble des pratiques de la classification stricto sensu en biologie, elle donne néanmoins une vision plus tranchée que la réalité. En effet, un même scientifique peut passer d'un point de vue à l'autre au cours de son travail.*

Dans d'autres catégories de problèmes, les algorithmes de classification sont exploités pour rechercher des arguments indiquant l'existence d'une structure classificatoire naturelle, hypothèse que l'on veut tester. Cette direction se retrouve en systématique dite « alpha » où la structure classificatoire produite, en servant de support à la recherche *a posteriori* de propriétés caractérisant les classes découvertes, permet de discuter l'existence d'une ou plusieurs espèces au sein d'un ensemble de spécimens. Par exemple, après avoir trouvé un ensemble de spécimens de mollusques dans une zone montagneuse peu explorée, il est légitime de se demander s'ils appartiennent ou non à la même espèce. Un moyen, parfois employé, est de calculer une partition de ces spécimens sur la base de la description de leur morphologie. Dans un deuxième temps, on essaiera d'interpréter les classes obtenues en recherchant des caractères pouvant avérer l'appartenance des spécimens de classes différentes à des espèces différentes.

Enfin les algorithmes de classification sont parfois employés pour la recherche d'une classification préexistante au regard de l'homme et qui aurait été construite par un processus classificatoire naturel dont on ne doute pas de l'existence. Ce type de but est relativement caractéristique de la reconstruction phylogénétique, où la certitude de l'existence d'une cladogénèse[3] faisant naître les groupes d'êtres vivants les uns des autres dans l'histoire, est le fondement des méthodes, et où l'existence d'une structure classificatoire unique à découvrir à partir des traces qu'elle laisse dans la Nature n'est pas remise en cause.

La théorie de l'évolution formant l'ossature de toute la biologie moderne, les travaux actuels en classification des êtres vivants relèvent quasiment tous de la troisième démarche : la découverte d'une classification préexistante. Même du temps de Karl Linné où l'évolution des êtres vivants était inconnue, la classification biologique, qui voulait être une démarche scientifique objective, relevait déjà de cette troisième démarche. En effet, là où un systématicien moderne recherche l'ordre naturel créé durant l'histoire par l'évolution, on cherchait à cette époque à découvrir

3. Processus aboutissant à l'apparition de deux espèces d'êtres vivants à partir d'une espèce préexistante.

l'ordre « surnaturel » conçu par le créateur. Mais dans les deux cas, la classification à obtenir s'impose à l'homme et ne peut être qualifiée d'utilitaire.

Les tâches de la systématique qui doit traiter de millions d'espèces sont gigantesques. Il faut donc garder à l'esprit que, malgré la clarification du but que doit atteindre la classification biologique, la classification de la plupart des groupes d'êtres vivants n'a pas été encore reconsidérée à la lumière de ces idées relativement récentes à l'échelle de l'histoire de la biologie. Ainsi, bien des classifications actuelles ne s'insèrent pas dans cette démarche évolutive, mais à terme toutes les classifications biologiques seront phylogénétiques réalisant ainsi l'espoir formulé par Charles Darwin il y a plus d'un siècle. Enfin, il faut remarquer qu'une classification n'est pas dite phylogénétique parce qu'elle est le reflet de l'évolution, mais seulement parce qu'elle cherche à l'être. Cela met encore plus l'accent sur le fait que le moyen de produire les classifications est au moins aussi important que la structure produite pour caractériser les traitements pertinents à faire subir aux classifications.

Un même algorithme peut parfois être utilisé pour les trois buts d'une démarche classificatoire. Par exemple un algorithme construisant une structure classificatoire rapprochant les objets les plus semblables et éloignant les plus différents, peut être utilisé soit parce que la structuration de la ressemblance peut être le but utilitaire explicitement poursuivi soit parce qu'il existe de bonnes raisons de penser que la mesure de ressemblance employée sera un bon guide vers la structure classificatoire naturelle que l'on soit sûr de son existence ou que l'on veuille la tester. Cela a été et est encore une source de confusions importantes à propos de la place des algorithmes de classification en biologie.

## 15.3 Classification et biologie

En biologie, le terme de classification est employé pour désigner diverses structures classificatoires spécialisées ayant des sujets et des buts très différents. On ne parlera ici que de la classification des êtres vivants eux-mêmes et non pas des classifications de leurs organes, de leurs associations, de leurs comportements, de leurs composants biochimiques . . .

On trouve en systématique des *classifications nomenclaturales*, qui sont des structures classificatoires dont les classes sont étiquetées par de simples noms et qui sont utilisées pour la représentation de la nomenclature des êtres vivants en dehors de toutes interprétations intensionnelles. Ce système de dénomination conçu par les systématiciens est intéressant à comparer avec ceux nés de l'informatique.

On trouve aussi plusieurs formes de *classifications attributives*, ensembles de caractérisations de classes emboîtées. On les rencontre notamment en reconstruction phylogénétique en tant que résultats de méthodes qui cherchent à proposer des hypothèses sur des relations de parenté historique entre des êtres vivants.

On rencontre aussi d'autres classification attributives, les clés d'identification, qui sont construites en systématique pour aider à l'identification, c'est-à-dire dans ce cas, à l'affectation d'un spécimen à un groupe d'êtres vivants préalablement établi. Il est à noter, comme on le verra plus loin, que ces exemples de classifications attributives ne sont pas toujours désignés par le terme de classification en systématique.

Les graphes d'héritage ou hiérarchies de concepts des bases de connaissances à objets étant des formes particulières de structures classificatoires attributives, la comparaison avec les classifications attributives en biologie est aussi particulièrement intéressante.

Pour chacune des trois formes de classification, le type de pratique de la classification auquel elle correspond, ainsi que la démarche qui permet de la justifier aux yeux des systématiciens sera précisée.

### 15.3.1 Classifications nomenclaturales

De tout temps et dans tous les pays, des noms communs ont été donnés à des groupes d'êtres vivants remarqués pour leurs propriétés. Mais ces noms populaires, qui ont leur utilité en tant que système pratique de référence locale, ne peuvent être la base d'une communication scientifique internationale. Cette dernière impose en effet que le système de référence soit global et accepté de tous. Il sera évident à tout informaticien que la création de principes de dénomination efficaces, robustes et extensibles est un préalable à toute forme de communication. Ainsi, une nomenclature biologique internationale[4] ayant pour but de normaliser la création des noms désignant les groupes d'êtres vivants et de légiférer sur leur usage a été créée dès le milieu du siècle dernier [Ride et Younes, 1986].

Parmi les conventions les plus simples de dénomination en biologie, on notera que tous les noms doivent être latinisés et que les noms des espèces doivent être binominaux[5] et suivis du nom d'auteur. Ainsi l'ancolie de nos jardins doit se nommer et se noter *Aquilegia vulgaris* L. ce qui pourrait se traduire par « ancolie commune selon Linné ». Les codes de nomenclature contiennent de nombreuses autres conventions dont il est important de comprendre qu'elles ne sont pas nées d'un désir outrancier de normalisation mais qu'elles répondent au besoin pratique de délimiter ce qui tombe ou ne tombe pas sous le coup de la nomenclature internationale. Cela permet de savoir si un nom est « scientifique » ou non, selon qu'il suit le code en vigueur.

Le choix d'une langue morte comme langue véhiculaire résulte d'une volonté de maintenir à tout prix la communication universelle entre les peuples. Elle évite d'avoir à choisir les noms scientifiques parmi la multitude de noms vernaculaires. En effet, si par exemple on avait retenu le français, qui peut savoir à l'étranger, et même chez les botanistes, que cette même ancolie est nommée en France selon les régions « Gants de Notre-Dame », « Cornette », « Colombine », « Bonne-Dame », « Aiglantine », « Cinq-Doigts » ... et comment un botaniste français pourrait-il connaître les noms communs en flamand, entre autres, « Akelei » et « Kapelleken ».

La dénomination binominale relève d'un principe ancien qui a été généralisé par Karl Linné dans son « *Systema Naturae* » publié jusqu'en 1758. Le premier nom, qui est un substantif commençant par une majuscule, est le nom du groupe incluant

4. Il existe quatre codes internationaux de nomenclature, un pour la zoologie, un pour la botanique, un pour la bactériologie et un pour la virologie.
5. C'est-à-dire composé de deux parties.

l'espèce : son genre. Le deuxième nom est un adjectif épithète identifiant chaque espèce du genre.

Il est important de faire suivre les noms de groupes biologiques par le ou les noms de leurs auteurs initiaux car les cas d'homonymie, où un même nom a été donné à des espèces différentes, ne sont pas rares. Il existe même des circonstances amusantes, et bien entendu gérées par les codes, où cela n'est pas suffisant car un même auteur a parfois produit des homonymes à vingt ans d'écart.

Cette construction des noms de groupes d'êtres vivants, bien qu'évoquant irrésistiblement le principe aristotélicien du *genus et differentia* et bien que se basant étymologiquement souvent sur des propriétés remarquables des groupes nommés, ne doit en aucun cas faire prendre les noms eux-mêmes pour des tentatives de définition. Cette idée est importante et souvent mal comprise, parfois même à l'intérieur de la biologie. Par exemple le nom Carnivora, qui désigne actuellement les ours, les chats, les chiens, les loutres, les ratons laveurs, les genettes etc., semble devoir désigner en fait tous les êtres vivants qui mangent de la viande et rien qu'eux. Mais les choses ne sont pas si simples car :

1. il n'est utilisé que pour les animaux, donc les plantes carnivores ne sont pas des Carnivora ;
2. de nombreux animaux sont carnivores sans être dans ce groupe, par exemple la plupart des requins ou, dans les mammifères, le chat marsupial *Dasyurus maculatus* ;
3. il existe des animaux dans ce groupe qui ne sont pas carnivores comme par exemple le panda géant *Ailuropoda melanoleuca* qui mange des bambous. La raison de ce regroupement repose sur la théorie de la classification employée en systématique et dont les grandes lignes sont présentées dans le chapitre prochain (voir § 15.3.2).

On pourrait multiplier les exemples de ce type à loisir et il faut retenir que de manière générale, les noms des groupes d'êtres vivants ne doivent pas être pris au pied de la lettre pour des attributs. Les noms sont et ne doivent être que de simples identificateurs, n'évoquant des propriétés des êtres vivants que pour éventuellement faciliter leur mémorisation.

Parmi les caractéristiques de la « démarche objet », on retient souvent le fait que le système doit être capable de maintenir seul l'identité des objets [Delobel *et al.*, 1991 ; Khoshafian, 1991 ; Booch, 1994a] et (voir *chapitre 5*). Idéalement, les données représentant l'identité, que se soient des identificateurs, des curseurs, des pointeurs ou des clés, devraient être totalement invisibles à l'utilisateur. Hélas, en systématique, faute de pouvoir s'appuyer sur un traitement automatique, les scientifiques doivent préserver eux-mêmes l'identité des groupes d'êtres vivants et donc gérer les noms qui identifient ces groupes. La création de ce système universel de dénomination est une tâche gigantesque qui est réalisée manuellement depuis près de trois siècles. À titre d'exemple chiffré, environ 2 millions d'espèces (= les éléments de l'espace des signifiés) sont actuellement acceptés par les systématiciens, chaque espèce avec, en théorie, un unique nom valide (= les éléments de l'espace des signifiants). Malheureusement, les difficultés de communication ont fait que plusieurs dizaines de millions de noms ont été créés dans l'histoire. Maintenir la cohérence du système est à la fois crucial, car les noms sont les principaux points d'accès à la connaissance en biologie, et problématique, du fait de l'ampleur du travail à accomplir. La nomenclature biologique est donc une partie importante de la systématique.

Les principaux moyens de normalisation en nomenclature sont l'usage des « types porte-nom » et du « principe de priorité » [Jeffrey, 1973]. À chaque nom d'espèce est associé de manière permanente un et un seul objet, son *holotype* déposé dans un musée. Il s'agit ici bien sûr d'un objet physique, le plus souvent un spécimen complet mais pas nécessairement.

Si deux espèces ont été proposées dans l'histoire, chacune avec un nom et un type différent, et qu'un scientifique estime plus tard que le type de l'une appartient à l'autre espèce, alors les deux noms sont déclarés synonymes et, selon le principe de priorité, *seul le plus ancien* doit être retenu. Ce principe est un important garde-fou à la prolifération des noms, qui a l'avantage de conduire à une plus grande stabilité de la nomenclature en empêchant un scientifique de changer un nom *a posteriori* parce qu'il ne lui plaît pas. Par incompréhension, on regrette parfois en biologie, hors du domaine de la systématique, la stricte application du principe de priorité car il aboutit parfois à la résurgence de noms anciens oubliés, se substituant à des noms d'usage courant mais plus récents. Mais c'est un faible prix à payer pour l'importante stabilité qu'il introduit en nomenclature.

De même, les nombreuses autres règles de la nomenclature, parfois très complexes, ont pour but principal cette stabilité. Ainsi, pour ne prendre que cette mesure de complexité, le dernier code international de nomenclature zoologique (ICZN) comporte plusieurs centaines de pages consacrées principalement à des définitions de concepts nomenclaturaux et à des énoncés de règles dont on ne peut maîtriser les détails sans comprendre l'esprit général : stabiliser les noms sans les fixer.

Afin de préserver cette stabilité, le nom d'un groupe d'êtres vivants est en nomenclature un simple identificateur pour un type déposé, à qui on ne demande même pas d'être représentatif du groupe. En particulier, les types en systématique ne doivent en aucun cas être considérés comme des prototypes au sens des psychologues (voir *chapitre 8*), c'est-à-dire des instances spéciales auxquelles on pense avant les autres. Par exemple le type associé au nom *Homo sapiens* est le crâne de Karl Linné lui-même.

Tous les noms de la nomenclature biologique sont organisés en une structure de préordre total ; chaque classe de ce préordre est une *catégorie taxinomique*. Parmi les catégories les plus connues on trouve l'espèce, le genre, la famille et l'ordre. Mais il existe aussi de nombreuses autres catégories intermédiaires comme par exemple le sous-genre entre l'espèce et le genre, la tribu entre le genre et la famille ou l'infra-ordre entre la famille et l'ordre ... Ainsi le nom *Aquilegia vulgaris* L. appartient à la catégorie espèce et le nom *Aquilegia* à la catégorie genre. L'ancolie appartient à une famille qui contient aussi les renoncules, et dont le nom scientifique est : Ranunculaceae.

À chaque nom correspond une signification, l'ensemble des êtres vivants qu'il désigne, le *taxon*. Un taxon est un concept classificatoire au sens de Rudolph Carnap [Carnap, 1973], ainsi le nom *Aquilegia vulgaris* L. ne désigne pas seulement les ancolies qui ont été vues par un botaniste mais bien tous les spécimens d'ancolies possibles de cette espèce ; ceux d'hier et de demain et ceux d'ici et d'ailleurs. Ce sont donc des généralisations idéales correspondant aux substances secondes selon Aristote [Lebbe, 1991], fondées sur des observations concrètes d'objets naturels, les substances premières. Ainsi, bien que ce point fasse encore l'objet de débats difficiles [Cracraft, 1983], les taxons sont considérés en biologie à la fois comme des entités individuelles ayant une réalité biologique, et comme des classes d'entités biologiques réelles. Comme dans de nombreux autres domaines, la distinction entre les noms et le sens des noms est très importante en systématique. En effet, un point souvent mal compris est que la nomenclature biologique ne cherche à traiter que des noms et de leurs rapports, et non de leur sens. Par exemple, si un même nom désigne, selon les opinions de deux scientifiques, deux groupes différents mais incluant le type associé à ce nom, il est évident qu'il y a un problème. Mais il n'est pas considéré comme relevant de la nomenclature. La justification à cela est qu'il est nécessaire, vue l'énorme quantité de noms par nature universels dans le temps et dans l'espace, de normaliser l'attribution des noms aux groupes d'êtres vivants sans pour autant interférer avec la liberté de jugement des scientifiques ou contraindre les possibilités d'évolution des connaissances.

Parmi les diverses causes de changement en nomenclature biologique, il faut distinguer deux cas. Parfois les modifications résultent d'un changement d'opinion scientifique sur les taxons eux-mêmes, ce qui sera l'objet du chapitre suivant, la nomenclature change alors, car ce quelle doit représenter change. On parle de changement taxinomique, et ce même si la nomenclature change aussi. Parfois seul le « monde des noms » subit des changements, par exemple par l'application des règles de la nomenclature. Dans ce cas on parle de changement nomenclatural. Ainsi, bien que pour un biologiste les objets importants soient les groupes d'êtres vivants, du fait que les noms pour les désigner ont leurs propres principes de cohérence contraints mais non totalement déterminés par ceux des taxons, s'est développée en systématique une sensible réification des noms eux-mêmes.

On remarquera que l'on est bien loin des procédures de maintien de l'identité dans les représentations par objets, où tout est fait pour que les identificateurs ne soient pas des objets et que les signifiés soient en bijection avec les signifiants. À l'évidence, ce dernier objectif n'est pas atteint en systématique, principalement pour des raisons pratiques, la diffusion des connaissances étant limitée aux média classiques (papier et communication orale ... ), mais aussi pour des raisons plus intrinsèques, les méthodes utilisées en nomenclature pour éviter la prolifération des noms entraînant une inévitable réutilisation de noms ayant changés de sens. Il ne faudrait surtout pas voir là une incohérence fondamentale dans la pratique de la systématique, mais plutôt une tentative, optimisée par trois siècles de pratique, de rendre compatibles plusieurs points de vues partiellement contradictoires. En effet, le point de vue de l'utilisateur, qui exige la plus totale stabilité des noms et de leur sens [6], est par principe différent de celui du systématicien qui tient à pouvoir ajuster à loisir le sens des noms pour les rendre les plus adéquats possible à la Nature. Le

6. Il suffit de penser au législateur ayant inclus un nom taxinomique dans un texte de loi.

vif débat autour du projet *Species 2000* qui vise à créer une base de données répartie accessible sur le *Web*[7] et contenant tous les noms d'espèces en est l'écho.

La situation est encore plus complexe, car la séparation entre les noms et leur sens n'est pas toujours, en pratique, aussi complète qu'on pourrait le souhaiter. Ainsi, aussi étrange que cela puisse paraître, le terme de « classification des êtres vivants » ne désigne pas seulement une structure classificatoire représentant les relations d'inclusion d'un ensemble de taxons mais une notion plus intriquée avec des préoccupations nomenclaturales que nous nommerons *classification linnéenne*. À titre d'illustration, si un systématicien associe aux taxons d'une classification des noms de catégories taxinomiques différentes des initiales, par exemple si ce qui était un sous-genre devient un genre, etc., il sera considéré comme ayant proposé une nouvelle classification, et ce, quand bien même l'extension de chaque groupe n'aura pas changé d'un iota.

Cette importance des catégories taxinomiques dans les classifications biologiques se retrouve aussi dans la notion de *taxon monotypique*, c'est-à-dire un taxon auquel correspond un seul taxon de catégorie inférieure. Par exemple, si la classe des Mammifères contient plusieurs ordres, l'ordre des Pholidotes, qui est l'un de ceux-ci, ne contient qu'une famille, celle des Manidés, contenant elle-même les sept espèces actuelles de pangolins[8]. On dit que l'ordre des Pholidotes est un taxon monotypique complètement différent de la famille des Manidés puisque n'appartenant pas aux mêmes catégories, et ce, bien que les extensions de ces taxons soient identiques. Cette apparente contradiction est levée si on garde en tête que la systématique ne traite pas des taxons dans une acception uniquement actuelle. Ainsi l'ordre des Pholidotes inclut, au moins potentiellement, bien d'autres espèces que les sept vivantes dans le temps présent. Malgré cette explication, cet exemple montre notamment que, du fait d'interférence avec les problèmes de nomenclature, l'ensemble des taxons d'une classification linnéenne ne forme pas nécessairement une hiérarchie sur les êtres vivants connus[9].

Aller plus loin dans la description des rapports entre la classification en biologie et la nomenclature est difficile pour au moins deux raisons. La première est que rentrer dans les détails nécessite d'exposer ce qui ne relève pas nécessairement d'un consensus entre tous les systématiciens. La seconde est que, faute d'une formalisation rigoureuse des processus de création et de mise à jour des taxons et de leurs noms, il serait très difficile de parler autrement qu'avec des exemples précis obligeant à rentrer trop profondément dans la biologie. Il paraît néanmoins évident que les nombreux projets internationaux d'informatisation en systématique ne pourront que bénéficier d'une telle formalisation qu'il est impossible d'envisager de mettre en œuvre sans la participation des informaticiens. Il est en retour tout aussi évident que l'informatique qui traite des connaissances pourrait trouver ici un formidable champ d'expérience.

---

7. Voir une démonstration partielle à `http://www.sp2000.org/`.

8. Les pangolins sont d'assez gros mammifères africains et asiatiques, étonnants notamment car ils sont recouverts de grosses écailles imbriquées.

9. Pour avoir une idée plus concrète de ce que sont les classifications linnéennes, on consultera avec intérêt les sites *Web* projetant d'indexer les ressources d'*Internet* grâce à ces classifications : « Tree of Life » `http://phylogeny.arizona.edu/tree/phylogeny.html` ou encore « Web Lift to Any Taxon » `http://ucmp1.berkeley.edu/taxaform.html`.

Cette première partie a traité du point de vue nomenclatural de la classification des êtres vivants, et c'est parce que les noms taxinomiques ne doivent pas être interprétés comme des attributs que l'on peut parler, dans ce contexte, de classification nomenclaturale. Les liens entre les noms, les sens des noms et les attributs sont maintenus dans d'autres formes de classification : les *classifications attributives*.

### 15.3.2 Classifications attributives et phylogénie

Une définition grossière et provisoire d'une classification phylogénétique peut être la suivante : structure classificatoire étant le reflet de l'histoire de la descendance des êtres vivants. Mais que sont les items à classifier ? Que veut dire « être fidèle à l'histoire des êtres vivants » ? Quel type de structure classificatoire doit-on produire ? Répondre à ces questions demande d'avoir en tête un certain nombre de principes fondamentaux pour la biologie :

- tous les êtres vivants actuels ont une ascendance commune et donc, il existe au moins un ancêtre commun pour toute paire d'êtres vivants ;
- à un temps donné de l'histoire, la fermeture transitive de la relation « être inter-féconds » entre des êtres vivants forme des classes d'équivalence appelées *espèces biologiques* ;
- à chaque espèce dans l'histoire correspond un ensemble d'espèces qui en descendent : un *clade* ;
- réciproquement, pour qu'un ensemble d'espèces soit un clade il faut qu'il contienne l'intégralité des descendants de la plus récente espèce ascendante commune à toutes ces espèces, ce qui évoque la notion d'*idéal* des ensembles ordonnés ;
- l'évolution des êtres vivants procède dans le temps par division: ainsi, l'ensemble des clades forme une hiérarchie sur les espèces terminales, les items à classifier.

Bien sûr, ces principes comme tout ce qui relève de la biologie n'ont pas valeur de loi. Ils ont tous des limites importantes, au moins théoriques, à leur application générale. Par exemple, si le fait que tous les êtres vivants actuels utilisent le même code génétique (comme toujours à de très rares exceptions près) ne fait pas douter qu'ils descendent tous d'un même ancêtre commun, rien ne dit que la vie ne soit pas apparue plusieurs fois sur la terre sous une forme inconnue n'ayant pas donné de descendants actuels.

La réalité de l'existence d'une partition naturelle des êtres vivants pose aussi problème. La notion d'espèce biologique est encore discutée par les biologistes, non seulement dans ses applications mais aussi dans ses fondements mêmes. Certains, rares, vont jusqu'à rejeter l'existence objective de tout groupe d'êtres vivants. Mais malgré ces difficultés, à une certaine échelle de différenciation, ce principe affirmant la réalité de séparation d'entités naturelles objectives est pourtant facilement vérifié. En effet, si deux groupes d'êtres vivants ne sont pas inter-féconds, leur isolement évolutif est acquis car il est impossible que les innovations génétiques apparaissant dans l'un des groupes puissent être transmises à l'autre. Cela est évidemment le cas entre, par exemple, une population de soles et une population de babouins.

La notion d'espèce dans le temps est aussi difficile : quand commence une espèce ? quand finit-elle ? De nombreux systématiciens, probablement maintenant majoritaires, rejetant le fait qu'une espèce puisse naître d'une autre sans se séparer d'une troisième et qu'une espèce puisse survivre après avoir donnée une espèce fille, s'accordent sur le fait qu'une espèce ne naît que lors de sa séparation d'une espèce sœur et ne meure que quand elle se divise en deux espèces filles.

Une vision strictement hiérarchique de l'évolution est, de même, approximative du fait de l'existence avérée d'hybridation pouvant mélanger les originalités génétiques apparues au sein de deux ensembles d'organismes semblant pourtant bien séparés. Mais là encore, au delà d'une certaine échelle de temps, l'approximation est sans doute très fidèle à la réalité.

Quelles que soient les difficultés de mise en œuvre de ces principes, le processus évolutif formant par divisions successives une hiérarchie, le sens de « être fidèle à l'histoire des êtres vivants » devient clair : il ne faut accepter comme fidèles à l'histoire des êtres vivants que des structures classificatoires dont les classes existent dans la hiérarchie naturelle, c'est-à-dire qui soient des clades. Cette décision, qui pourrait sembler aller de soi, a des conséquences importantes et est la cause de nombreux bouleversements de la classification des êtres vivants. Par exemple, comme il est établi que le plus proche ancêtre commun à tous les reptiles connus, comme les serpents, les lézards, les tortues, les crocodiles, pour ne parler que de quelques groupes actuels, a pour descendant non seulement les reptiles mais aussi tous les oiseaux, il faut logiquement :

- soit rejeter le taxon reptile des classifications biologiques car ce n'est pas un clade ;
- soit inclure le taxon oiseaux dans le taxon reptile et donc rejeter l'acception actuelle de ce dernier.

Il en est de même pour de nombreux autres groupes qui sont dits *paraphylétiques*, comme les poissons ou les vers, pourtant utilisés communément même par des biologistes. Là encore, le consensus en systématique n'est pas complètement atteint, car il est à noter qu'un courant, maintenant minoritaire, existe et rejette l'idée que seuls les clades doivent être retenus dans les classifications biologiques.

Il peut sembler étonnant à un non biologiste que tous ces principes de base soient autant discutés, mais d'une part, un accord est difficile à atteindre car, la Nature étant vaste et diverse, chaque systématicien ne peut en embrasser qu'une petite part ce qui biaise son jugement et, d'autre part, la Nature est rétive et ne se laisse pas facilement couler dans un moule universel. Ces principes, aussi discutés soient-ils, nous permettent maintenant de fournir une définition plus précise de la classification phylogénétique : classification hiérarchique d'un ensemble de clades et dont les classes sont des clades [10].

10. Cette définition est évidemment totalement contingente aux pratiques majoritaires de la systématique actuelle et ne préjuge pas de l'évolution future des idées.

Mais pourquoi les biologistes veulent-il des classifications phylogénétiques ? Tout d'abord connaître l'histoire a en soi un intérêt majeur. C'est cette histoire qui donne toute la cohérence à notre vision du monde vivant. C'est de sa connaissance détaillée que l'on pourra vraisemblablement tester nos idées sur la nature des processus évolutifs. De plus, parmi toutes les formes envisageables de classification, les classifications phylogénétiques sont celles qui ont le plus de chance d'être prédictives de propriétés inconnues car elle s'appuient sur une structure inhérente à la Nature elle-même.

La notion de classification prédictive fait l'objet de discussions abondantes en systématique [Archie, 1984 ; Colless, 1986], en statistique [Gower, 1974] et en épistémologie [Gilmour, 1951], mais tous les points de vue tournent autour de l'idée que plus on peut prédire de propriétés des êtres vivants à partir de la seule connaissance de leur appartenance aux classes d'une classification, meilleure est cette dernière. Tenter d'être fidèle au processus même de production de la biodiversité est un moyen crédible aux yeux de nombreux biologistes d'obtenir des classifications prédictives. Enfin, ce que la systématique gagne fondamentalement à rechercher des classifications phylogénétiques, c'est la spécification du but à atteindre : il devient ainsi possible de justifier les classifications. En effet, avant cette orientation, on pouvait parler de classification acceptée ou non, maintenant on peut parler, au moins en théorie et dans les limites de notre savoir, de classification vraie ou fausse.

Ce dernier point explique pourquoi les systématiciens mettent en avant les clades, car il veulent traiter de groupes ayant une existence objective, indépendante de notre regard et des méthodes pour les reconnaître. En suivant ce principe, la connaissance de l'histoire des êtres vivants détermine complètement les classes susceptibles d'être retenues dans nos structures classificatoires.

Mais comment connaître cette histoire ? Il est évidemment impossible de présenter toutes les approches possibles de ce problème [Darlu et Tassy, 1993], mais toutes relèvent à l'évidence du troisième but de la construction d'une classification : « la découverte d'une classification préexistante » (voir page 428) et font en majorité appel à des principes d'optimisation. Donc, quelles mesures de qualité des classifications garantissent pratiquement la fidélité à l'histoire ? Sur la base de quelle représentation des items doit se mesurer cette qualité ? Des principes très divers existent allant depuis la recherche de la classification de vraisemblance maximale dans une modélisation probabiliste markovienne des processus évolutifs [Felsenstein, 1973] jusqu'à des méthodes plus exploratoires formant ce que l'on appelle la *cladistique* [Hennig, 1966 ; Wiley, 1981].

À titre d'illustration, les grandes lignes de l'analyse cladistique d'un ensemble de données artificielles vont être montrées. Une telle analyse est particulièrement intéressante ici car elle montre sur un exemple simple comment il est possible d'obtenir une classification attributive dont les propriétés sont fort éloignées de celles des bases de connaissances par objets (voir *chapitre 10*).

Dans le tableau 15.3, chaque colonne représente un taxon (noté de $a$ à $f$) que l'on supposera être des espèces et chaque ligne un caractère (noté de $C_1$ à $C_{20}$). Le taxon $a$ pourrait par exemple représenter l'homme, le caractère $C_1$ le fait de

| | $a$ | $b$ | $c$ | $d$ | $e$ | $f$ |
|---|---|---|---|---|---|---|
| $C_1$ | 1 | 1 | 1 | 1 | 1 | 0 |
| $C_2$ | 1 | 1 | 1 | 0 | 0 | 0 |
| $C_3$ | 1 | 1 | 1 | 0 | 0 | 0 |
| $C_4$ | 0 | 0 | 0 | 1 | 1 | 0 |
| $C_5$ | 0 | 0 | 0 | 1 | 1 | 0 |
| $C_6$ | 0 | 0 | 0 | 1 | 1 | 0 |
| $C_7$ | 0 | 0 | 0 | 1 | 0 | 0 |
| $C_8$ | 0 | 0 | 0 | 0 | 1 | 0 |
| $C_9$ | 0 | 0 | 0 | 0 | 1 | 0 |
| $C_{10}$ | 1 | 1 | 0 | 0 | 0 | 0 |
| $C_{11}$ | 1 | 1 | 0 | 0 | 0 | 0 |
| $C_{12}$ | 1 | 1 | 0 | 0 | 0 | 0 |
| $C_{13}$ | 1 | 1 | 0 | 0 | 0 | 0 |
| $C_{14}$ | 1 | 0 | 1 | 0 | 0 | 0 |
| $C_{15}$ | 1 | 0 | 1 | 0 | 0 | 0 |
| $C_{16}$ | 1 | 0 | 1 | 0 | 0 | 0 |
| $C_{17}$ | 1 | 0 | 0 | 0 | 0 | 0 |
| $C_{18}$ | 0 | 1 | 0 | 0 | 0 | 0 |
| $C_{19}$ | 0 | 0 | 1 | 0 | 0 | 0 |
| $C_{20}$ | 0 | 0 | 1 | 0 | 0 | 0 |

TAB. 15.3: *Tableau de données artificielles (adapté de [Darlu et Tassy, 1993]). Chaque colonne représente un taxon (noté de $a$ à $f$) et chaque ligne un caractère (noté de $C_1$ à $C_{20}$).*

posséder ou non une clavicule et le 1 à l'intersection des deux, le fait que l'homme en possède une.

Le but de l'analyse cladistique est de former à la fois une hiérarchie sur les taxons initiaux, ou ce qui est équivalent dans ce cas une arborescence (= arbre enraciné ou arbre hiérarchique) appelé *cladogramme*, et un étiquetage des classes par des transformations de caractères. On voit sur la figure 15.1 qu'à la classe $\{d, e\}$ correspondent trois transformations ($C_4 : 0 \to 1$, $C_5 : 0 \to 1$, $C_6 : 0 \to 1$) placées juste en dessous du nœud correspondant à la classe. La transformation $C_4 : 0 \to 1$ doit être interprétée comme : le caractère $C_4$ passe de la valeur 0 qu'il avait dans la super-classe à la valeur 1.

Une transformation de caractère est un événement historique hypothétique dont il faut supposer l'existence pour expliquer les données. Par exemple la description connue du taxon $d$ (le vecteur colonne correspondant du tableau 15.3 peut être reconstitué à partir du cladogramme de la figure 15.1 en appliquant à la description du taxon le plus externe dans la hiérarchie, qui représente le taxon ancestral dont tous les autres dérivent ($f$ dans ce cas, dont la description n'est formée arbitrairement dans cet exemple artificiel que de zéro), les transformations qui étiquettent toutes les classes qui incluent $d$ ($C_1 : 0 \to 1$, $C_4 : 0 \to 1$, $C_5 : 0 \to 1$, $C_6 : 0 \to 1$, $C_7 : 0 \to 1$).

Pour construire la hiérarchie, la cladistique recherchera, parmi toutes les hiérarchies et tous les étiquetages, la classification qui minimisera le nombre de ces transformations de caractère hypothétiques, en suivant en cela le *principe de parcimonie*. Comme la simple utilisation de ce principe ne détermine seulement qu'un arbre particulier, compatible avec plusieurs hiérarchies, le choix final est effectué en considérant un taxon particulier (ou plusieurs) comme *extra-groupe* (le taxon $f$ dans notre exemple) qui enracinera l'arbre et orientera les caractères [Darlu et Tassy, 1993]. Dans l'exemple, il existe deux solutions minimales supposant 23 transformations. Ces deux solutions donnent la même hiérarchie et ne diffèrent donc que par l'étiquetage ; la figure 15.1 représente une de ces solutions.

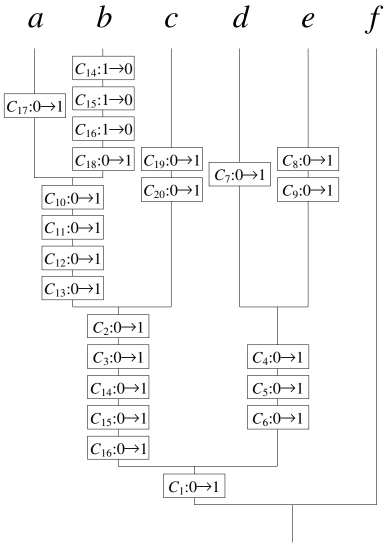



FIG. 15.1: *Un des deux arbres les plus parcimonieux que l'on peut obtenir à partir du tableau 15.3. Dans un tel arbre, il suffit de 23 transformations de caractères (représentées ici par des boîtes) pour expliquer les données.*

Ce qui est intéressant pour les représentations par objets dans ce principe de construction de classification tient principalement à deux points. Premièrement, il montre qu'il est possible d'associer à des classes autre chose que des propriétés nécessaires et suffisantes ou seulement nécessaires (voir *chapitres 10 et 12*), tout en reposant sur des principes rationnels. En effet, on remarquera dans l'exemple que bien que le groupe $\{a, b, c\}$ soit apparu suite, entre autre, à la transformation des caractères ($C_{14} : 0 \rightarrow 1$, $C_{15} : 0 \rightarrow 1$, $C_{16} : 0 \rightarrow 1$), le taxon $b$ possède les trois caractères correspondant ($C_{14}$, $C_{15}$, $C_{16}$) à 0 car la hiérarchie contient les transformations inverses pour passer de $\{a, b\}$ à $\{b\}$.

On voit ainsi que, dans ce type de classification, les propriétés associées aux classes ne sont pas des caractérisations au sens habituel. En effet, ces propriétés caractérisent le groupe concerné seulement au moment de sa différenciation et non toute sa descendance. Deuxièmement, cela montre aussi qu'il est difficile d'envisager des traitements automatiques utilisant un cladogramme comme base d'une représentation de connaissances biologiques sans tenir compte des principes originaux qui permettent de les obtenir, et notamment s'ils sont le résultat d'une optimisation globale. Cela vient en grande partie du fait que la théorie de l'évolution n'est pas une simple théorie superficielle mais au contraire une théorie profonde capable de générer des explications causales testables ; ainsi les systématiciens désirent produire des classifications à l'image de cette théorie, profondément justifiées.

De la manière dont est construit un cladogramme et de ces deux remarques, on conclura qu'un tel arbre ne permet pas de réaliser des identifications par parcours (affectation d'un spécimen à une classe préexistante, voir tableau 15.1). En effet, si la systématique semble avoir hésité longtemps entre la production de classifications linnéennes permettant des identifications aisées (voir § 15.3.3) et la production de classifications naturelles, l'époque actuelle opte clairement pour le deuxième objectif.

Il existe une intéressante analogie entre ces deux points de vue et les deux manières de justifier l'architecture hiérarchique des représentations par d'objets. En effet, cette dernière est parfois présentée, dans une perspective informatique, comme une méthode efficace d'indexation des connaissances et parfois, en se rapprochant de la psychologie cognitive, comme étant une structuration naturelle aux yeux des êtres humains, notamment auprès des experts à qui l'on demande de s'exprimer sous cette forme.

Bien sûr dans les deux domaines, les deux points de vue sont parfaitement acceptables. Mais l'histoire de la systématique montre clairement qu'ils ne peuvent que difficilement coexister dans une même classification. Structurer efficacement les connaissances et être fidèle aux idées sont souvent des buts contradictoires. En effet, si ayant trois groupes d'êtres vivants, on regroupe les deux premiers parce que l'on pense que leur plus proche ancêtre commun est plus récent que le plus proche ancêtre commun aux trois, rien ne garantit en théorie l'existence de propriétés nécessaires et suffisantes du nouveau groupe formé parmi les descripteurs auxquels on s'intéressent. Ainsi, si les biologistes ont décidé qu'une classification phylogénétique est la bonne manière de structurer le monde vivant cela ne conduit pas, de manière inhérente, à une factorisation efficace des connaissances. De manière plus générale, si les groupes nous sont imposés par nos propres principes de classification parce que nous nous conformons à la troisième pratique de la classification « découverte d'une classification préexistante » :

- les caractérisations n'existent pas nécessairement ;
- elles sont à découvrir *a posteriori* ;
- elles ne sont pas uniques.

Dans ce contexte, forcer un expert à s'exprimer en termes de telles caractérisations l'oblige parfois, d'une part à forcer son génie et à donner des caractérisations fausses et, d'autre part, à choisir de manière arbitraire parmi toutes les caractérisations possibles, échappant de la sorte à son obligation scientifique de justification rationnelle. Si on veut que les concepts représentés dans un système informatique aient un sens fort, ils ne peuvent qu'être justifiés au sein d'une théorie intégrative de la classification, nécessairement complexe et spécifique. L'exemple de la systématique montre qu'une telle théorie est longue et difficile à obtenir.

Pourquoi, alors que les biologistes s'accordent sur cette vision historique de la classification, existe-t-il d'une part des classifications linnéennes, au sens défini plus haut, et d'autre part des classifications phylogénétiques ? La raison est qu'un cladogramme n'est pas en soi une classification pour un biologiste, sous entendu qu'il ne donne pas nécessairement une classification linnéenne unique. En effet, d'une part, un systématicien ne souhaitera pas nécessairement nommer toutes les classes de la hiérarchie historique qu'il vient de produire (dans une hiérarchie saturée, il y en a $N - 1$ pour $N$ taxons) ; dans l'exemple, il pourrait choisir de ne pas nommer la classe $\{a, b, c\}$ parce que ce taxon n'a jamais été trouvé dans les différentes études publiées précédemment. D'autre part, il existe de nombreuses manières d'attribuer des catégories aux noms retenus tout en respectant les relations d'inclusion entre les classes. Si les six espèces étudiées forment une famille on peut choisir de faire des taxons $\{a, b, c\}$ et $\{d, e\}$ soit des sous-familles, soit des genres, soit tout autre catégorie sub-familiale. Il est à remarquer que dans ce processus de construction de classifications linnéennes, on devra créer des taxons monotypiques pour traiter le cas de la classe $\{f\}$, s'éloignant ainsi de la structure strictement hiérarchique initiale. En effet, si l'expert choisit de considérer $\{a, b, c\}$ et $\{d, e\}$ comme des genres et $\{a, b\}$ comme un sous-genre il devra aussi associer à $\{f\}$, à la fois un nom de genre et un nom de sous-genre.

Ce qu'il est important de remarquer dans ce processus, c'est que si les systématiciens ont développé des théories et des pratiques pour l'obtention d'hypothèses phylogénétiques profondément justifiées, et que si un consensus général a été obtenu sur le fait qu'il faut baser les classifications linnéennes sur ces hypothèses pour évacuer autant que faire se peut la subjectivité, par contre la manière dont sont produites ces classifications est fortement empreinte d'arbitraire, notamment dans l'affectation des noms à des catégories.

### 15.3.3 Classifications attributives et clé d'identification

Nous avons vu que les deux formes précédentes de classification des êtres vivants étaient inadaptées à l'identification : la classification linnéenne, car elle est seulement nominative et la classification phylogénétique car ses principes s'y opposent. C'est pourquoi les systématiciens fournissent habituellement dans leurs publications, en plus des deux autres formes de classification, des structures classificatoires attributives particulières, nommées *clés d'identification* (parfois aussi clés de détermination, clés dichotomiques ou clés diagnostiques), et spécialement conçues pour aider à réaliser des identifications.

Ces clés imprimées se présentent comme une suite de questions disposées sous deux formes principales, la forme dite « parallèle » (voir la figure 15.2) et la forme dite « indentée » (voir la figure 15.3). Le principe de leur utilisation est très simple, il suffit de répondre à la première question et, suivant sa réponse, de se faire orienter soit vers une autre question soit vers le résultat de l'identification.

Ces clés sont, et de loin, le moyen le plus couramment utilisé pour réaliser des identifications. Plusieurs millions de ces clés ont été produites à ce jour par les systématiciens selon le groupe taxinomique, la zone géographique, les types de descripteurs et de manière plus générale, selon le contexte d'utilisation. Il existe ainsi une forte culture en biologie de l'usage mais aussi de la production de ces clés car

| | | | |
|---|---|---|---|
| 1:1 | Bractéoles = absentes | ⇒ | 2 |
| 1:2 | Bractéoles = 2 | ⇒ | 3 |
| 1:3 | Bractéoles = plus de 2 | ⇒ | **Genre : *Weddellina*** |
| 2:1 | Longueur de l'anthère des étamines = inférieure à 0.7 | ⇒ | **Genre : *Dalzellia*** |
| 2:2 | Longueur de l'anthère des étamines = supérieure à 1.2 | ⇒ | **Genre : *Indotristicha*** |
| 3:1 | Longueur de l'anthère des étamines = inférieure à 0.4 | ⇒ | **Genre : *Malacotristicha*** |
| 3:2 | Longueur de l'anthère des étamines = 0.4 à 0.7 | ⇒ | 4 |
| 3:3 | Longueur de l'anthère des étamines = 0.8 à 1.2 | ⇒ | **Genre : *Tristicha*** |
| 3:4 | Longueur de l'anthère des étamines = supérieure à 1.2 | ⇒ | **Genre : *Indotristicha*** |
| 4:1 | Hauteur de l'ovaire = inférieure à 1.0 | ⇒ | **Genre : *Dalzellia*** |
| 4:2 | Hauteur de l'ovaire = 1.0 à 2.8 | ⇒ | **Genre : *Tristicha*** |

FIG. 15.2: *Exemple de clé d'identification simple telle qu'elle est habituellement publiée sous la forme dite « parallèle » et concernant les genres des Tristichaceae (tirée de [Vignes, 1996]).*

tous les systématiciens et tous les utilisateurs de la systématique (écologistes, physiologistes, généticiens, étudiants en biologie, naturalistes amateurs, etc.) se sont un jour retrouvés à parcourir des clés d'identification sous une forme imprimée, ne serait-ce que pour faire un bon repas de champignons.

L'origine des clés d'identification en systématique est fort ancienne. La première clé publiée, sous une forme très proche des clés actuelles, se trouve dans la *Flore françoise* de Jean-Baptiste de Monnet, chevalier de Lamarck [Lamarck, 1778]. Depuis, le principe des clés est resté le même.

Des structures équivalentes aux clés ont été produites dans de nombreux autres domaines. Mais si les clés sont anciennes et considérées comme des méthodes traditionnelles en systématique, leur apparition dans ces autres domaines est par contre récente. En médecine, des clés, appelées *algorithmes cliniques*, sont apparues et se sont multipliées depuis une quinzaine d'années [Varma, 1987]. Elles sont perçues comme une méthode moderne de transmission de stratégies diagnostiques ou thérapeutiques et comme d'excellents outils pédagogiques [Margolis, 1983]. Des idées équivalentes sont apparues indépendamment sous le nom d'*arbres de décision* en apprentissage automatique [Quinlan, 1986 ; Mingers, 1989], d'*arbres de segmentation* ou *de discrimination* en analyse des données et en statistiques [Williams et Lambert, 1960 ; Breiman *et al.*, 1984 ; Ciampi *et al.*, 1996] et de *questionnaires* [Picard, 1972] en mathématiques appliquées.

Formellement une clé peut être considérée comme un graphe connexe, orienté, sans circuit et enraciné [Berge, 1970], bénéficiant d'un étiquetage spécifique [Vignes, 1991] :

- Un nœud du graphe est étiqueté par un ou plusieurs *descripteurs* [11] ; il correspond à un numéro de question dans la clé.

11. Le terme de descripteur est utilisé ici pour désigner une formule qui spécifie ce qui doit être décrit. On emploi parfois aussi les termes de propriété, de caractère, de trait ou d'attribut pour le même sens. Mais, l'emploi de ces termes est rejeté ici, car ces derniers sont parfois aussi employés pour dénoter le résultat de la description.

♣ Bractéoles = absentes

♡ Longueur de l'anthère des étamines = inférieure à 0.7 ⇒ **Genre : *Dalzellia***

♡ Longueur de l'anthère des étamines = supérieure à 1.2 ⇒ **Genre : *Indotristicha***

♣ Bractéoles = 2

♡ Longueur de l'anthère des étamines = inférieure à 0.4 ⇒ **Genre : *Malacotristicha***

♡ Longueur de l'anthère des étamines = 0.4 à 0.7

♠ Hauteur de l'ovaire = inférieure à 1.0 ⇒ **Genre : *Dalzellia***

♠ Hauteur de l'ovaire = 1.0 à 2.8 ⇒ **Genre : *Tristicha***

♡ Longueur de l'anthère des étamines = 0.8 à 1.2 ⇒ **Genre : *Tristicha***

♡ Longueur de l'anthère des étamines = supérieure à 1.2 ⇒ **Genre : *Indotristicha***

♣ Bractéoles = plus de 2 ⇒ **Genre : *Weddellina***

FIG. 15.3: *Clé d'identification de la figure 15.2 sous la forme dite « indentée ».*

- Un arc est étiqueté par des ensembles de valeurs du (ou des) descripteur(s) du nœud origine de l'arc ; les arcs partant d'un même nœud correspondent aux différentes alternatives proposées à un numéro de la clé.
- Une feuille du graphe est une des conclusions de la clé, elle est étiquetée par un taxon (plus rarement par plusieurs taxons en cas d'impossibilité de les discriminer).

La figure 15.4 montre le tracé du graphe correspondant aux clés précédentes des figures 15.2 et 15.3. Un parcours en largeur de ce graphe a généré la clé parallèle de la figure 15.2 et un parcours en profondeur à produit celle indentée de la figure 15.3.

La construction d'une clé, qu'elle soit manuelle ou automatique, consiste à partir d'un graphe vide auquel on ajoute, de manière heuristique et pas à pas, des nœuds qui partitionnent l'espace de représentation. Par un mode de raisonnement qui s'apparente à l'abduction, quand une zone de l'espace de représentation n'est accessible que par un seul taxon la construction s'arrête et la feuille obtenue est étiquetée par le taxon restant [Vignes, 1991]. Ce mode de raisonnement, très utilisé en systématique, et qui demande de connaître l'ensemble exhaustif des taxons, semble être une alternative supplémentaire intéressante aux autres modes de classification incertaine utilisés par les représentations par objets (voir *chapitre 12*).

L'utilisation d'une clé revient au parcours du graphe en partant de la racine et en suivant le chemin en accord avec la description du spécimen à identifier. Un seul chemin est parcouru à chaque identification, sauf en cas de doute entre plusieurs arcs partant d'un même nœud (c'est-à-dire de doute au niveau d'une question de la clé), pouvant conduire l'utilisateur à parcourir successivement le chemin correspondant à la première hypothèse dans la description, puis un autre chemin correspondant à une autre hypothèse.

Un chemin maximal du graphe d'identification, c'est-à-dire allant de la racine jusqu'à une feuille, correspond à la description d'un ensemble de spécimens possibles identifiés comme appartenant au taxon indiqué dans la feuille. De manière

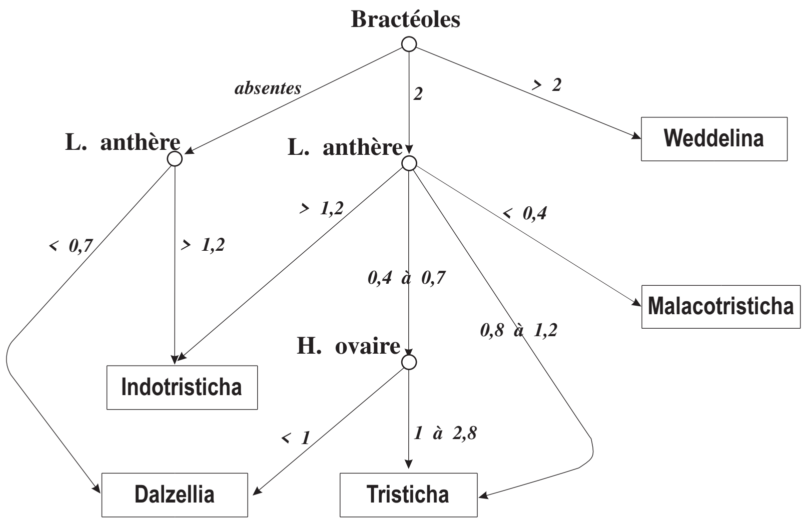



FIG. 15.4: *Schéma du graphe correspondant à la clé de la figure 15.3. Les nœuds sont représentés par un cercle avec le descripteur correspondant au-dessus. Les arcs sont représentés par une flèche, avec les valeurs correspondantes à coté. Les feuilles sont représentées par un rectangle contenant les noms taxinomiques correspondants.*

plus générale, toute clé définit une structure classificatoire sur les êtres vivants, avec des classes définies en intension (voir *chapitre 12*, une pour chaque nœud. On remarquera que cette structure n'est évidemment pas une hiérarchie dans le cas général.

Plusieurs applications informatiques pour l'identification en biologie reposent sur le principe du parcours d'une clé [Kendrick, 1972 ; Hureau, 1994], certaines faisant preuve d'une certaine généricité [Estep *et al.*, 1989]. L'avantage de ces méthodes d'identification réside principalement dans le fait que de nombreuses clés sont disponibles et que leur informatisation directe très aisée peut être l'occasion de les enrichir par des illustrations.

Parmi la multiplicité des nouveaux services apparaissant sur le *World Wide Web*, on trouve ainsi des clés d'identification dont le parcours est assuré par des liens hypertextes. Chaque étape de la clé peut être représentée par un fichier HTML avec pour chaque réponse possible un lien vers un nouveau fichier correspondant au nœud suivant dans la clé ou à un taxon en cas de conclusion de l'identification. Sur ce principe, le *Marine Biology Laboratory* de Woods Hole propose des clés concernant la plupart des invertébrés marins de la côte est de l'Amérique du Nord [12]. Ce site a été produit par un programme traduisant le texte préformatté des clés en un ensemble de fichiers HTML liés entre eux.

Informatiser une clé d'identification existante est certainement le moyen le plus simple de créer un programme d'identification mais n'apporte pas un accès très innovant aux informations taxinomiques [Pankhurst, 1991]. En effet, contrairement

12. Voir `http://www.mbl.edu/html/KEYS/INVERTS/contents.html`.

aux programmes dits d'*identification assistée par ordinateur* [Pankhurst et Aitchinson, 1975 ; Lebbe, 1991], qui permettent de décrire librement le spécimen à identifier, ces programmes de parcours de clés, comme avec les classiques clés sur papier, contraignent fortement la démarche de l'utilisateur [Durrieu, 1996]. Il faut cependant noter que certains programmes autorisent des formes plus sophistiquées de parcours de clés qui seraient irréalisables à la main, comme par exemple le système SYSEX [Gammerman *et al.*, 1986], reposant sur la théorie de la crédibilité de Dempster-Shafer [Schafer, 1976].

La représentation d'une clé est aussi aisée dans la plupart des outils de représentation de connaissances par d'objets. Il suffit de faire correspondre un concept dans la base de connaissances à chaque nœud de la clé. L'avantage de cette utilisation des outils informatiques à objets est de pouvoir reposer, durant l'informatisation, sur des procédures bien comprises d'acquisition de bases de connaissances et, durant l'utilisation de la clé, de procédures d'identification avérées [Sieffer, 1988 ; Pavé *et al.*, 1991]. Mais, du point de vue de l'utilisateur, cela revient à utiliser l'informatique principalement, là encore, pour s'aider au parcours correcte de la clé. Or ce n'est souvent qu'un maigre apport, vu la façon dont les systématiciens ont l'habitude de « feuilleter » efficacement les clés sur papier.

Mais le principal écueil à l'utilisation des systèmes de représentation de connaissances par d'objets semble résider dans l'absence de signification des nœuds des clés. En effet, ces nœuds ne sont pas des concepts, en tout cas au sens où la biologie le désire. Par exemple, dans la clé précédente, la sous-classe des Tristichaceae ayant 2 bractéoles et dont l'anthère des étamines mesure de 0.4 cm à 0.7 cm, qui apparaît dans la clé, est un groupe purement artificiel, ou, en tout cas, dont on ne connaît pas de justification intrinsèque à sa formation. Ainsi, cette classe de plantes n'est non seulement pas un *concept primitif* (voir *chapitre 11*), car elle n'a pas d'existence propre, mais elle n'est pas non plus un *concept défini* (voir *chapitre 11*), car personne n'a besoin de le former, cette classe n'ayant aucune utilité connue hors de cette clé. Des remarques précédentes, il ne faudrait pas conclure qu'il n'y a pas de justification aux clés produites par les systématiciens, mais plutôt que cette justification ne s'exprime pas en termes d'une justification des classes induites dans les clés.

Bien comprendre la nature des clés impose de distinguer les trois types de connaissances véhiculées par elles. La première composante est évidemment diagnostique : elles sont faites pour aider à identifier. Il existe aussi une composante purement descriptive car, de la lecture des clés, on peut facilement déduire quels sont certains des aspects possibles des taxons concernés. Enfin, et surtout, les clés véhiculent une certaine connaissance stratégique car elles expriment comment on peut sélectionner et ordonner l'usage des descripteurs et leurs formulations pour réaliser des identifications aisées, fiables, rapides, justes, etc. Par exemple, pourquoi commencer la clé de notre exemple par le nombre de bractéoles ? Qu'est-ce qui, dans la nature, pourrait nous imposer cela si ce n'est le fait que commencer par là nous facilite, quel que soit le sens précis derrière ce mot, l'identification ? En effet, il est possible de changer l'ordre de l'emploi des descripteurs et à chaque fois obtenir une clé fonctionnelle différente. Si parmi toutes ces clés un systématicien en choisit une, c'est parce qu'il dispose de connaissances stratégiques qu'il utilise pour optimiser empiriquement certaines propriétés de la clé qu'il juge utiles. Ainsi, ce qui justifie une clé, ce sont les critères objectifs que l'on veut atteindre en construisant celle-ci plutôt qu'une autre.

Si on juge qu'avoir des concepts signifiants est indispensable en représentation des connaissances, il est encore possible d'exiger de construire des clés ayant cette propriété [Aïmeur et Kieu, 1993]. Mais dans ce cas, on retombe sur la difficulté déjà indiquée plus haut (voir § 15.3.2) : il est très difficile d'avoir à la fois de bons concepts et de bonnes caractérisations. Ce problème est connu depuis fort longtemps en systématique et s'est souvent exprimé sous la forme d'une discussion des différences entre les clés dites « artificielles » car ne s'intéressant qu'à l'identification et les clés dites « naturelles » car reflet d'une organisation de la Nature.

Il est fréquent de rencontrer dans les publications de systématique des clés qui ne « marchent pas », c'est-à-dire qui ne donnent pas toujours le bon résultat. Le plus souvent ce sont des clés « naturelles » dans lesquelles leurs auteurs se sont attachés à exprimer les grandes lignes d'une classification biologique [13]. Ces clés fonctionnent rarement pour deux raisons principales, d'une part car, devant la difficulté de les réaliser, le biologiste a volontairement oublié de considérer des formes connues, mais rares, d'êtres vivants. Par exemple, il est courant dans une clé des grands groupes d'animaux de caractériser les insectes comme ayant une paire d'antennes, oubliant un petit groupe, les protoures, qui n'en a pas. D'autre part, les descripteurs utilisés dans une clé naturelle s'appliquent souvent aux êtres vivants en tant que tout [14] et non pas à des spécimens à un temps et à un lieu donné [15] qui sont les seuls à être couramment soumis à l'identification. En effet, il est très incommode pour un utilisateur, par exemple durant une identification d'une grenouille, de lui demander de décrire ce qui concerne l'aspect larvaire et l'aspect adulte d'un même spécimen car, celui-ci, est a un moment donné soit l'un soit l'autre. À l'évidence, qu'il dispose de l'un ou de l'autre, un utilisateur d'une clé naturelle bloquera et se trompera fréquemment.

Comment réconcilier les deux points de vues ? L'approche qui semble la plus intéressante pour une identification assistée par ordinateur serait de découpler complètement la composante stratégique de la composante descriptive [Causse et Lebbe, 1995] en faisant bénéficier chacune d'une représentation par d'objets individualisée. Cela aurait comme avantage, non seulement de préserver explicitement une part de l'expertise en identification non considérée actuellement, mais surtout de permettre de représenter en toute liberté les concepts biologiques pertinents et leurs relations.

## 15.4 Conclusion

Les divers aspects de la classification en systématique sont complexes et ce chapitre avait pour but de participer à la clarification de cette partie de la biologie et à

13. C'est-à-dire précisément que l'auteur s'est attaché à ce que corresponde à chaque nœud de la clé un et un seul taxon d'une classification linnéenne.

14. On parle dans ce cas d'holomorphe pour désigner un être vivant dans toute son histoire.

15. On parle dans ce cas de sémaphoronte (porteur de sens).

la démonstration que les similarités avec les systèmes de représentation par objets ne sont pas superficielles.

En revanche, une différence importante à relever est l'accent tout particulier qui est mis en biologie sur la justification des classifications et donc sur les algorithmes de construction de structures classificatoires. Cela est dû au fait que, contrairement aux applications industrielles, les classifications résultent en systématique d'une activité scientifique cherchant à évacuer l'arbitraire autant que faire se peut.

Si la systématique a su se munir de méthodes et d'outils informatiques de calcul performants, les problèmes de représentation n'ont été principalement abordés jusqu'à présent qu'avec des méthodes *ad hoc* dont on ne sait pas si elles résisteront aux changements d'échelle imposés par les grands projets internationaux d'informatisation.

Le rapprochement de la biologie et de l'informatique est aujourd'hui en cours en biologie moléculaire dans de nombreux pays. Dans d'autres disciplines de la biologie comme la systématique il apparaît plus timidement, vraisemblablement en raison de la complexité plus grande des connaissances traitées et des problèmes posés, mais aussi du fait des bouleversements inéluctables qu'un tel rapprochement génère dans la pratique de la biologie. Pourtant dans le cas de la systématique, le bénéfice à attendre d'une organisation efficace des relations avec l'informatique n'est pas seulement pragmatique mais aussi et surtout de nature conceptuelle. En effet, la pratique de cette forme de biologie doit pouvoir bénéficier non seulement des outils de l'informatique mais surtout de ses approches formelles du traitement de l'information [Lebbe, 1995]. Ce qui pourrait paraître plus surprenant, c'est qu'inversement l'intelligence artificielle peut vraisemblablement aussi bénéficier de la systématique et des réflexions des systématiciens sur leurs propres pratiques, notamment des méthodes de représentation, de formation de concepts, de classification et d'identification éprouvées par trois siècles de pratique.

# Bibliographie

[Abadi et Cardelli, 1996] M. Abadi et L. Cardelli. *A Theory Of Objects*. Monographs in Computer Science. Springer-Verlag, Berlin, 1996.

[Abiteboul *et al.*, 1987] éditeurs S. Abiteboul, P.C. Fisher, et H.-J. Schek. *Proceedings of the Workshop on Theory and Applications of Nested Relations and Complex Objects, Darmstadt, Germany*, Lecture Notes in Computer Science 361, 1987.

[Abiteboul *et al.*, 1995] S. Abiteboul, R. Hull, et V. Vianu. *Foundations of Databases*. Addison-Wesley, Reading (MA), USA, 1995.

[Abiteboul et Bidoit, 1986] S. Abiteboul et N. Bidoit. Non first normal form relations: An algebra allowing data restructuring. *Journal of Computer and System Sciences (special issue, Selected Papers of PODS'85)*, 33(3):361–393, 1986.

[Abiteboul et Bonner, 1991] S. Abiteboul et A. Bonner. Objects and views. In *Proceedings of the ACM/SIGMOD International Conference on the Management of Data, Denver (CO), USA*, ACM SIGMOD Record, 20(2), pages 238–247, 1991.

[Abiteboul et Hull, 1987] S. Abiteboul et R. Hull. IFO, a formal semantic database model. *ACM Transactions on Database Systems*, 12(4):119–132, 1987.

[Accetta *et al.*, 1986] M. Accetta, R. Baron, W. Bolosky, D. Golub, E. Rashid, A. Tevanian, et M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the USENIX Conference, Atlanta (GA), USA*, pages 93–112, 1986.

[Ada, 1983] Reference manual for the Ada programming language. Rapport Technique ANSI/MIL-STD-1815A, ALSYS, La Celle Saint Cloud, France, 1983.

[AFIA, 1995] Dossier I.A. et musique, réalisé par F. Pachet et G. Assayag. Bulletin de l'Association Française pour l'Intelligence Artificielle, 23:30–48, 1995.

[Agesen *et al.*, 1993] O. Agesen, L. Bak, C. Chambers, B.W. Chang, U. Hølzle, J. Maloney, R.B. Smith, D. Ungar, et M. Wolczko. *The SELF 3.0 Programmer's Reference Manual*. Sun Microsystems, Inc., and Stanford University, 1993.

[Agesen *et al.*, 1995] O. Agesen, L. Bak, C. Chambers, B.W. Chang, U. Hølzle, J. Maloney, R.B. Smith, D. Ungar, et M. Wolczko. *The SELF 4.0 Programmer's Reference Manual*. Sun Microsystems, Inc., and Stanford University, 1995.

[Agha *et al.*, 1993] G. Agha, S. Frølund, R. Panwar, et D. Sturman. A linguistic framework for dynamic composition of dependability protocols. In *Dependable Computing for Critical Applications III (DCCA-3)*, IFIP Transactions, pages 197–207. Elsevier, New York (NY), USA, 1993.

[Agha et Hewitt, 1987] G. Agha et C. Hewitt. Concurrent programming using actors. In *Object-Oriented Concurrent Programming*, éditeurs A. Yonezawa et M. Tokoro, pages 37–53. MIT Press, Cambridge (MA), USA, 1987.

[Agha, 1986] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. Series in Artificial Intelligence. MIT Press, Cambridge (MA), USA, 1986.

[Aho *et al.*, 1989] A. Aho, R. Sethi, et J. Ullman. *Compilateurs : principes, techniques et outils*. InterEditions, Paris, 1989.

[Aïmeur et Kieu, 1993] E. Aïmeur et Q. Kieu. A method of incremental acquisition of structured objects by discrimination: Application to organisms' biology. In *Proceedings of the Knowledge and Data Engineering Workshop, Strasbourg, France*, pages 169–182, 1993.

[Aït-Kaci et Podelski, 1991] H. Aït-Kaci et A. Podelski. Towards a meaning of LIFE. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming, Passau, Germany*, éditeurs W. Bibel et Ph. Jorrand, Lecture Notes in Computer Science 528, pages 255–274. Springer-Verlag, Berlin, 1991.

[Alander, 1985] J. Alander. On interval arithmetic range approximation methods of polynomials and rational functions. *Computer and Graphics*, 9(4):365–372, 1985.

[Albert, 1984] P. Albert. KOOL at a glance. In *Proceedings of ECAI'84, Pisa, Italy*, page 345, 1984.

[Allen, 1984] J.F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.

[America, 1986] P. America. POOL-T: A parallel object-oriented language. In *Object-Oriented Concurrent Programming*, éditeurs A. Yonezawa et M. Tokoro, pages 199–220. MIT Press, Cambridge (MA), USA, 1986.

[America, 1987] P. America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of ECOOP'87, Paris, France*, éditeurs J. Bézivin, J.-M. Hullot, P. Cointe, et H. Lieberman, Lecture Notes in Computer Science 276, pages 234–242. Springer-Verlag, Berlin, 1987.

[André, 1996] P. André. Vers un modèle formel en analyse à objets. In *Actes du Colloque Langages et Modèles à Objets (LMO'96), Leysin, Suisse*, éditeur Y. Dennebouy, pages 62–78. École Polytechnique Fédérale de Lausanne, 1996.

[Andrew et Harris, 1987] T. Andrew et C. Harris. Combining language and database advances in an object-oriented development environment. In *Proceedings of OOPSLA'87, Orlando (FL), USA*, special issue of ACM SIGPLAN Notices, 22(12), pages 430–440, 1987.

[ANSI X3.135-1986, 1986] ANSI X3.135-1986. *The Database Language SQL*. American National Standards Institut, New York (NY), USA, 1986.

[Apple, 1989] Apple Computer, Inc. *Macintosh Allegro Common Lisp Reference Manual, Version 1.3*, 1989.

[Archie, 1984] J.W. Archie. A new look at the predictive value of numerical classifications. *Systematic Zoology*, 33:30–51, 1984.

[Atkinson *et al.*, 1983] M.P. Atkinson, K.J. Chisholm, P. Cockshott, et R.M. Marshall. Algorithms for a persistent heap. *Software – Practice and Experience*, 13(3):259–271, 1983.

[Atkinson *et al.*, 1989] M.P. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, et S. Zdonick. The object-oriented database system manifesto. In *The $O_2$ Book*, éditeurs F. Bancilhon, C. Delobel, et P. Kannelakis, pages 25–42. GIP Altaïr, INRIA Rocquencourt, 1989. First published in the Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, 1989, pages 40–57.

[Atkinson *et al.*, 1996] M.P. Atkinson, M.J. Jordan, L. Daynè, et S. Spence. Design issues for persistent java: A type-safe, object-oriented, orthogonally persistent system. In *Proceedings of the 7th Workshop on Persistent Object Systems (POS'96), Cape May (NJ), USA*, pages 33–47, 1996.

[Atkinson et Lausen, 1987] R. Atkinson et J. Lausen. Opus: A Smalltalk production system. In *Proceedings of OOPSLA'87, Orlando (FL), USA*, special issue of ACM SIGPLAN Notices, 22(12), pages 377–387, 1987.

[Attali *et al.*, 1993] I. Attali, D. Caromel, et M. Oudshoorn. A formal definition of the dynamic semantics of the Eiffel language. In *Proceedings of the 16th Australian Computer Science Conference, Brisbane, Australia*, éditeurs G. Gupta, G. Mohay, et R. Topor, pages 109–120. Griffith University, 1993.

[Attali *et al.*, 1995] I. Attali, D. Caromel, et S.O. Ehmety. An operational semantics for the Eiffel// language. In *Actes des Journées du GDR Programmation, Grenoble*, 1995. Également Rapport de Recherche 2732, INRIA Sophia Antipolis.

[Attali *et al.*, 1996] I. Attali, D. Caromel, S.O. Ehmety, et S. Lippi. Semantic-based visualization for parallel object-oriented programming. In *Proceedings of OOPSLA'96, San Jose (CA), USA*, special issue of ACM SIGPLAN Notices, 31(10), pages 421–440, 1996.

[Attardi, 1991] G. Attardi. An analysis of taxonomic reasoning. In *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, éditeurs M. Lenzerini, D. Nardi, et M. Simi, pages 29–49. John Wiley & Sons, Chichester (West Sussex), UK, 1991.

[Avesani *et al.*, 1990] P. Avesani, A. Perini, et F. Ricci. COOL: An object system with constraints. In *Proceedings of TOOLS'90, Washington (DC), USA*, pages 221–228, 1990.

[Ayache *et al.*, 1995] S. Ayache, M. Haziza, et D. Cayrac. A hybrid method approach for object-oriented knowledge-based systems specification and design. In *Proceedings of TOOLS-Europe'95, Versailles, France*, éditeurs I. Graham, B. Magnusson, B. Meyer, et J.-M. Nerson, pages 99–108. Prentice-Hall, Hertfordshire, UK, 1995.

[Baader *et al.*, 1990] F. Baader, H.-J. Bürckert, J. Heinsohn, B. Hollunder, J. Müller, B. Nebel, W. Nutt, et H.-J. Profitlich. Terminological knowledge representation: A proposal for a terminological logic. Technical Memo TM-90-04, DFKI, Saarbrücken Universität, Germany, 1990.

[Baader *et al.*, 1994] F. Baader, B. Hollunder, B. Nebel, H.-J. Profitlich, et E. Franconi. An empirical analysis of optimization techniques for terminological representation systems. *Journal of Applied Intelligence*, 4(2):109–132, 1994.

[Baader et Hanschke, 1991] F. Baader et P. Hanschke. A scheme for integrating concrete domains into concept languages. In *Proceedings of IJCAI'91, Sydney, Australia*, pages 452–457, 1991.

[Baader et Hollunder, 1991a] F. Baader et B. Hollunder. KRIS: Knowledge Representation and Inference System. *ACM SIGART Bulletin*, 2(3):8–14, 1991.

[Baader et Hollunder, 1991b] F. Baader et B. Hollunder. A terminological knowledge representation system with complete inference algorithms. In *Proceedings of the Conference on Processing Declarative Knowledge, Kaiserslautern, Germany*, éditeurs H. Boley et M.M. Richter, Lecture Notes in Computer Science 567, pages 67–86. Springer-Verlag, Berlin, 1991.

[Baader et Hollunder, 1993] F. Baader et B. Hollunder. How to prefer more specific defaults in terminological default logic. In *Proceedings of IJCAI'93, Chambéry, France*, pages 669–674, 1993.

[Baader, 1996] F. Baader. A formal definition for the expressive power of terminological knowledge representation languages. *Journal of Logic and Computation*, 6(1):33–54, 1996.

[Ballesta, 1994] P. Ballesta. *Contraintes et objets : clefs de voûte d'un outil d'aide à la composition*. Thèse d'Informatique, Université du Maine, Le Mans, 1994.

[Balter *et al.*, 1994] R. Balter, S. Lacourte, et M. Riveill. The Guide language. *The Computer Journal (special issue, Distributed Operating Systems)*, 37(6):519–530, 1994.

[Banatre, 1990] J.-P. Banatre. *La programmation parallèle, outils, méthodes et éléments de mise en œuvre*. Eyrolles, Paris, 1990.

[Bancilhon *et al.*, 1988] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, et F. Velez. The design and implementation of $O_2$, an object-oriented database system. In *Advances in Object-Oriented Database Systems, Proceedings of the 2nd International Workshop on Object-Oriented Database Systems, Bad-Münster am Stein-Ebernburg, Germany*, éditeur K.R. Dittrich, Lecture Notes in Computer Science 334, pages 1–32, 1988.

[Bancilhon *et al.*, 1991] F. Bancilhon, C. Delobel, et F. Kanellakis. *Building an Object-Oriented Database System: The Story of $O_2$*. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1991.

[Banerjee *et al.*, 1987] J. Banerjee, W. Kim, K.J. Kim, et H. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM/SIGMOD International Conference on the Management of Data, San Francisco (CA), USA*, ACM SIGMOD Record, 16(3), pages 311–322, 1987.

[Bardou *et al.*, 1996] D. Bardou, C. Dony, et J. Malenfant. Comprendre et interpréter la délégation. Une application aux objets morcelés. In *Actes des Journées du GDR Programmation, Orléans*, 1996. Session Programmation Objet.

[Bardou et Dony, 1995] D. Bardou et C. Dony. Propositions pour un nouveau modèle d'objets dans les langages à prototypes. In *Actes du Colloque Langages et Modèles à Objets (LMO'95), Nancy, France*, éditeur A. Napoli, pages 93–109. INRIA Lorraine, Nancy, 1995.

[Bardou et Dony, 1996] D. Bardou et C. Dony. Split objects: A disciplined use of delegation within objects. In *Proceedings of OOPSLA'96, San Jose (CA), USA*, special issue of ACM SIGPLAN Notices, 31(10), pages 122–137, 1996.

[Bardou, 1998] D. Bardou. *Étude des langages à prototypes, du mécanisme de délégation, et de son rapport à la notion de point de vue*. Thèse de doctorat en informatique, Université Montpellier 2, 1998.

[Barga et Pu, 1995] R. Barga et C. Pu. A practical and modular implementation of extended transaction models. Technical Report 95-004, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, Portland (OR), USA, 1995.

[Barnes, 1995] J.G.P. Barnes. *Programmer en Ada, 4e édition*. Addison-Wesley, Paris, 1995.

[Baude *et al.*, 1996a] F. Baude, F. Belloncle, J.-C. Bermond, D. Caromel, O. Dalle, E. Darrot, O. Delmas, N. Furmento, B. Gaujal, P. Mussi, S. Perennes, Y. Roudier, et G. Siegel. The SLOOP Project: Simulations, parallel object-oriented languages, interconnection networks. In *Proceedings of the 2nd European School of Computer Science, Parallel Programming Environments for High Performance Computing, Alpe d'Huez, France*, pages 85–88, 1996.

[Baude *et al.*, 1996b] F. Baude, F. Belloncle, D. Caromel, N. Furmento, P. Mussi, Y. Roudier, et G. Siegel. Parallel object-oriented programming for parallel simulations. *Information Sciences Journal, Informatics and Computer Science (special issue, Object-Oriented Programming)*, 93(1–2):35–64, 1996.

[Beeri *et al.*, 1989] C. Beeri, P.A. Bernstein, et N. Goodman. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230–269, 1989.

[Beeri, 1989] C. Beeri. Formal models for object-oriented databases. In *Proceedings of the 1st Conference on Deductive and Object-Oriented Databases (DOOD'89), Kyoto, Japan*, pages 405–430, 1989.

[Bellahsene *et al.*, 1996] Z. Bellahsene, P. Poncelet, et M. Teisseire. Views for information design without reorganization. In *Proceedings of the 8th International Conference on Advanced Information Systems Engineering (CAiSE'96), Crete, Greece*, pages 496–513, 1996.

[Benatallah, 1996] B. Benatallah. Un compromis : modification et versionnement du schéma. In *Actes du 11e Colloque Bases de Données Avancées (BDA'96), Cassis, France*, pages 373–396, 1996.

[Benhamou *et al.*, 1994] F. Benhamou, D. McAllester, et P. Van Hentenryck. CLP(Intervals) revisited. Technical Report CS-94-18, Brown University, Providence (RI), USA, 1994.

[Benzaken et Doucet, 1993] V. Benzaken et A. Doucet. *Bases de données orientées objet : concepts et principes*. Armand Colin, Paris, 1993.

[Benzaken et Schaefer, 1997] V. Benzaken et X. Schaefer. Static integrity constraint management in object-oriented database programming languages via predicate transformers. In *Proceedings of ECOOP'97, Jyväskylä, Finland*, éditeurs M. Aksit et S. Matsuoka, Lecture Notes in Computer Science 1241. Springer--Verlag, Berlin, 1997.

[Berge, 1970] C. Berge. *Graphes et hypergraphes*. Dunod, Paris, 1970.

[Bergmans, 1994] L.M.J. Bergmans. *Composing Concurrent Objects*. PhD thesis, Universiteit Twente, The Netherlands, 1994.

[Bergstein et Lieberherr, 1991] P.L. Bergstein et K.J. Lieberherr. Incremental class dictionary learning and optimization. In *Proceedings of ECOOP'91, Geneva, Switzerland*, pages 337–396, 1991.

[Berlandier, 1992] P. Berlandier. *Étude de mécanismes d'interprétation de contraintes et de leur intégration dans un système à base de connaissances*. Thèse d'Informatique, Université de Nice Sophia Antipolis, 1992.

[Berners-Lee *et al.*, 1992] T. Berners-Lee, R. Cailliau, J.-F. Groff, et B. Pollerman. World Wide Web: The information universe. *Electronic Networking: Research, Applications and Policy*, 2(1):45–51, 1992.

[Bernstein *et al.*, 1987] P. Bernstein, V. Hadzilacos, et N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading (MA), USA, 1987.

[Bertino, 1992] E. Bertino. A view mechanism for object-oriented databases. In *Proceedings of the International Conference on Extensive Data Base Technology (EDBT'92), Vienna, Austria*, Lecture Notes in Computer Science 580, pages 136–151. Springer-Verlag, Berlin, 1992.

[Bessière *et al.*, 1995] C. Bessière, E.C. Freuder, et J.-C. Régin. Using inference to reduce arc consistency computation. In *Proceedings of IJCAI'95, Montréal, Canada*, pages 592–598, 1995.

[Bézivin, 1987] Jean Bézivin. Some experiments in object-oriented simulation. In *Proceedings of OOPSLA'87, Orlando (FL), USA*, special issue of ACM SIGPLAN Notices, 22(12), pages 394–405, 1987.

[Bionnerstedt et Brigitts, 1988] A. Bionnerstedt et S. Brigitts. An object management system. In *Proceedings of OOPSLA'88, San Diego (CA), USA*, pages 206–221, 1988.

[Birtwistle *et al.*, 1973] G. Birtwistle, O.J. Dahl, B. Myhrhaug, et K. Nygaard. *SIMULA Begin*. Petrocelli Charter, New York (NY), USA, 1973.

[Black et Immel, 1993] A.P. Black et M.P. Immel. Encapsulating plurality. In *Proceedings of ECOOP'93, Kaiserslautern, Germany*, éditeur O. Nierstrasz, Lecture Notes in Computer Science 707, pages 57–79. Springer-Verlag, Berlin, 1993.

[Black, 1991] A.P. Black. Understanding transactions in the operating system context. *Operating Systems Review*, 25(28):73–77, 1991.

[Blaschek, 1994] G. Blaschek. *Object-Oriented Programming With Prototypes*. Springer-Verlag, Berlin, 1994.

[Bobrow et Stefik, 1983] D.G. Bobrow et M. Stefik. The LOOPS manual: A data and object oriented programming system for Interlisp. Knowledge-Based VLSI Design Group Memo, Xerox PARC, Palo Alto (CA), USA, 1983.

[Bobrow et Winograd, 1977] D.G. Bobrow et T. Winograd. An overview of KRL, a Knowledge Representation Language. *Cognitive Science*, 1(1):3–46, 1977.

[Boehm, 1976] B.W. Boehm. Software engineering. *IEEE Transactions on Computers*, 25(12):1226–1241, 1976.

[Booch *et al.*, 1996] G. Booch, J. Rumbaugh, et I. Jacobson. Unified method for object-oriented development. Documentation set, version 0.91 addendum. Technical Report, Rational Software Corporation, Santa Clara (CA), USA, 1996.

[Booch et Rumbaugh, 1995] G. Booch et J. Rumbaugh. Unified method for object-oriented development. Documentation set, version 0.8. Technical Report, Rational Software Corporation, Santa Clara (CA), USA, 1995.

[Booch et Vilot, 1996] G. Booch et M. Vilot. Simplifying the Booch components. http://www.objectplace.com/technology/articles/9306.shtml, 1996. Première publication dans *The C++ Report*, juin 1993.

[Booch, 1986] G. Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12:211–221, 1986.

[Booch, 1987] G. Booch. *Software Engineering with Ada, Second Edition*. Benjamin/Cummings Publishing, Reading (MA), USA, 1987.

[Booch, 1990] G. Booch. *Object Oriented Design with Applications*. Benjamin/-Cummings Publishing, Reading (MA), USA, 1990.

[Booch, 1994a] G. Booch. *Analyse et conception orientées objet, 2e édition*. Addison-Wesley, Paris, 1994.

[Booch, 1994b] G. Booch. *Object Oriented Analysis and Design with Applications, Second Edition*. Benjamin/Cummings Publishing, Reading (MA), USA, 1994.

[Borgida *et al.*, 1989] A. Borgida, R.J. Brachman, D.L. McGuinness, et L.A. Resnick. CLASSIC: A structural data model for objects. In *Proceedings of the ACM/SIGMOD International Conference on the Management of Data, Portland (OR), USA*, ACM SIGMOD Record, 18(2), pages 59–67, 1989.

[Borgida et Patel-Schneider, 1994] A. Borgida et P.F. Patel-Schneider. A semantics and complete algorithm for subsumption in the CLASSIC description logic. *Journal of Artificial Intelligence Research*, 1:277–308, 1994.

[Borgida, 1986] A. Borgida. Exceptions in object-oriented languages. *ACM SIGPLAN Notices*, 21(10):107–119, 1986.

[Borgida, 1992] A. Borgida. From type systems to knowledge representation: Natural semantics specifications for description logics. *International Journal of Intelligent and Cooperative Information Systems*, 1(1):93–126, 1992.

[Borgida, 1995] A. Borgida. Description logics in data management. *IEEE Transactions on Knowledge and Data Engineering*, 7(5):671–682, 1995.

[Borgida, 1996] A. Borgida. On the relative expressiveness of description logics and predicate logics. *Artificial Intelligence*, 82(1–2):353–367, 1996.

[Borning *et al.*, 1987] A. Borning, R. Duisberg, B. Freeman-Benson, A. Kramer, et M. Woolf. Constraint hierarchies. In *Proceedings of OOPSLA'87, Orlando (FL), USA*, special issue of ACM SIGPLAN Notices, 22(12), pages 48–60, 1987.

[Borning et Duisberg, 1986] A. Borning et R. Duisberg. Constraint-based tools for building user interfaces. *ACM Transactions on Graphics*, 5(4):345–374, 1986.

[Borning, 1981] A.H. Borning. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, 1981.

[Borning, 1986] A.H. Borning. Classes versus prototypes in object-oriented languages. In *Proceedings of the ACM-IEEE Fall Joint Computer Conference, Montvale (NJ), USA*, pages 36–39, 1986.

[Boussard *et al.*, 1990] J.-C. Boussard, C. Michel, J.-P. Partouche, R. Rousseau, et M. Rueher. Une évaluation du langage Eiffel. *Technique et Science Informatiques*, 9(4):331–373, 1990.

[Boussard et Rousseau, 1995] J.-C. Boussard et R. Rousseau. Why and how to teach Eiffel in software university education. Our experience at UNSA for seven years. In *Urkunde der Konferenz auf "Eiffel in Schule und Hochschule; Ausbildung in objektorientiertem Programmieren", Darmstadt, Deutschland*, éditeur H.-J. Hoffmann. Electronic proceedings, Technische Hochschule Darmstadt (http://www.pu.informatik.tu-darmstadt.de/eis/), 1995. 25 slides.

[Boyland et Castagna, 1996] J. Boyland et G. Castagna. Type-safe compilation of covariant specialization: A practical case. In *Proceedings of ECOOP'96, Linz, Austria*, éditeur P. Cointe, Lecture Notes in Computer Science 1098, pages 3–25. Springer-Verlag, Berlin, 1996.

[Brachman *et al.*, 1983] R.J. Brachman, R.E. Fikes, et H.J. Levesque. KRYPTON: A functional approach to knowledge representation. *Computer*, 16(10):67–73, 1983.

[Brachman *et al.*, 1991] R.J. Brachman, D.L. McGuinness, P.F. Patel-Schneider, L.A. Resnick, et A. Borgida. Living with CLASSIC: When and how to use a KL-ONE language. In *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, éditeur J.F. Sowa, pages 401–456. Morgan Kaufmann Publishers, San Mateo (CA), USA, 1991.

[Brachman et Levesque, 1984] R.J. Brachman et H.J. Levesque. The tractability of subsumption in frame-based description languages. In *Proceedings of AAAI'84, Austin (TX), USA*, pages 34–37, 1984.

[Brachman et Schmolze, 1985] R.J. Brachman et J.G. Schmolze. An overview of the KL-ONE knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[Brachman, 1977] R.J. Brachman. What's in a concept: Structural foundations for semantic networks. *International Journal of Man-Machine Studies*, 9:127–152, 1977.

[Brachman, 1979] R.J. Brachman. On the epistemological status of semantic networks. In *Associative Networks: Representation and Use of Knowledge by Computers*, éditeur N.V. Findler, pages 3–50. Academic Press, New York (NY), USA, 1979.

[Brachman, 1983] R.J. Brachman. What IS-A is and isn't: An analysis of taxonomic links in semantic networks. *Computer*, 16(10):30–37, 1983.

[Brachman, 1985] R.J. Brachman. "I lied about the trees" or defaults and definitions in knowledge representation. *The AI Magazine*, 6(3):80–93, 1985.

[Brant, 1995] J.M. Brant. HotDraw. Master's thesis, University of Illinois at Urbana-Champaign, 1995.

[Breiman *et al.*, 1984] L. Breiman, J.H. Friedman, R.A. Olshen, et C.J. Stone. *Classification and Regression Trees*. Wadsworth, Belmont (CA), USA, 1984.

[Bretl *et al.*, 1989] R. Bretl, D. Maier, A. Otis, D.J. Penney, B. Schuchardt, J. Stein, E.H. Williams, et M. Williams. The GemStone data management system. In *Object-Oriented Concepts, Databases and Applications*, éditeurs W. Kim et F.H. Lochovsky, pages 283–308. Addison-Wesley, Reading (MA), USA, 1989.

[Briot *et al.*, 1996] éditeurs J.-P. Briot, J.-M. Geib, et A. Yonezawa. *Object-Based Parallel and Distributed Computation*. Lecture Notes in Computer Science 1107. Springer-Verlag, Berlin, 1996.

[Briot et Cointe, 1989] J.-P. Briot et P. Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proceedings of OOPSLA'89, New Orleans (LA), USA*, special issue of ACM SIGPLAN Notices, 24(10), pages 419–431, 1989.

[Briot et Guerraoui, 1996] J.-P. Briot et R. Guerraoui. Smalltalk for concurrent and distributed programming. In *Informatik/Informatique, 1, special issue on Smalltalk*, éditeur R. Guerraoui, pages 16–19. Swiss Informaticians Society, Switzerland, 1996.

[Briot et Yonezawa, 1987] J.-P. Briot et A. Yonezawa. Inheritance and synchronization in concurrent OOP. In *Proceedings of ECOOP'87, Paris, France*, éditeurs J. Bézivin, J.-M. Hullot, P. Cointe, et H. Lieberman, Lecture Notes in Computer Science 276, pages 35–43. Springer-Verlag, Berlin, 1987.

[Briot, 1984] J.-P. Briot. *Instanciation et héritage dans les langages objets*. Thèse de 3e Cycle, Université Pierre et Marie Curie (Paris VI), 1984. Rapport LITP 85-21.

[Briot, 1994] J.-P. Briot. Modélisation et classification de langages de programmation concurrente à objets : l'expérience Actalk. In *Actes du Colloque Langages et Modèles à Objets (LMO'94), Grenoble, France*, éditeur F. Rechenmann, pages 153–165. INRIA Rhônes-Alpes – IMAG-LIFIA, Grenoble, 1994.

[Briot, 1996] J.-P. Briot. An experiment in classification and specialization of synchronization schemes. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software, Kanazawa, Japan*, éditeurs K. Futatsugi et S. Matsuoka, Lecture Notes in Computer Science 1049, pages 227–249. Springer-Verlag, Berlin, 1996.

[Brissi et Gautero, 1991] P. Brissi et M. Gautero. Un prototype d'interprète pour le langage Eiffel. Mémoire de DEA, Université de Nice Sophia Antipolis, Laboratoire I3S, 1991.

[Brissi et Rousseau, 1995] P. Brissi et R. Rousseau. IREC: An object-oriented abstract representation to handle software components in a persistent framework. In

*Object-Oriented Technology for Database and Software Systems*, éditeurs V.S. Alagar et R. Missaoui, pages 6–21. World Scientific, Singapore, 1995.

[Brun, 1988] J. Brun. *Aristote et le Lycée*. Que sais-je ?, N° 928. Presses Universitaires de France, Paris, 1988.

[Buchheit *et al.*, 1993] M. Buchheit, F.-M. Donini, et A. Schaerf. Decidable reasoning in terminological knowledge representation systems. *Journal of Artificial Intelligence Research*, 1:109–138, 1993.

[Buchheit *et al.*, 1994] M. Buchheit, F.-M. Donini, W. Nutt, et A. Schaerf. Refining the structure of terminological systems: Terminology = schema + views. In *Proceedings of AAAI'94, Seattle (WA), USA*, pages 199–204, 1994.

[Burns, 1985] A. Burns. *Concurrent Programming in Ada*. Ada Companion Series. Cambridge University Press, Cambridge, UK, 1985.

[Buttner et Simonis, 1987] W. Buttner et H. Simonis. Embedding boolean expression into logic programming. *Journal of Symbolic Computation*, 4(2):191–205, 1987.

[Calvanese *et al.*, 1994] D. Calvanese, M. Lenzerini, et D. Nardi. A unified framework for class-based representation formalisms. In *Proceedings of KR'94, Bonn, Germany*, pages 109–120, 1994.

[Campbell *et al.*, 1993] R.H. Campbell, N. Islam, D. Raila, et P. Madany. Designing and implementing choices: An object-oriented system in C++. *Communications of the ACM (special issue, Concurrent Object-Oriented Programming, B. Meyer, editor)*, 36(9):117–126, 1993.

[Campbell et Haberman, 1974] R.H. Campbell et A.N. Haberman. The specification of process synchronization by path expression. In *Actes du Colloque sur les Aspects Théoriques et Pratiques des Systèmes d'Exploitation, Paris*, Lecture Notes in Computer Science 16, pages 89–102. Springer-Verlag, Berlin, 1974.

[Candolle, 1813] A.P. de Candolle. *Théorie élémentaire de la botanique ou exposition du principe de classification naturelle et de l'art de décrire et d'étudier les végétaux*. Déterville, Paris, 1813.

[Capobianchi *et al.*, 1992] R. Capobianchi, R. Guerraoui, A. Lanusse, et P. Roux. Lessons from implementing active objects on a parallel machine. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems, Newport Beach (CA), USA*, pages 13–26, 1992.

[Capponi *et al.*, 1995] C. Capponi, J. Euzenat, et J. Gensel. Objects, types and constraints as classification schemes. In *International Conference on Knowledge Re-use, Storage and Efficiency (KRUSE'95), Santa Cruz (CA), USA*, éditeurs G. Ellis, R.A. Levinson, A. Fall, et V. Dahl, pages 69–73, 1995.

[Capponi et Gensel, 1997] C. Capponi et J. Gensel. Typage souple pour objets contraints. In *Actes du Colloque Langages et Modèles à Objets (LMO'97), Roscoff, France*, éditeurs R. Ducournau et S. Garlatti, pages 191–206. Hermès, Paris, 1997.

[Capponi, 1995] C. Capponi. *Identification et exploitation des types dans un modèle de connaissances à objets*. Thèse d'Informatique, Université Joseph Fourier, Grenoble, 1995.

[Cardelli et Wegner, 1985] L. Cardelli et P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[Cardelli, 1995] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[Carnap, 1973] R. Carnap. *Les fondements philosophiques de la physique*. Collection U. Armand Colin, Paris, 1973.

[Carnegie Group, 1988] Carnegie Group, Inc., Pittsburgh (PA), USA. *Knowledge Craft, CRL Technical Manual, Version 3.2*, 1988.

[Caromel *et al.*, 1993] D. Caromel, Y. Roudier, et O. Roux. Programmation réactive en Eiffel// (expression de synchronisations selon le modèle Electre). Rapport de Recherche RR-93-70, Laboratoire I3S, Sophia Antipolis, 1993. http://alto.unice.fr/~roudier/RR93-70.ps (100 pages).

[Caromel *et al.*, 1996] D. Caromel, F. Belloncle, et Y. Roudier. The C++// language. In *Parallel Programming Using C++*, éditeurs G. Wilson et P. Lu, pages 257–296. MIT Press, Cambridge (MA), USA, 1996.

[Caromel et Roudier, 1996] D. Caromel et Y. Roudier. Reactive programming in Eiffel//. In *Proceedings of the Conference on Object-Based Parallel and Distributed Computation, Tokyo, Japan*, éditeurs J.-P. Briot, J.M. Geib, et A. Yonezawa, Lecture Notes in Computer Science 1107, pages 125–147. Springer-Verlag, Berlin, 1996.

[Caromel, 1991] D. Caromel. *Programmation parallèle asynchrone et impérative : études et propositions (une extension parallèle du langage objet Eiffel).* Thèse d'Informatique, Université Henri Poincaré Nancy 1, 1991.

[Caromel, 1993] D. Caromel. Towards a method of object-oriented programming. *Communications of the ACM (special issue, Concurrent Object-Oriented Programming, B. Meyer, editor)*, 36(9):90–102, 1993.

[Carré *et al.*, 1995] B. Carré, R. Ducournau, J. Euzenat, A. Napoli, et F. Rechenmann. Classification et objets : programmation ou représentation ? In *Actes des 5es Journées Nationales PRC-GDR Intelligence Artificielle, Nancy*, pages 213–237. Teknea, Toulouse, 1995.

[Carré, 1989] B. Carré. *Méthodologie orientée objet pour la représentation des connaissances. Concepts de points de vue, de représentation multiple et évolutive d'objet.* Thèse d'Informatique, Université des Sciences et Technologies de Lille, 1989.

[Cart et Ferrié, 1989] M. Cart et J. Ferrié. Integrating concurrency control into an object-oriented database system. In *Proceedings of the International Conference on Extensive Data Base Technology (EDBT'90), Venice, Italy*, Lecture Notes in Computer Science 416, pages 363–377. Springer-Verlag, Berlin, 1989.

[Caseau, 1985] Y. Caseau. Les objets et ensembles dans LORE. In *Actes du 5e Congrès AFCET Reconnaissance des Formes et Intelligence Artificielle, Grenoble*, pages 101–118 (tome I), 1985.

[Caseau, 1987] Y. Caseau. *Etude et réalisation d'un langage objet : LORE.* Thèse d'Informatique, Université de Paris-Sud, 1987.

[Caseau, 1991] Y. Caseau. An object-oriented deductive language. *Annals of Mathematics and Artificial Intelligence*, 3:211–258, 1991.

[Caseau, 1994] Y. Caseau. Constraint satisfaction with an object-oriented knowledge representation language. *Journal of Applied Intelligence*, 4(2):157–184, 1994.

[Castagna, 1995] G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–437, 1995.

[Castagna, 1996] G. Castagna. Le modèle fondé sur la surcharge : une visite guidée. *Technique et Science Informatiques*, 15(6):673–708, 1996.

[Castagna, 1997] G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theorical Computer Science. Birkhäuser, Berlin, 1997.

[Cattell *et al.*, 1994] R.G.G. Cattell, T. Atwood, J. Dubl, G. Ferran, M. Loomis, et D. Wade. *The Object Database Standard: ODMG*. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1994. Nouvelle édition : [Cattell *et al.*, 1997].

[Cattell *et al.*, 1997] R.G.G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, et D. Wade. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1997.

[Cattell, 1997] R.G.G. Cattell. *Bases de données orientées objets, 2e édition*. International Thomson Publishing France/Addison-Wesley, Paris, 1997.

[Causse et Lebbe, 1995] K. Causse et J. Lebbe. Modélisation des stratégies d'identification par la méthode MCC. In *Actes des Journées JAVA'95, Grenoble, France*, pages 345–358, 1995.

[Celeux *et al.*, 1989] G. Celeux, E. Diday, G. Govaert, Y. Lechevallier, et H. Ralambondrainy. *Classification automatique des données*. Dunod Informatique, Paris, 1989.

[Cellary et Jomier, 1990] W. Cellary et G. Jomier. Consistency of versions in object-oriented databases. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB'90), Brisbane, Australia*, pages 432–441. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1990.

[Chabre, 1988] B. Chabre. Le filtrage en YAFOOL. Rapport de stage ENST, Sema Group, Montrouge, 1988.

[Chailloux *et al.*, 1986] J. Chailloux, M. Devin, F. Dupont, J.-M. Hullot, B. Serpette, et J. Vuillemin. *Le_Lisp de l'Inria, version 15.2. Le manuel de référence*. INRIA, Rocquencourt, 1986.

[Chambers *et al.*, 1991] C. Chambers, D. Ungar, B.W. Chang, et U. Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *LISP and Symbolic Computation*, 4(3):207–222, 1991.

[Chambers, 1993] C. Chambers. Predicate classes. In *Proceedings of ECOOP'93, Kaiserslautern, Germany*, éditeur O. Nierstrasz, Lecture Notes in Computer Science 707, pages 268–296. Springer-Verlag, Berlin, 1993.

[Champeaux et Faure, 1992] D. de Champeaux et P. Faure. A comparative study of object-oriented analysis methods. *Journal of Object-Oriented Programming*, 5(1):21–33, 1992.

[Channon, 1996] D. Channon. Persistence for C++. *Dr. Dobb's Journal*, 21(10):46–77, 1996.

[Chein et Mugnier, 1992] M. Chein et M.-L. Mugnier. Conceptual graphs: Fundamental notions. *Revue d'intelligence artificielle*, 6(4):365–406, 1992.

[Chen, 1976] P.P.-S. Chen. The entity-relationship model—Towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[Chen, 1979] P.P.-S. Chen. *Entity-Relationship Model: Towards a Unified View of Data*. North-Holland, Amsterdam, 1979.

[Chiba, 1995] S. Chiba. A metaobject protocol for C++. In *Proceedings of OOPSLA'95, Austin (TX), USA*, special issue of ACM SIGPLAN Notices, 30(10), pages 285–299, 1995.

[Chignoli *et al.*, 1995] R. Chignoli, J. Farré, P. Lahire, et R. Rousseau. FLOO: A strong coupling between Eiffel language and $O_2$ DBMS. In *Object-Oriented Technology for Database and Software Systems*, éditeurs V.S. Alagar et R. Missaoui, pages 206–223. World Scientific, Singapore, 1995.

[Chignoli *et al.*, 1996] R. Chignoli, J. Farré, P. Lahire, et R. Rousseau. FLOO : un environnement pour la programmation persistante en Eiffel. *Technique et Science Informatiques*, 15(6):735–763, 1996. Numéro spécial *Systèmes à objets : tendances actuelles et évolution*, rédacteurs A. Napoli et J.-F. Perrot.

[Chouvet *et al.*, 1996] M.-P. Chouvet, F. Le Ber, J. Lieber, L. Mangelinck, A. Napoli, et A. Simon. Analyse des besoins en représentation et raisonnement dans une représentation à objets — L'exemple de Y3. In *Actes du Colloque Langages et Modèles à Objets (LMO'96), Leysin, Suisse*, éditeur Y. Dennebouy, pages 150–169. École Polytechnique Fédérale de Lausanne, 1996.

[Ciampi *et al.*, 1996] A. Ciampi, E. Diday, J. Lebbe, E. Perinel, et R. Vignes. Tree growing with probabilistically imprecise data. In *Proceedings of the International Conference on Ordinal and Symbolic Data Analysis, Paris, France*, éditeurs E. Diday, Y. Lechevallier, et O. Opitz, pages 201–214. Springer-Verlag, Berlin, 1996.

[Coad et Yourdon, 1991a] P. Coad et E. Yourdon. *Object-Oriented Analysis, Second Edition*. Yourdon Press/Prentice Hall, 1991.

[Coad et Yourdon, 1991b] P. Coad et E. Yourdon. *Object-Oriented Design*. Yourdon Press/Prentice Hall, 1991.

[Codani, 1988] J.-J. Codani. *Microprogrammation, architectures, langages à objets : NAS*. Thèse d'Informatique, Université Pierre et Marie Curie (Paris VI), 1988.

[Codd, 1970] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Codd, 1979] E.F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, 1979.

[Cohen et Murphy, 1984] B. Cohen et G.L. Murphy. Models of concepts. *Cognitive Science*, 8(1):27–58, 1984.

[Coleman *et al.*, 1996] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, et P. Jeremaes. *FUSION. La méthode orientée-objet de 2^e génération*. Masson, Paris, 1996.

[Colless, 1986] D.H. Colless. Classification et information. In *L'ordre et la diversité du vivant*, éditeur P. Tassy, pages 121–142. Fayard, Paris, 1986.

[Collet *et al.*, 1997] P. Collet, R. Rousseau, et J.-C. Royer. Efficacité et programmation par objets : rumeurs, problèmes et solutions. Rapport Technique 97–54, Laboratoire I3S, Sophia Antipolis, 1997.

[Collet et Adiba, 1993] C. Collet et M. Adiba. *Objets et bases de données : le système de gestion de bases de données $O_2$*. Hermès, Paris, 1993.

[Collet et Rousseau, 1996a] P. Collet et R. Rousseau. Assertions are objects too! In *Proceedings of the 1st White Object-Oriented Nights Conference (WOON'96), St-Petersburg, Russia*, éditeur F. Monninger, pages 1–13. Electronic proceedings (www.sigco.com/woon), 1996.

[Collet et Rousseau, 1996b] P. Collet et R. Rousseau. Classification et réification des assertions — Application au langage Eiffel. In *Actes du Colloque Langages et Modèles à Objets (LMO'96), Leysin, Suisse*, éditeur Y. Dennebouy, pages 10–27. École Polytechnique Fédérale de Lausanne, 1996.

[Collet, 1997] P. Collet. *Un modèle à objets fondé sur les assertions pour le génie logiciel et les bases de données : application au langage OQUAL, une extension d'Eiffel*. Thèse d'Informatique, Université de Nice Sophia Antipolis, Laboratoire I3S, 1997.

[Cook *et al.*, 1990] W.R. Cook, W.L. Hill, et P.S. Canning. Inheritance is not subtyping. In *Proceedings of the 17th International Conference on the Principles Of Programming Languages (POPL'90), San Francisco (CA), USA*, pages 125–135, 1990.

[Cook et Daniels, 1994] S. Cook et J. Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1994.

[Corey et Cheng, 1989] E.J. Corey et X.M. Cheng. *The Logic of Chemical Synthesis*. John Wiley & Sons, New York (NY), USA, 1989.

[Corey, 1967] E.J. Corey. General methods for the construction of complex molecules. *Pure & Applied Chemistry*, 14:19–37, 1967.

[Coupey et Fouqueré, 1994] P. Coupey et C. Fouqueré. Classer des concepts définis avec des défauts et des exceptions. In *Actes du Colloque Langages et Modèles à Objets (LMO'94), Grenoble, France*, éditeur F. Rechenmann, pages 69–80. INRIA Rhônes-Alpes – IMAG-LIFIA, Grenoble, 1994.

[Coupey et Fouqueré, 1997] P. Coupey et C. Fouqueré. Extending conceptual definitions with default knowledge. *Computational Intelligence*, 13(2):258–299, 1997.

[Cracraft, 1983] J. Cracraft. The significance of phylogenetic classifications for systematic and evolutionary biology. In *Numerical Taxonomy*, éditeur J. Felsenstein, pages 1–17. Springer-Verlag, Berlin, 1983.

[Crampé et Euzenat, 1996] I. Crampé et J. Euzenat. Fondements de la révision dans un langage d'objets simple. In *Actes du Colloque Langages et Modèles à Objets (LMO'96), Leysin, Suisse*, éditeur Y. Dennebouy, pages 134–149. École Polytechnique Fédérale de Lausanne, 1996.

[Dao et Richard, 1990] M. Dao et J.-P. Richard. CASSIS — Manuel de référence. Rapport Technique NT/PAA/ATR/RIP/2626, Centre National d'Etudes des Télécommunications, Lannion, 1990.

[Darlu et Tassy, 1993] P. Darlu et P. Tassy. *Reconstruction phylogénétique : concepts et méthodes*. Masson, Paris, 1993.

[Date et Hopewell, 1971] C.J. Date et P. Hopewell. File definition and logical data independence. In *Proceedings of the ACM SIGFIDET Workshop on Data Description, Access and Control, San Diego (CA), USA*, pages 117–138, 1971.

[Date, 1984] C.J. Date. A critique of the SQL database language. *ACM SIGMOD Record*, 14(3):8–54, 1984.

[Date, 1990] C.J. Date. *Introduction to Database Systems*, volume 1. Addison--Wesley, Reading (MA), USA, 1990.

[David *et al.*, 1993] J.-M. David, J.-P. Krivine, et R. Simmons. Second generation expert systems: A step forward in knowledge engineering. In *Second Generation Expert Systems*, éditeurs J.-M. David, J.-P. Krivine, et R. Simmons, pages 405–427. Springer-Verlag, Berlin, 1993.

[David et Krivine, 1987] J.-M. David et J.-P. Krivine. Utilisation de prototypes dans un système expert de diagnostic : le projet DIVA. In *Actes des 7es Journées Internationales sur les Systèmes Experts et leurs Applications, Avignon*, pages 889–907, 1987.

[David et Krivine, 1988] J.-M. David et J.-P. Krivine. Acquisition de connaissances expertes à partir de situations types. In *Actes des 8es Journées Internationales sur les Systèmes Experts et leurs Applications, Avignon*, pages 45–58, 1988.

[Davis, 1987] H.E. Davis. VIEWS: Multiple perspectives and structured objects in a knowledge representation language. Bachelor and Master of Science, Department of Electrical Engineering and Computer Science, MIT, Reading (MA), USA, 1987.

[De Giacomo et Lenzerini, 1997] G. De Giacomo et M. Lenzerini. A uniform framework for concept definitions in description logics. *Journal of Artificial Intelligence Research*, 6:87–110, 1997.

[Dechter, 1992] R. Dechter. From local to global consistency. *Artificial Intelligence*, 55(1):87–107, 1992.

[Decouchant *et al.*, 1989] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveill, et X. Rousset de Pina. A synchronization mechanism for typed objects. *ACM SIGPLAN Notices*, 24(4):105–107, 1989.

[Dekker, 1993] L. Dekker. La réification de filtres en FROME. In *Actes de la Conférence Représentations Par Objets (RPO'93), La Grande Motte, France*, éditeurs M. Habib et M. Oussalah, pages 23–35. EC2, Nanterre, 1993.

[Dekker, 1994] L. Dekker. *FROME : représentation multiple et classification d'objets avec points de vue*. Thèse d'Informatique, Université des Sciences et Technologies de Lille, 1994.

[Delobel *et al.*, 1991] C. Delobel, C. Lecluse, et P. Richard. *Base de données : des systèmes relationnels aux systèmes à objets*. InterEditions, Paris, 1991.

[Demphlous et Lebastard, 1996] S. Demphlous et F. Lebastard. Intégration de langages et de bases de données à objets : une approche par les méta-objets. In *Actes du Colloque Langages et Modèles à Objets (LMO'96), Leysin, Suisse*, éditeur Y. Dennebouy, pages 108–119. École Polytechnique Fédérale de Lausanne, 1996.

[Desfray, 1996] P. Desfray. *Modélisation par objets : la fin de la programmation*. Méthodes informatiques et pratique des systèmes. Masson, Paris, 1996.

[Dicky *et al.*, 1996] H. Dicky, C. Dony, M. Huchard, et T. Libourel. On automatic class insertion with overloading. In *Proceedings of OOPSLA'96, San Jose (CA), USA*, special issue of ACM SIGPLAN Notices, 31(10), pages 251–267, 1996.

[Dijkstra, 1975] E.W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[Dionne *et al.*, 1993] R. Dionne, E. Mays, et F.J. Oles. The equivalence of model-theoretic and structural subsumption in description logics. In *Proceedings of IJCAI'93, Chambéry, France*, pages 710–716, 1993.

[Dojat et Sayettat, 1994] M. Dojat et C. Sayettat. A realistic model for temporal reasoning in real-time patient monitoring. *Applied Artificial Intelligence*, 10(2):121–143, 1994.

[Donini *et al.*, 1991a] F.-M. Donini, M. Lenzerini, D. Nardi, et W. Nutt. The complexity of concept languages. In *Proceedings of KR'91, Cambridge (MA), USA*, pages 151–162, 1991.

[Donini *et al.*, 1991b] F.-M. Donini, M. Lenzerini, D. Nardi, et W. Nutt. Tractable concept languages. In *Proceedings of IJCAI'91, Sydney, Australia*, pages 458–463, 1991.

[Donini *et al.*, 1994] F.-M. Donini, M. Lenzerini, D. Nardi, et A. Schaerf. Deduction in concept languages: From subsumption to instance checking. *Journal of Logic and Computation*, 4:1–30, 1994.

[Donini *et al.*, 1996] F.-M. Donini, M. Lenzerini, D. Nardi, et A. Schaerf. Reasoning in description logics. In *Principles of Knowledge Representation*, éditeur G. Brewka, pages 191–236. CSLI Publications, Stanford (CA), USA, 1996.

[Donini *et al.*, 1997] F.-M. Donini, M. Lenzerini, D. Nardi, et W. Nutt. The complexity of concept languages. *Information and Computation*, 134(1):1–58, 1997.

[Dony *et al.*, 1992] C. Dony, J. Malenfant, et P. Cointe. Prototype-based languages: From a new taxonomy to constructive proposals and their validation. In *Proceedings of OOPSLA'92, Vancouver, Canada*, éditeur A. Paepcke, special issue of ACM SIGPLAN Notices, 27(10), pages 201–217, 1992.

[Dony, 1989] C. Dony. *Langages à objets et génie logiciel. Applications à la gestion des exceptions et à l'environnement de mise au point*. Thèse d'Informatique, Université Pierre et Marie Curie (Paris VI), 1989.

[Doyle et Patil, 1991] J. Doyle et R.S. Patil. Two theses of knowledge representation: Language restrictions, taxonomic classification, and the utility of representation services. *Artificial Intelligence*, 48(3):261–297, 1991.

[Ducasse, 1997] S. Ducasse. Réification des schémas de conception : une expérience. In *Actes du Colloque Langages et Modèles à Objets (LMO'97), Roscoff, France*, éditeurs R. Ducournau et S. Garlatti, pages 95–110. Hermès, Paris, 1997.

[Ducournau *et al.*, 1995] R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, et A. Napoli. Le point sur l'héritage multiple. *Technique et Science Informatiques*, 14(3):309–345, 1995.

[Ducournau et Habib, 1989] R. Ducournau et M. Habib. La multiplicité de l'héritage dans les langages à objets. *Technique et Science Informatiques*, 8(1):41–62, 1989.

[Ducournau et Quinqueton, 1986] R. Ducournau et J. Quinqueton. YAFOOL : encore un langage à objets à base de *frames*. Rapport Technique 72, INRIA Rocquencourt, 1986.

[Ducournau, 1991] R. Ducournau. *Y3 : YAFOOL, le langage à objets, et YAFEN, l'interface graphique*. Sema Group, Montrouge, 1991.

[Ducournau, 1996a] R. Ducournau. Des langages à objets aux logiques terminologiques : les systèmes classificatoires. Rapport de Recherche 96-030, LIRMM, Montpellier, 1996.
ftp://ftp.lirmm.fr/pub/LIRMM/papers/1996/RRI-Ducournau-96.ps.

[Ducournau, 1996b] R. Ducournau. Les incertitudes de la classification incertaine. In *Actes du Colloque Langages et Modèles à Objets (LMO'96), Leysin, Suisse*, éditeur Y. Dennebouy, pages 183–200. École Polytechnique Fédérale de Lausanne, 1996.

[Ducournau, 1997] R. Ducournau. La compilation de l'envoi de message dans les langages dynamiques. *L'Objet*, 3(3):241–276, 1997.

[Dugerdil, 1988] P. Dugerdil. *Contribution à l'étude de la représentation des connaissances fondée sur les objets. Le langage OBJLOG*. Thèse d'Informatique, Université d'Aix-Marseille II, 1988.

[Durrieu, 1996] G. Durrieu. L'informatique, un outil pédagogique pour enseigner la systématique ? *Biosystema*, 14:109–116, 1996. Société Française de Systématique, Paris.

[Ebcioglu, 1992] K. Ebcioglu. An expert system for harmonizing chorales in the style of J.-S. Bach. In *Understanding Music with AI: Perspectives on Music Cognition*, éditeurs M. Balaban, K. Ebcioglu, et O. Laske, pages 294–333. AAAI Press, Menlo Park (CA), USA, 1992.

[Ellis *et al.*, 1991] C.A. Ellis, S.J. Gibbs, et G.L. Rein. Groupware: Some issues and experiences. *Communications of the ACM*, 34(1):39–58, 1991.

[Ellis et Stroustrup, 1990] M.A. Ellis et B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading (MA), USA, 1990.

[Ellis, 1993] G. Ellis. Efficient retrieval form hierarchies of objects using lattice operations. In *Conceptual Graphs for Knowledge Representation*, éditeurs G. Mineau, B. Moulin, et J.F. Sowa, Lecture Notes in Computer Science 699, pages 274–293. Springer-Verlag, Berlin, 1993.

[Estep *et al.*, 1989] K.W. Estep, A. Hasle, L. Omli, et F. MacIntyre. Linnaeus: Interactive taxonomy using the Macintosh computer and HyperCard. *Bioscience*, 39:635–638, 1989.

[Etherington et Reiter, 1983] D.W. Etherington et R. Reiter. On inheritance hierarchies with exceptions. In *Proceedings of AAAI'83, Washington (DC), USA*, pages 104–108, 1983.

[Etherington, 1986] D. Etherington. *Reasoning with Incomplete Information: Investigation of Non Monotonic Reasoning*. PhD thesis, University of British Columbia, Vancouver, Canada, 1986. Revised version: Reasoning with Incomplete Information, Pitman, London, 1988.

[Euzenat et Rechenmann, 1995] J. Euzenat et F. Rechenmann. SHIRKA, 10 ans, c'est TROPES ? In *Actes du Colloque Langages et Modèles à Objets (LMO'95), Nancy, France*, éditeur A. Napoli, pages 13–34. INRIA Lorraine, Nancy, 1995.

[Euzenat, 1993a] J. Euzenat. Brief overview of T-TREE: the TROPES taxonomy building tool. In *Proceedings of the 4th ASIS SIG/CR Classification Research Workshop, Columbus (OH), USA*, éditeurs P. Smith, C. Beghtol, R. Fidel, et B. Kwasnik, pages 69–87, 1993. Également publié dans *Advances in Classification Research 4, Learning Information*, Medford (NJ), USA, 1994. ftp://ftp.inrialpes.fr/pub/sherpa/publications/euzenat93c.ps.gz.

[Euzenat, 1993b] J. Euzenat. Définition abstraite de la classification et son application aux taxonomies d'objets. In *Actes de la Conférence Représentations Par Objets (RPO'93), La Grande Motte, France*, éditeurs M. Habib et M. Oussalah, pages 235–246. EC2, Nanterre, 1993.

[Euzenat, 1994] J. Euzenat. Classification dans les représentations par objets : produits de systèmes classificatoires. In *Actes du 9e Congrès AFCET Reconnaissance des Formes et Intelligence Artificielle, Paris*, pages 185–196, 1994.

[Fahlman, 1979] S.E. Fahlman. *NETL: A System for Representing and Using Real-World Knowledge*. MIT Press, Cambridge (MA), USA, 1979.

[Felsenstein, 1973] J. Felsenstein. Maximum likelihood estimation of evolutionary trees from continuous characters. *American Journal of Human Genetics*, 25:471–492, 1973.

[Ferber et Carle, 1991] J. Ferber et P. Carle. Actors and agents as reflective concurrent objects: A Mering IV perspective. *IEEE Transactions on Systems, Man and Cybernetics*, 21(6):1420–1436, 1991.

[Ferber et Volle, 1988] J. Ferber et P. Volle. Using coreference in object-oriented representations. In *Proceedings of ECAI'88, Munich, Germany*, pages 238–240, 1988.

[Ferber, 1989a] J. Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings of OOPSLA'89, New Orleans (LA), USA*, special issue of ACM SIGPLAN Notices, 24(10), pages 317–326, 1989.

[Ferber, 1989b] J. Ferber. *Objets et agents : une étude des structures de représentation et de communications en intelligence artificielle*. Thèse de Doctorat d'État, Université Pierre et Marie Curie (Paris VI), 1989.

[Ferber, 1990] J. Ferber. *Conception et programmation par objets*. Hermès, Paris, 1990.

[Ferrandina *et al.*, 1995] F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, et J. Madec. Schema and database evolution in the $O_2$ object database system. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95), Zurich, Switzerland*, pages 170–181. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1995.

[Fikes et Kehler, 1985] R. Fikes et T. Kehler. The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9):904–920, 1985.

[Filman, 1988] R.E. Filman. Reasoning with worlds and truth maintenance in a knowledge-based programming environment. *Communications of the ACM*, 31(4):382–401, 1988.

[Finin, 1986] T.W. Finin. Interactive classification: A technique for acquiring and maintaining knowledge bases. *Proceedings of the IEEE*, 74(10):1414–1421, 1986.

[Fisher et Langley, 1986] D. Fisher et P. Langley. Conceptual clustering and its relation to numerical taxonomy. In *Artificial Intelligence and Statistics*, éditeur W. Gale, pages 77–116. Addison-Wesley, Reading (MA), USA, 1986.

[Fitting, 1990] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag, Berlin, 1990.

[Foote et Johnson, 1989] B. Foote et R.E. Johnson. Reflective facilities in Smalltalk-80. In *Proceedings of OOPSLA'89, New Orleans (LA), USA*, special issue of ACM SIGPLAN Notices, 24(10), pages 327–335, 1989.

[Fornarino et Pinna, 1990] M. Fornarino et A.-M. Pinna. *Un modèle objet logique et relationnel : le langage OTHELO*. Thèse d'Informatique, Université de Nice Sophia Antipolis, 1990.

[Freeman-Benson *et al.*, 1990] B.N. Freeman-Benson, J. Maloney, et A. Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, 1990.

[Freeman-Benson, 1990] B.N. Freeman-Benson. KALEIDOSCOPE: Mixing objects, constraints and imperative programming. *ACM SIGPLAN Notices*, 25(10):77–88, 1990.

[Freuder *et al.*, 1995] E. Freuder, R. Dechter, M. Ginsberg, B. Selman, et E. Tsang. Systematic versus stochastic constraint satisfaction. In *Proceedings of IJCAI'95, Montréal, Canada*, pages 2027–2032, 1995.

[Frolund, 1992] S. Frolund. Inheritance of synchronization constraints in concurrent object-oriented programming languages. In *Proceedings of ECOOP'92, Utrecht, The Netherlands*, éditeur O. Lehrmann Madsen, Lecture Notes in Computer Science 615, pages 185–196. Springer-Verlag, Berlin, 1992.

[Fron, 1994] A. Fron. *Programmation par contraintes*. Addison-Wesley France, Paris, 1994.

[Frost et Dechter, 1994] D. Frost et R. Dechter. In search of the best constraint satisfaction search. In *Proceedings of AAAI'94, Seattle (WA), USA*, pages 301–306, 1994.

[Furmento et Baude, 1995] N. Furmento et F. Baude. Design and implementation of communications for the C++// system. Rapport de Recherche RR 95-50, Laboratoire I3S, Sophia Antipolis, 1995.

[Gamma *et al.*, 1994] E. Gamma, R. Helm, R. Johnson, et J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading (MA), USA, 1994.

[Gammerman *et al.*, 1986] A.J. Gammerman, B. Skullerud, et W. Aitchinson. SYSEX: An expert system for biological identification. In *Proceedings of Applications of Artificial Intelligence IV, Innsbruck, Austria*, éditeur J.S. Gilmore, SPIE 657, pages 34–38, 1986.

[Gançarski et Jomier, 1994] S. Gançarski et G. Jomier. Managing entity versions within their context: A formal approach. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA'94), Athens, Greece*, éditeur D. Karagiannis, pages 400–409. Springer-Verlag, Berlin, 1994.

[Garbinato *et al.*, 1994] B. Garbinato, R. Guerraoui, et K.R. Mazouni. Distributed programming in GARF. In *Object-Based Distributed Programming*, éditeurs R. Guerraoui, O. Nierstrasz, et M. Riveill, Lecture Notes in Computer Science 791, pages 225–239. Springer-Verlag, Berlin, 1994.

[Garbinato *et al.*, 1995] B. Garbinato, R. Guerraoui, et K.R. Mazouni. Implementation of the GARF replicated objects platform. *Distributed Systems Engineering Journal*, 2(1):14–27, 1995.

[Gardarin et Valduriez, 1990] G. Gardarin et P. Valduriez. *SGBD avancés. Base de données objets, déductives, réparties*. Eyrolles, Paris, 1990.

[Gardarin, 1993] G. Gardarin. *Maîtriser les bases de données. Modèles et langages*. Eyrolles, Paris, 1993.

[Gardarin, 1994] G. Gardarin. Convergence des modèles relationnels et à objets. *Ingénierie des Systèmes d'Information*, 2(3):317–346, 1994.

[Garey et Johnson, 1979] M.R. Garey et D.S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco (CA), USA, 1979.

[Gaschnig, 1979] J. Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms*. PhD thesis, Carnegie-Mellon University, Pittsburgh (PA), USA, 1979.

[Gasser et Briot, 1992] L. Gasser et J.-P. Briot. Object-based concurrent programming and distributed artificial intelligence. In *Distributed Artificial Intelligence: Theory and Praxis*, éditeurs N.M. Avouris et L. Gasser, pages 81–107. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1992.

[Gautier *et al.*, 1996] M. Gautier, G. Masini, et K. Proch. *Cours de programmation par objets. Applications à Eiffel et comparaison avec C++*. Masson, Paris, 1996.

[Gehani et Roome, 1989] N. Gehani et W.D. Roome. *The Concurrent C Programming Language*. Silicon Press, Los Gatos (CA), USA, 1989.

[Gehani, 1984] N. Gehani. *Ada Concurrent Programming*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1984.

[Genesereth et Fikes, 1992] M. Genesereth et R. Fikes. Knowledge Interchange Format, version 3.0. Reference manual. Research Report Logic-92-1, Stanford University, Palo Alto (CA), USA, 1992.

[Gensel *et al.*, 1993] J. Gensel, P. Girard, et O. Schmeltzer. Integrating constraints, composite objects and tasks in a knowledge representation system. In *Proceedings of the 5th IEEE International Conference on Tools with Artificial Intelligence, Cambridge (MA), USA*, pages 127–130, 1993.

[Gensel et Girard, 1992] J. Gensel et P. Girard. Expression d'un modèle de tâches à l'aide d'une représentation par objets. In *Actes de la Conférence Représentations Par Objets (RPO'92), La Grande Motte, France*, pages 225–236. EC2, Nanterre, 1992.

[Gensel, 1995] J. Gensel. *Contraintes et représentation des connaissances par objets. Application au modèle TROPES*. Thèse d'Informatique, Université Joseph Fourier, Grenoble, 1995.

[Gervet, 1997] C. Gervet. Interval propagation to reason about sets : Definition and implementation of a practical language. *International Journal of Constraints*, 1(1):191–246, 1997.

[Giliberti *et al.*, 1997] V. Giliberti, M. Gorgoglione, et R. Vitulli. An innovative model for object-oriented costs estimating. In *Actes du Colloque Langages et Modèles à Objets (LMO'97), Roscoff, France*, éditeurs R. Ducournau et S. Garlatti, pages 243–258. Hermès, Paris, 1997.

[Gilmour, 1951] J.S.L. Gilmour. The developement of taxonomic theory since 1851. *Nature*, 168:400–402, 1951.

[Gindre et Sada, 1989] C. Gindre et F. Sada. A development in Eiffel: Design and implementation of a network simulator. *Journal of Object-Oriented Programming*, 2(1):27–33, 1989.

[Ginsberg, 1991] M.L. Ginsberg. Knowledge Interchange Format: The KIF of death. *The AI Magazine*, 12(3):57–63, 1991.

[Ginsberg, 1993] M.L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[Girard, 1995] P. Girard. *Construction hypothétique d'objets complexes*. Thèse d'Informatique, Université Joseph Fourier, Grenoble, 1995.

[Gochet et Gribomont, 1991] P. Gochet et P. Gribomont. *Logique*, volume 1. Hermès, Paris, 1991.

[Godin *et al.*, 1995] R. Godin, G. Mineau, et R. Missaoui. Incremental structuring of knowledge bases. In *Proceedings of the 1st International Symposium on Knowledge Retrieval, Use, and Storage for Efficiency (KRUSE'95), Santa Cruz (CA), USA*, éditeurs G. Ellis, R.A. Levinson, A. Fall, et V. Dahl, pages 179–193. Department of Computer Science, University of California at Santa Cruz, 1995.

[Goldberg et Robson, 1983] A. Goldberg et D. Robson. *Smalltalk-80, the Language and its Implementation*. Addison-Wesley, Reading (MA), USA, 1983.

[Goldberg, 1984] A. Goldberg. *Smalltalk-80, the Interactive Programming Environment*. Addison-Wesley, Reading (MA), USA, 1984.

[Gommaa, 1989] H. Gommaa. Structuring criteria for real-time system design. In *Proceedings of the 11th International Conference on Software Engineering, Pittsburgh (PA), USA*, pages 290–301, 1989.

[Gonzalez-Gomez, 1996] M. Gonzalez-Gomez. *Représentation des connaissances approximatives et classification approchée d'instances dans un langage de schémas : le système CAIN*. Thèse d'Informatique, Université d'Aix-Marseille III, 1996.

[Goodwin, 1979] J.W. Goodwin. Taxonomic programming with KL-ONE. Research Report, Informatics Laboratory, Linköping University, Sweden, 1979.

[Gordon, 1987] A.D. Gordon. A review of hierarchical classification. *Journal of the Royal Statistical Society (A)*, 150(2):119–137, 1987.

[Gore, 1996] J. Gore. *Object Structures: Building Object-Oriented Components*. Addison-Wesley, Reading (MA), USA, 1996.

[Gower, 1974] J.C. Gower. Maximal predictive classification. *Biometrics*, 30:643–654, 1974.

[Graham, 1994] I. Graham. *Object-Oriented Methods, Second Edition*. Addison--Wesley, Reading (MA), USA, 1994.

[Granger, 1986] C. Granger. Fuzzy reasoning in a knowledge-based system for object classification. In *Proceedings of ECAI'86, Brighton, UK*, pages 163–170, 1986.

[Granger, 1988] C. Granger. An application of possibility theory to object recognition. *Fuzzy Sets and Systems*, 28(4):351–362, 1988.

[Gransart, 1995] C. Gransart. *BOX : un modèle et un langage à objets pour la programmation parallèle et distribuée*. Thèse d'Informatique, Université de Lille I, 1995.

[Grass et Campbell, 1986] J. E. Grass et R.H. Campbell. Mediators: A synchronization mechanism. In *Proceedings of the 6th International Conference on Distributed Computing Systems, Washington (DC), USA*, pages 468–477, 1986.

[Gray et Reuter, 1993] J. Gray et A. Reuter. *Transaction Processing*. Morgan Kaufmann Publishers, San Mateo (CA), USA, 1993.

[Greussay, 1985] P. Greussay. Musique, sens et pensée. In *Quoi ? Quand ? Comment ? La recherche musicale*, éditeur IRCAM, pages 165–183. Christian Bourgois, Paris, 1985.

[Guerraoui *et al.*, 1992] R. Guerraoui, R. Capobianchi, A. Lanusse, et P. Roux. Nesting actions through asynchronous message passing: The ACS protocol. In *Proceedings of ECOOP'92, Utrecht, The Netherlands*, éditeur O. Lehrmann Madsen, Lecture Notes in Computer Science 615, pages 170–184. Springer-Verlag, Berlin, 1992.

[Guerraoui *et al.*, 1994] éditeurs R. Guerraoui, O. Nierstrasz, et M. Riveill. *Object-Based Distributed Programming*. Lecture Notes in Computer Science 791. Springer-Verlag, Berlin, 1994.

[Guerraoui et Schiper, 1995] R. Guerraoui et A. Schiper. Transaction model vs virtual synchrony model: Bridging the gap. In *Distributed Systems: From Theory to Practice*, éditeurs K. Birman, F. Cristian, F. Mattern, et A. Schiper, Lecture Notes in Computer Science 938, pages 121–132. Springer-Verlag, Berlin, 1995.

[Guerraoui, 1995a] R. Guerraoui. Les langages concurrents à objets. *Technique et Science Informatiques*, 14(8):945–972, 1995.

[Guerraoui, 1995b] R. Guerraoui. Modular atomic objects. *Theory and Practice of Object Systems*, 1(2):89–100, 1995.

[Guinaldo, 1996] O. Guinaldo. *Etude d'un gestionnaire d'ensembles de graphes conceptuels*. Thèse d'Informatique, Université des Sciences et Techniques du Languedoc, Montpellier, 1996.

[Habert et Fleury, 1993] B. Habert et S. Fleury. Suivi fin de l'analyse automatique du langage naturel basé sur l'héritage et la discrimination multiple. In *Actes de la Conférence Représentations Par Objets (RPO'93), La Grande Motte, France*, éditeurs M. Habib et M. Oussalah, pages 77–88. EC2, Nanterre, 1993.

[Habert, 1995] B. Habert. *Objectif CLOS*. Masson, Paris, 1995.

[Halbert et O'Brien, 1987] D.C. Halbert et P.D. O'Brien. Using types and inheritance in object-oriented languages. In *Proceedings of ECOOP'87, Paris, France*, éditeurs J. Bézivin, J.-M. Hullot, P. Cointe, et H. Lieberman, Lecture Notes in Computer Science 276, pages 23–34. Springer-Verlag, Berlin, 1987.

[Halstead, 1985] R. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[Hammer et McLeod, 1981] M. Hammer et D. McLeod. Database description with SDM: A semantic data model. *ACM Transactions on Database Systems*, 6(3):351–386, 1981.

[Hao *et al.*, 1997] J.-K. Hao, P. Galinier, et M. Habib. Méthodes heuristiques pour l'optimisation combinatoire et l'affectation sous contraintes. In *Actes des 6es Journées Nationales du PRC-GDR Intelligence Artificielle*, éditeurs S. Pesty et P. Siegel, pages 107–144. Hermès, Paris, 1997.

[Haralick et Elliot, 1980] R.M. Haralick et G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980.

[Harito Shteto, 1997] M. Harito Shteto. *Technique à objets pour les algorithmes de configuration du réseau téléphonique commuté*. Thèse d'Informatique, Université Pierre et Marie Curie (Paris VI), 1997.

[Harris, 1986] D.R. Harris. A hybrid structured object and constraint representation system. In *Proceedings of AAAI'86, Philadelphia (PA), USA*, pages 986–990, 1986.

[Hautamäki, 1986] A. Hautamäki. Points of view and their logical analysis. *Acta Philosophica Fennica*, 41:3–126, 1986.

[Heinsohn *et al.*, 1994] J. Heinsohn, D. Kudenko, B. Nebel, et H.-J. Profitlich. An empirical analysis of terminological representation systems. *Artificial Intelligence*, 68(2):367–397, 1994.

[Heitz et Labreuille, 1988] M. Heitz et B. Labreuille. Design and development of distributed software using hierarchical object-oriented design and Ada. In *Ada in Industry — Proceedings of Ada Europe International Conference, Munich, Germany*, pages 143–156. Cambridge University Press, Cambridge, UK, 1988.

[Henderson-Sellers et Edwards, 1990] B. Henderson-Sellers et J.M. Edwards. The object-oriented systems life cycle. *Communications of the ACM*, 33(9):142–159, 1990.

[Hennig, 1966] W. Hennig. *Phylogenetic Systematics*. University of Illinois Press, Urbana-Champaign, 1966.

[Hewitt, 1977] C.E. Hewitt. Viewing control structure as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, 1977.

[Hoare, 1978] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Hoare, 1985] C.A.R. Hoare. *Communications Sequential Processes*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1985. Traduction française : [Hoare, 1987].

[Hoare, 1987] C.A.R. Hoare. *Processus séquentiels communiquants*. Masson, Paris, 1987. Traduction française de [Hoare, 1985].

[Hofstadter, 1985] D. Hofstadter. *Gödel, Escher, Bach, les brins d'une guirlande éternelle*. InterEditions, Paris, 1985.

[Hoppe *et al.*, 1993] T. Hoppe, C. Kindermann, J.J. Quantz, A. Schmiedel, et M. Fischer. Back V5—Tutorial & manual. KIT-Report 100, Technische Universität, Berlin, 1993.

[Hull et King, 1987] R. Hull et R. King. Semantic database modeling: Survey, applications and research issues. *ACM Computing Surveys*, 19(3):202–260, 1987.

[Hureau, 1994] éditeur J.-C. Hureau. *Fishes of the Northeastern Atlantic and the Mediterranean (CD-ROM)*. ETI-UNESCO, Amsterdam, 1994.

[Hyvönen, 1992] E. Hyvönen. Constraint reasoning based on interval arithmetic: The tolerance propagation approach. *Artificial Intelligence*, 58(1–3):71–112, 1992.

[Ichbiah *et al.*, 1979] J.D. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Brueckner, O. Roubine, et B.A. Wichman. Ada reference manual and rationale for the design of the Ada programming language. *ACM SIGPLAN Notices*, 14(6):159–198, 1979.

[ILOG, 1991a] ILOG S.A., Gentilly, France. *PECOS version 1.1 : manuel de référence*, 1991.

[ILOG, 1991b] ILOG S.A., Gentilly, France. *SMECI : manuel de référence*, 1991.

[ILOG, 1991c] ILOG S.A., Gentilly, France. *SOLVER version 3.0 : manuel de référence*, 1991.

[Ingalls, 1978] D.H.H. Ingalls. The smalltalk-76 programming system design and implementation. In *Proceedings of the 5th International Conference on the Principles Of Programming Languages (POPL'78), Tucson (AR), USA*, pages 9–17, 1978.

[INMOS Limited, 1988] éditeur INMOS Limited. *Occam 2 Reference Manual*. C.A.R. Hoare Series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1988.

[Ishikawa et Tokoro, 1987] Y. Ishikawa et M. Tokoro. Orient84/K: An object-oriented concurrent programming language for knowledge representation. In *Object-Oriented Concurrent Programming*, éditeurs A. Yonezawa et M. Tokoro, pages 159–198. MIT Press, Cambridge (MA), USA, 1987.

[Itasca Systems, 1993] Itasca Systems. OODBMS feature checklist. Rapport Technique TM-92-001, Itasca Systems, Inc., Minneapolis (MN), USA, 1993.

[Jacobson *et al.*, 1992] I. Jacobson, M. Christerson, P. Jonsson, et G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading (MA), USA, 1992.

[Jambaud, 1996] P. Jambaud. *Le dialogue comme processus de résolution de problème. Une application en chimie organique*. Thèse d'Informatique, Université des Sciences et Techniques du Languedoc, Montpellier, 1996.

[Jeffrey, 1973] C. Jeffrey. *Biological Nomenclature*. Arnold, London, 1973.

[Jézéquel, 1993a] J.-M. Jézéquel. EPEE: An Eiffel environment to program distributed memory parallel computers. *Journal of Object-Oriented Programming*, 6(2):48–54, 1993.

[Jézéquel, 1993b] J.-M. Jézéquel. Transparent parallelisation through reuse: Between a compiler and a library approach. In *Proceedings of ECOOP'93, Kaiserslautern, Germany*, éditeur O. Nierstrasz, Lecture Notes in Computer Science 707, pages 384–405. Springer-Verlag, Berlin, 1993.

[Jézéquel, 1996] J.-M. Jézéquel. *Object-Oriented Software Engineering with Eiffel*. Addison-Wesley, Reading (MA), USA, 1996.

[Jul, 1994] E. Jul. Separation of distribution and objects. In *Object-Based Distributed Programming*, éditeurs R. Guerraoui, O. Nierstrasz, et M. Riveill, Lecture Notes in Computer Science 791, pages 47–55. Springer-Verlag, Berlin, 1994.

[Kafura et Lee, 1989a] D.G. Kafura et K.H. Lee. Inheritance in actor-based concurrent object-oriented languages. In *Proceedings of ECOOP'89, Nottingham, UK*, éditeur S. Cook, pages 131–145. Cambridge University Press, Cambridge, UK, 1989.

[Kafura et Lee, 1989b] D.G. Kafura et K.H. Lee. Inheritance in actor-based concurrent object-oriented languages. *The Computer Journal*, 32(4):297–304, 1989.

[Karaorman et Bruno, 1993] M. Karaorman et J. Bruno. Introducing concurrency to a sequential language. *Communications of the ACM (special issue, Concurrent Object-Oriented Programming, B. Meyer, editor)*, 36(9):103–116, 1993.

[Karp *et al.*, 1995] P.D. Karp, K.L. Myers, et T. Gruber. The generic frame protocol. In *Proceedings of IJCAI'95, Montréal, Canada*, pages 768–774, 1995.

[Karp, 1993] P.D. Karp. The design space of frame knowledge representation systems. Technical Note 520, SRI International, Menlo Park (CA), USA, 1993.

[Keene, 1989] S.E. Keene. *Object-Oriented Programming in Common Lisp. A Programmer's Guide to CLOS*. Addison-Wesley, Reading (MA), USA, 1989.

[Kemper et Kossmann, 1993] A. Kemper et D. Kossmann. Adaptable pointer swizzling strategies in object bases. In *Proceedings of the 9th IEEE International Conference on Data Engineering (ICDE'93), Vienna, Austria*, pages 155–162, 1993.

[Kemper et Moerkotte, 1994] A. Kemper et G. Moerkotte. *Object-Oriented Database Management*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1994.

[Kendrick, 1972] B. Kendrick. Computer graphics in fungal identification. *Canadian Journal of Botany*, 50:2171–2175, 1972.

[Kernighan et Ritchie, 1988] B.W. Kernighan et D.M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1988.

[Kernighan et Ritchie, 1990] B.W. Kernighan et D.M. Ritchie. *Le langage C, norme ANSI*. Masson, Paris, 1990.

[Khoshafian, 1991] S. Khoshafian. *Object-Orientation: Concepts, Languages, Databases, User Interfaces*. John Wiley & Sons, New York (NY), USA, 1991.

[Kiczales *et al.*, 1991] G. Kiczales, J. des Rivieres, et D.G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, Cambridge (MA), USA, 1991.

[Kiczales, 1994] G. Kiczales. Foil for the Workshop on Open Implementation. http://www.parc.xerox.com/spl/projects/oi/workshop-94/foil/main.html, 1994.

[Kifer *et al.*, 1995] M. Kifer, G. Lausen, et J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[Kim *et al.*, 1989] W. Kim, N. Ballou, H.T. Chou, J.-F. Garza, et D. Woelk. The GemStone data management system. In *Features of the ORION Object-Oriented Database System*, éditeurs W. Kim et F.H. Lochovsky, pages 251–282. Addison-Wesley, Reading (MA), USA, 1989.

[Kim *et al.*, 1990] W. Kim, J.-F. Garza, N. Ballou, et D. Woelk. Architecture of the ORION next-generation database system supporting shared and private database. *ACM Transactions on Information Systems*, 2(1):109–124, 1990.

[Kim et Chou, 1988] W. Kim et H.T. Chou. Versions of schema for object-oriented databases. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB'88), Los Angeles (CA), USA*, pages 148–159. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1988.

[Kim, 1990] W. Kim. *Introduction to Object-Oriented Databases*. MIT Press, Cambridge (MA), USA, 1990.

[Kindermann, 1992] C. Kindermann. Retraction in terminological knowledge bases. In *Proceedings of ECAI'92, Vienna, Austria*, pages 420–424, 1992.

[Kirkerud, 1989] B. Kirkerud. *Object-Oriented Programming with Simula*. International Computer Science Series. Addison-Wesley, Reading (MA), USA, 1989.

[Kishimoto *et al.*, 1995] Y. Kishimoto, N. Kotaka, et S. Honiden. Adapting object-communication methods dynamically. *IEEE Software*, 12(3):65–74, 1995.

[Kleiber, 1991] G. Kleiber. Prototype et prototypes : encore une affaire de famille. In *Sémantique et cognition — Catégories, prototypes, typicalité*, éditeur D. Dubois, pages 103–129. Editions du CNRS, Paris, 1991.

[Koenig et Stroustrup, 1989] A. Koenig et B. Stroustrup. C++: As close to C as possible—But no closer. *The C++ Report*, 1989.

[Kökény, 1994] T. Kökény. *Satisfaction de contraintes dans un environnement orienté objets*. Thèse d'Informatique, Université des Sciences et Techniques du Languedoc, Montpellier, 1994.

[Kowalski et Sergot, 1986] R. Kowalski et M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[Koyré, 1973] A. Koyré. *Études d'histoire de la pensée scientifique*. Gallimard, Paris, 1973.

[Kristensen *et al.*, 1987] B.B. Kristensen, O. Lehrmann Madsen, B. Møller-Pedersen, et K. Nygaard. Classification of actions or inheritance also for methods. In *Proceedings of ECOOP'87, Paris, France*, éditeurs J. Bézivin, J.-M. Hullot, P. Cointe, et H. Lieberman, Lecture Notes in Computer Science 276, pages 109–118. Springer-Verlag, Berlin, 1987.

[Laasch *et al.*, 1991] C. Laasch, M.H. Scholl, et M. Tresch. Updatable views in object-oriented databases. In *Proceedings of the 2nd Conference on Deductive and Object-Oriented Databases (DOOD'91), Munich, Germany*, pages 189–207, 1991.

[Lacroix *et al.*, 1997] Z. Lacroix, C. Delobel, et P. Brèche. Object views constructed with an object algebra. In *Actes du 12e Colloque Bases de Données Avancées (BDA'97), Grenoble, France*, pages 219–239, 1997.

[Lahire et Jugant, 1995] P. Lahire et J.-M. Jugant. Lessons learned with Eiffel 3: The K2 project. In *Proceedings of TOOLS-USA'95, Santa Barbara, (CA), USA*, éditeur R. Ege, pages 207–215. Prentice-Hall, Englewood Cliffs (NJ), USA, 1995.

[Lahire, 1992] P. Lahire. *Conception et réalisation d'un modèle de persistance pour le langage Eiffel*. Thèse d'Informatique, Université de Nice Sophia Antipolis, Laboratoire I3S, 1992.

[Lalande, 1926] A. Lalande. *Vocabulaire technique et critique de la philosophie*. Presses Universitaires de France, Paris, 1926.

[LaLonde *et al.*, 1986] W.R. LaLonde, D.A. Thomas, et J.R. Pugh. An exemplar-based Smalltalk. In *Proceedings of OOPSLA'86, Portland (OR), USA*, éditeur N.K. Meyrowitz, special issue of ACM SIGPLAN Notices, 21(11), pages 322–330, 1986.

[Lalonde, 1989] W.R. Lalonde. Designing families of data types using examplars. *ACM Transactions on Programming Languages and Systems*, 11(2):212–248, 1989.

[Lamarck, 1778] J.-B. Lamarck. *Flore françoise, 1re édition*. Imprimerie Royale, Paris, 1778.

[Lassez *et al.*, 1993] J.-L. Lassez, T. Huyinh, et K. McAloon. Simplifications and elimination of redundant linear arithmetic constraints. In *Constraint Logic Programming: Selected Research*, éditeurs F. Benhamou et A. Colmerauer, pages 73–88. MIT Press, Cambridge (MA), USA, 1993.

[Le Verrand, 1982] D. Le Verrand. *Le langage Ada. Manuel d'évaluation*. Dunod Informatique, Paris, 1982.

[Lea *et al.*, 1993] R. Lea, C. Jacquemot, et E. Pillevesse. COOL: System support for distributed programming. *Communications of the ACM (special issue, Concurrent Object-Oriented Programming, B. Meyer, editor)*, 36(9):37–47, 1993.

[Lebbe, 1991] J. Lebbe. *Représentation des concepts en biologie et médecine (introduction à l'analyse des connaissances et à l'identification assistée par ordinateur)*. Thèse, Université Pierre et Marie Curie (Paris VI), 1991.

[Lebbe, 1995] J. Lebbe. Systématique et informatique. In *Systématique et Biodiversité*, éditeur T. Bourgoin, Biosystema 13, pages 71–79. Société Française de Systématique, Paris, 1995.

[Legeard *et al.*, 1993] B. Legeard, H. Lombardi, E. Legros, et M. Hibti. Constraint satisfaction approach to set unification. In *Actes des 13$^{es}$ Journées Internationales sur les Systèmes Experts et leurs Applications, Avignon*, pages 265–276 (volume 1), 1993.

[Legendre, 1996] S. Legendre. *Modèles en dynamique des populations : mise en œuvre informatique*. Thèse, Université Pierre et Marie Curie (Paris VI), 1996.

[Lerner et Habermann, 1990] B. Lerner et A. Habermann. Beyond schema evolution to database reorganisation. *ACM SIGPLAN Notices*, 25(10):67–76, 1990.

[Leroy, 1992] X. Leroy. *Typage polymorphe d'un langage algorithmique*. Thèse d'Informatique, Université de Paris VII, 1992.

[Lescaudron, 1992] L. Lescaudron. *Prototypage d'environnements de programmation pour les langages à objets concurrents : une réalisation en Smalltalk-80 pour Actalk*. Thèse d'Informatique, Université Pierre et Marie Curie (Paris VI), 1992. LITP, Rapport de Recherche TH93.11.

[Levesque et Brachman, 1987] H.J. Levesque et R.J. Brachman. Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, 3(2):78–93, 1987.

[Levinson, 1992] R. Levinson. Pattern associativity and the retrieval of semantic networks. *Computers & Mathematics with Applications*, 23(6–9):573–600, 1992.

[Levinson, 1994] R.A. Levinson. UDS: A Universal Data Structure. In *Conceptual Structures: Current Practices. Proceedings of the 2nd International Conference on Conceptual Structures, The University of Maryland at College Park (MD), USA*, éditeurs W.M. Tepfenhart, J.P. Dick, et J.F. Sowa, Lecture Notes in Artificial Intelligence 835, pages 230–250. Springer-Verlag, Berlin, 1994.

[Levy et Rousset, 1996] A.Y. Levy et M.-C. Rousset. CARIN: A representation language combining Horn rules and description logics. In *Proceedings of ECAI'96, Budapest, Hungary*, pages 323–327, 1996.

[Libera, 1996] A. de Libera. *La querelle des universaux, de Platon à la fin du Moyen-Âge*. Le Seuil, Paris, 1996.

[Libourel, 1993] T. Libourel. Evolutivité dans les SGBD et les systèmes d'objets. In *Actes du Congrès AFCET Systèmes Flexibles, Versailles, France*, pages 125–135, 1993.

[Lieberherr et Holland, 1989] K.J. Lieberherr et I. Holland. Formulations and benefits of the law of DEMETER. *ACM SIGPLAN Notices*, 24(3):67–78, 1989.

[Lieberman, 1981] H. Lieberman. A preview of Act 1. AI Memo 625, Artificial Intelligence Laboratory, Cambridge (MA), USA, 1981.

[Lieberman, 1986] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of OOPSLA'86, Portland (OR), USA*, special issue of ACM SIGPLAN Notices, 21(11), pages 214–223, 1986.

[Lieberman, 1987] H. Lieberman. Concurrent object-oriented programming in Act 1. In *Object-Oriented Concurrent Programming*, éditeurs A. Yonezawa et M. Tokoro, pages 9–36. MIT Press, Cambridge (MA), USA, 1987.

[Lieberman, 1990] H. Lieberman. *Habilitation à diriger des recherches*. Institut Blaise Pascal, Université Pierre et Marie Curie (Paris VI), 1990.

[Lipkis, 1982] T. Lipkis. A KL-ONE classifier. In *Proceedings of the KL-ONE Workshop, Jackson (NH), USA*, éditeurs J.G. Schmolze et R.J. Brachman, BBN Report 4842/Fairchild Technical Report 618, pages 126–143, 1982.

[Lippman, 1991] S.B. Lippman. *A C++ Primer, Second Edition*. Addison-Wesley, Reading (MA), USA, 1991.

[Liskov et Guttag, 1990] B. Liskov et J. Guttag. *La maîtrise du développement de logiciel : abstraction et spécification*. Les Editions d'Organisation, Paris, 1990.

[Liskov et Scheifler, 1983] B. Liskov et R. Scheifler. Guardians and actions: Linguistics support for robust, distributed systems. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, 1983.

[Lopez *et al.*, 1993] G. Lopez, B. Freeman-Benson, et A. Borning. KALEIDOSCOPE: A constraint imperative programming language. Technical Report 93-09-04, Department of Computer Science and Engineering, University of Washington, Seattle (WA), USA, 1993.

[Lorenz et Kidd, 1994] M. Lorenz et J. Kidd. *Object-Oriented Software Metrics*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1994.

[Loveland, 1978] D.W. Loveland. *Automated Theorem Proving: A Logical Basis*. North-Holland, Amsterdam, 1978.

[MacGregor, 1991] R. MacGregor. The evolving technology of classification-based knowledge representation systems. In *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, éditeur J.F. Sowa, pages 385–400. Morgan Kaufmann Publishers, San Mateo (CA), USA, 1991.

[Mackworth *et al.*, 1985] A.K. Mackworth, J.A. Mulder, et W.S. Havens. Hierarchical arc consistency: Exploiting structured domains in constraint satisfaction problems. *Computational Intelligence*, 1(1):118–126, 1985.

[Mackworth, 1977] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[MacLennan, 1982] B. MacLennan. Values and objects in programming languages. *ACM SIGPLAN Notices*, 17(12):70–79, 1982.

[Maes, 1987] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87, Orlando (FL), USA*, special issue of ACM SIGPLAN Notices, 22(12), pages 147–155, 1987.

[Maffeis, 1995] S. Maffeis. *Run-Time Support for Object-Oriented Distributed Programming*. PhD thesis, Université de Zurich, Suisse, 1995.

[Magnan, 1994] M. Magnan. *Réutilisation de composants : les exceptions dans les objets composites*. Thèse d'Informatique, Université des Sciences et Techniques du Languedoc, Montpellier, 1994.

[Maier *et al.*, 1986] D. Maier, A. Otis, et A. Purdy. Development of an object-oriented DBMS. In *Proceedings of OOPSLA'86, Portland (OR), USA*, special issue of ACM SIGPLAN Notices, 21(11), pages 472–482, 1986.

[Malenfant, 1995] J. Malenfant. On the semantic diversity of delegation-based programming languages. In *Proceedings of OOPSLA'95, Austin (TX), USA*, special issue of ACM SIGPLAN Notices, 30(10), pages 215–230, 1995.

[Malenfant, 1996] J. Malenfant. Abstraction et encapsulation en programmation par prototypes. *Technique et Science Informatiques*, 15(6):709–734, 1996.

[Maloney *et al.*, 1989] J.H. Maloney, A. Borning, et B.N. Freeman-Benson. Constraint technology for user-interface in ThingLab II. Technical Report 89-05-02, Department of Computer Science and Engineering, University of Washington, Seattle (WA), USA, 1989.

[Margolis, 1983] C.Z. Margolis. Uses of clinical algorithms. *Journal of the American Medical Association*, 249:627–632, 1983.

[Mariño *et al.*, 1990] O. Mariño, F. Rechenmann, et P. Uvietta. Multiple perspectives and classification mechanism in object-oriented representation. In *Proceedings of ECAI'90, Stockholm, Sweden*, pages 425–430, 1990.

[Mariño, 1993] O. Mariño. *Raisonnement classificatoire dans une représentation à objets multi-points de vue*. Thèse d'Informatique, Université Joseph Fourier, Grenoble, 1993.

[Masini *et al.*, 1989] G. Masini, A. Napoli, D. Colnet, D. Léonard, et K. Tombre. *Les langages à objets*. InterEditions, Paris, 1989. Traduction anglaise : [Masini *et al.*, 1991].

[Masini *et al.*, 1991] G. Masini, A. Napoli, D. Colnet, D. Léonard, et K. Tombre. *Object-Oriented Languages*. Academic Press, London, 1991. Traduction anglaise de [Masini *et al.*, 1989].

[Masuhara *et al.*, 1995] H. Masuhara, S. Matsuoka, K. Asai, et A. Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *Proceedings of OOPSLA'95, Austin (TX), USA*, special issue of ACM SIGPLAN Notices, 30(10), pages 300–315, 1995.

[Matile *et al.*, 1987] L. Matile, P. Tassy, et D. Goujet. Introduction à la systématique zoologique (concepts, principes, méthodes). *Biosystema*, 1, 1987. Société Française de Systématique, Paris.

[Matsuoka et Yonezawa, 1993] S. Matsuoka et A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, éditeurs

G. Agha, P. Wegner, et A. Yonezawa, pages 107–150. MIT Press, Cambridge (MA), USA, 1993.

[Maxwell, 1992] H.J. Maxwell. An expert system for harmonic analysis of tonal music. In *Understanding Music with AI: Perspectives on Music Cognition*, éditeurs M. Balaban, K. Ebcioglu, et O. Laske, pages 335–353. AAAI Press, Menlo Park (CA), USA, 1992.

[Mazouni *et al.*, 1995] K. Mazouni, B. Garbinato, et R. Guerraoui. Building reliable client-server software using actively replicated objects. In *Proceedings of TOOLS-Europe'95, Versailles, France*, éditeurs I. Graham, B. Magnusson, B. Meyer, et J.-M. Nerson, pages 37–53. Prentice-Hall, Hertfordshire, UK, 1995.

[McAffer, 1995] J. McAffer. Meta level programming with CodA. In *Proceedings of ECOOP'95, Aarhus, Denmark*, éditeur W. Olthoff, Lecture Notes in Computer Science 952, pages 190–214. Springer-Verlag, Berlin, 1995.

[McDermott, 1982] D. McDermott. A temporal logic for reasoning about processes and plans. *Cognitive Science*, 6:101–155, 1982.

[McHale, 1994] C. McHale. *Synchronization in Concurrent Object-oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, Department of Computer Science, Trinity College, Dublin, Ireland, 1994.

[McIlroy, 1968] D. McIlroy. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany*, pages 88–98, 1968.

[Mendelzon *et al.*, 1994] A.O. Mendelzon, T. Milo, et E. Waller. Object migration. In *Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS'94), Minneapolis (MN), USA*, pages 232–242. ACM Press, New York, 1994.

[Meseguer, 1993] J. Meseguer. Solving the inheritance anomaly in concurrent object-oriented programming. In *Proceedings of ECOOP'93, Kaiserslautern, Germany*, éditeur O. Nierstrasz, Lecture Notes in Computer Science 707, pages 220–246. Springer-Verlag, Berlin, 1993.

[Meyer, 1986] B. Meyer. Genericity versus inheritance. In *Proceedings of OOPSLA'86, Portland (OR), USA*, éditeur N. Meyrowitz, special issue of ACM SIGPLAN Notices, 21(11), pages 391–405, 1986.

[Meyer, 1988] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall International Series in Computer Science, C.A.R. Hoare Series Editor. Prentice Hall International, Hemel Hempstead, UK, 1988. Nouvelle édition révisée : [Meyer, 1997].

[Meyer, 1990] B. Meyer. *Conception et programmation par objets, pour du logiciel de qualité*. InterEditions, Paris, 1990. Traduction française de [Meyer, 1988].

[Meyer, 1991] B. Meyer. Programming as contracting. In *Advances in Object-Oriented Software Engineering*, éditeurs D. Mandrioli et B. Meyer, pages 1–15. Prentice-Hall, Englewood Cliffs (NJ), USA, 1991.

[Meyer, 1992a] B. Meyer. *Eiffel: The Language*. Prentice Hall Object-Oriented Series. Prentice Hall International, Hemel Hempstead, UK, 1992. Traduction française : [Meyer, 1994a].

[Meyer, 1992b] M. Meyer. Using hierarchical constraint satisfaction for lathe-tool selection in a CIM environment. Research Report 92-35, DFKI, Kaiserslautern Universität, Germany, 1992.

[Meyer, 1993a] éditeur B. Meyer. *Communications of the ACM, (36)3, special issue, Concurrent Object-Oriented Programming*, 1993.

[Meyer, 1993b] B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM (special issue, Concurrent Object-Oriented Programming, B. Meyer, editor)*, 36(9):56–80, 1993.

[Meyer, 1994a] B. Meyer. *Eiffel, le langage*. InterEditions, Paris, 1994. Traduction française de [Meyer, 1992a].

[Meyer, 1994b] B. Meyer. *Reusable Software: The Base Object-Oriented Libraries*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1994.

[Meyer, 1995a] B. Meyer. ISE Eiffel: The environment. Technical Report TR-EI-39/IE, Interactive Software Engineering, Goleta, Santa Barbara (CA), USA, 1995.

[Meyer, 1995b] B. Meyer. *Object Success: A Manager's Guide to Object Technology, its Impact on the Corporation, and its Use for Reengineering the Software Process*. The Object-Oriented series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1995.

[Meyer, 1997] B. Meyer. *Object-Oriented Software Construction, Second Edition*. The Object-Oriented Series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1997.

[Michel, 1997] C. Michel. *Modèles et implémentations d'interprètes réflexifs*. Thèse d'Informatique, Université de Nice Sophia Antipolis, 1997.

[Mingers, 1989] J. Mingers. An empirical comparison of selection measures for decision-tree induction. *Machine Learning*, 3(4):319–343, 1989.

[Minsky et Laske, 1992] M. Minsky et O. Laske. A conversation with Marvin Minsky (foreword). In *Understanding Music with AI: Perspectives on Music Cognition*, éditeurs M. Balaban, K. Ebcioglu, et O. Laske, pages 9–30. AAAI Press, Menlo Park (CA), USA, 1992.

[Minsky, 1975] M. Minsky. A framework for representing knowledge. In *The Psychology of Computer Vision*, éditeur P. Winston, pages 211–281. McGraw-Hill, New York (NY), USA, 1975.

[Monarchi et Puhr, 1992] D.E. Monarchi et G.I. Puhr. A research typology for object-oriented analysis. *Communications of the ACM*, 35(9):35–47, 1992.

[Monin, 1996] J.-F. Monin. *Comprendre les méthodes formelles. Panorama et outils logiques*. Collection technique et scientifique des télécommunications. Masson, Paris, 1996.

[Monk et Sommerville, 1993] S. Monk et I. Sommerville. Schema evolution in object-oriented databases systems using class versionning. *ACM SIGMOD Record*, 22(3):16–22, 1993.

[Montanari, 1974] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(3):95–132, 1974.

[Moore et Clement, 1996] I. Moore et T. Clement. A simple and efficient algorithm for inferring inheritance hierarchies. In *Proceedings of TOOLS-Europe'96, Paris, France*, éditeur R. Mitchell, pages 173–184. Prentice-Hall, Hertfordshire, UK, 1996.

[Moore, 1966] R. Moore. *Interval Arithmetic*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1966.

[Moore, 1996] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings of OOPSLA'96, San Jose (CA), USA*, special issue of ACM SIGPLAN Notices, 31(10), pages 235–250, 1996.

[Moreira et Clark, 1994] A.M.D. Moreira et R.G. Clark. Combining object-oriented analysis and formal description techniques. In *Proceedings of ECOOP'94, Bologna, Italy*, éditeurs M. Tokoro et R. Pareschi, Lecture Notes in Computer Science 821, pages 344–364. Springer-Verlag, Berlin, 1994.

[Morris *et al.*, 1996] D. Morris, G. Evans, P. Green, et C. Theaker. *Object-Oriented Computer Systems Engineering*. Applied Computing Series. Springer-Verlag, London, 1996.

[Moss, 1985] J. Moss. Log-based recovery for nested transactions. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB'87), Brighton, UK*, pages 427–432, 1985.

[Mouton et Pachet, 1995] R. Mouton et F. Pachet. Numeric vs symbolic controversy in automatic analysis of tonal music. In *Proceedings of the IJCAI'95 Workshop "Music and AI", Montréal, Canada*, pages 32–40, 1995.

[Mulet et Cointe, 1993] P. Mulet et P. Cointe. Definition of a reflective kernel for a prototype-based language. In *Proceedings of the 1st JSSST International Symposium on Object Technologies for Advanced Software, Kanazawa, Japan*, éditeurs S. Nishio et A. Yonezawa, Lecture Notes in Computer Science 742, pages 128–144. Springer-Verlag, Berlin, 1993.

[Mulet, 1995] Philippe Mulet. *Réflexion et langages à prototypes*. Thèse d'Informatique, Université de Nantes, 1995.

[Musser et Saini, 1996] D. R. Musser et A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Reading (MA), USA, 1996.

[Myers *et al.*, 1990] B.A. Myers, D.A. Giuse, R.B. Dannenberg, B. Van der Zanden, D. Kosbie, E. Previn, A. Mickish, et P. Marchal. Garnet: Comprehensive support for graphical highly-interactive user interfaces. *IEEE Computer*, 23(11):71–85, 1990.

[Myers *et al.*, 1992] B.A. Myers, D.A. Giuse, et B. Van der Zanden. Declarative programming in a prototype-instance system: Object-oriented programming without writing methods. In *Proceedings of OOPSLA'92, Vancouver, Canada*, éditeur A. Paepcke, special issue of ACM SIGPLAN Notices, 27(10), pages 184–200, 1992.

[Naja et Mouaddib, 1995] H. Naja et N. Mouaddib. Un modèle pour la représentation multiple dans les bases de données orientées-objet. In *Actes du Colloque*

*Langages et Modèles à Objets (LMO'95), Nancy, France*, éditeur A. Napoli, pages 173–189. INRIA Lorraine, Nancy, 1995.

[Napoli et Laurenço, 1993] A. Napoli et C. Laurenço. Représentations à objets et classification. Conception d'un système d'aide à la planification de synthèses organiques. *Revue d'Intelligence Artificielle*, 7(2):175–221, 1993.

[Napoli et Lieber, 1994] A. Napoli et J. Lieber. A first study on case-based planning in organic synthesis. In *Proceedings of the 1st European Workshop on Topics in Case-Based Reasoning, Kaiserslautern, Germany*, éditeurs S. Wess, K.D. Althoff, et M.M. Richter, Lecture Notes in Artificial Intelligence 837, pages 458–469. Springer-Verlag, Berlin, 1994.

[Napoli, 1992] A. Napoli. *Représentations à objets et raisonnement par classification en intelligence artificielle*. Thèse de Doctorat d'État en Informatique, Université Henri Poincaré Nancy 1, 1992.

[Napoli, 1997] A. Napoli. Une introduction aux logiques de descriptions. Rapport de Recherche RR-3314, INRIA, 1997.

[Nebel, 1990a] B. Nebel. *Reasoning and Revision in Hybrid Representation Systems*. Lecture Notes in Artificial Intelligence 422. Springer-Verlag, Berlin, 1990.

[Nebel, 1990b] B. Nebel. Terminological reasoning is inherently intractable. *Artificial Intelligence*, 43(2):235–249, 1990.

[Nebel, 1991] B. Nebel. Terminological cycles: Semantics and computational properties. In *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, éditeur J.F. Sowa, pages 331–361. Morgan Kaufmann Publishers, San Mateo (CA), USA, 1991.

[Nguyen et Rieu, 1989] G.T. Nguyen et D. Rieu. Schema evolution in object-oriented database systems. *Data and Knowledge Engineering*, 4:43–67, 1989.

[Nicol *et al.*, 1993] J. Nicol, T. Wilkes, et F. Manola. Object-orientation in heterogeneous distributed computing systems. *IEEE Computer*, 26(6):57–67, 1993.

[Nierstrasz et Tsichritzis, 1995] O. Nierstrasz et D. Tsichritzis. *Object-Oriented Software Composition*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1995.

[Nierstrasz, 1987] O.M. Nierstrasz. Active objects in Hybrid. In *Proceedings of OOPSLA'87, Orlando (FL), USA*, special issue of ACM SIGPLAN Notices, 22(12), pages 243–253, 1987.

[Object Management Group, 1996] Object Management Group. CORBA 2.0/IIOP specification. Technical Report PTC/96-03-04, Framingham Corporate Center, Framingham (MA), USA, 1996.

[ODP, 1995] Union Internationale des Télécommunications, Genève. *ODP, introduction. Recommandation IUT-T X.901*, 1995.

[Okamura et Ishikawa, 1994] H. Okamura et Y. Ishikawa. Object location control using meta-level programming. In *Proceedings of ECOOP'94, Bologna, Italy*, éditeurs M. Tokoro et R. Pareschi, Lecture Notes in Computer Science 821, pages 299–319. Springer-Verlag, Berlin, 1994.

[Omohundro et Stoutamire, 1995] S.M. Omohundro et D. Stoutamire. The Sather 1.0 specification. Technical Report TR-95-057, International Computer Science Institute, Berkeley (CA), USA, 1995.

[Ott et Noordik, 1992] M.A. Ott et J.H. Noordik. Computer tools for reaction retrieval and synthesis planning in organic chemistry. A brief review of their history, methods, and programs. *Recueil des Travaux Chimiques des Pays-Bas*, 111:239–246, 1992.

[Otte *et al.*, 1996] R. Otte, P. Patrick, et M. Roy. *Understanding CORBA*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1996.

[Ouaggag et Godin, 1997] H. Ouaggag et R. Godin. Étude empirique de l'influence de l'héritage multiple sur l'entropie conceptuelle : comparaison avec l'héritage simple. In *Actes du Colloque Langages et Modèles à Objets (LMO'97), Roscoff, France*, éditeurs R. Ducournau et S. Garlatti, pages 161–172. Hermès, Paris, 1997.

[Oussalah, 1997] éditeur C. Oussalah. *Ingénierie objet : concepts et techniques*. InterEditions, Paris, 1997.

[Ovans et Davison, 1992] R. Ovans et R. Davison. An interactive constraint-based expert assistant for music composition. In *Proceedings of the 9th Canadian Conference on Artificial Intelligence, University of British Columbia, Vancouver, Canada*, pages 76–81, 1992.

[Pachet *et al.*, 1996] F. Pachet, G. Ramalho, et J. Carrive. Representing temporal musical objects and reasoning in the MusES system. *Journal of New Music Research*, 5(3):252–275, 1996.

[Pachet et Carrive, 1996] F. Pachet et J. Carrive. Propriétés des intervalles temporels circulaires et application à l'analyse harmonique automatique. In *Actes des 3es Journées d'Informatique Musicale, Ile de Tatihou, France*, pages 230–247, 1996.

[Pachet et Dojat, 1995] F. Pachet et M. Dojat. Un framework pour la représentation de connaissances temporelles en NéOpus. Rapport Technique 95-19, LAFORIA-IBP, Paris, 1995.

[Pachet et Roy, 1995a] F. Pachet et P. Roy. Integrating constraint satisfaction techniques with complex object structures. In *Proceedings of the 15th Annual Conference of the British Computer Society Specialist Group on Expert Systems, Cambridge, UK*, pages 11–22, 1995.

[Pachet et Roy, 1995b] F. Pachet et P. Roy. Mixing constraints and objects: A case study in automatic harmonization. In *Proceedings of TOOLS-Europe'95, Versailles, France*, éditeurs I. Graham, B. Magnusson, et J.-M. Nerson, pages 119–126. Prentice-Hall, Hertfordshire, UK, 1995.

[Pachet, 1994a] F. Pachet. The MusES system: An environment for experimenting with knowledge representation techniques in tonal harmony. In *Proceedings of the 1st Brazilian Symposium on Computer Music, Caxambu, Minas Gerais, Brazil*, pages 195–201, 1994.

[Pachet, 1994b] F. Pachet. An object-oriented representation of pitch-classes, intervals, scales and chords. In *Actes des 1res Journées d'Informatique Musicale, Bordeaux*, pages 19–34, 1994.

[Pachet, 1994c] F. Pachet. Vers un modèle du raisonnement dans les langages à objets. In *Actes du Colloque Langages et Modèles à Objets (LMO'94), Grenoble,* *France*, éditeur F. Rechenmann, pages 111–123. INRIA Rhônes-Alpes – IMAG-LIFIA, Grenoble, 1994.

[Pachet, 1998] F. Pachet. Computer analysis of jazz chord sequences: Is Solar a blues? À paraître dans *Readings in Music and Artificial Intelligence*, Harwood Academic Publishers, Basel, Switzerland, 1998.

[Padgham et Lambrix, 1994] L. Padgham et P. Lambrix. A framework for part-of hierarchies in terminological logics. In *Proceedings of KR'94, Bonn, Germany*, pages 485–496, 1994.

[Padgham et Nebel, 1993] L. Padgham et B. Nebel. Combining classification and nonmonotonic inheritance reasoning: A first step. In *Methodologies for Intelligent Systems*, éditeurs J. Komorowski et Z.W. Raś, Lecture Notes in Computer Science 689, pages 132–141. Springer-Verlag, Berlin, 1993.

[Padgham et Zhang, 1993] L. Padgham et T. Zhang. A terminological logic with defaults: A definition and an application. In *Proceedings of IJCAI'93, Chambéry, France*, pages 662–668, 1993.

[Pankhurst et Aitchinson, 1975] R.J. Pankhurst et R.R. Aitchinson. An on-line identification. In *Biological Identification with Computers*, éditeur R.J. Pankhurst, pages 181–196. Academic Press, London, 1975.

[Pankhurst, 1991] R.J. Pankhurst. *Practical Taxonomic Computing*. Cambridge University Press, Cambridge, UK, 1991.

[Papathomas, 1989] M. Papathomas. Concurrency issues in object-oriented langages. In *Object Oriented Development*, éditeur D. Tsichritzis, pages 207–245. University of Geneva, 1989.

[Paramasivam et Plaisted, 1996] M. Paramasivam et D.A. Plaisted. Automated deduction techniques for classification in concept languages. Technical Report TR96-015, Computer Science Department, University of North Carolina, Chapel Hill (NC), USA, 1996. Available by anonymous ftp at ftp.cs.unc.edu in pub/technical-reports.

[Parent *et al.*, 1989] C. Parent, H. Rolin, K. Yetongnon, et S. Spaccapietra. An ER calculus for the entity-relationship complex model. In *Proceedings of the 8th International Conference on Entity-Relationship Approach (ER'89), Toronto, Canada*, éditeur F.H. Lochovsky, Entity-Relationship Approach to Database Design and Querying, pages 361–384. North-Holland, Amsterdam, 1989.

[Parrington et Shrivastava, 1988] G.D. Parrington et S.K. Shrivastava. Implementing concurrency control in reliable distributed object-oriented systems. In *Proceedings of ECOOP'88, Oslo, Norway*, éditeurs S. Gjessing et K. Nygaard, Lecture Notes in Computer Science 322, pages 234–249. Springer-Verlag, Berlin, 1988.

[Patel-Schneider *et al.*, 1984] P.F. Patel-Schneider, R.J. Brachman, et H.J. Levesque. ARGON: Knowledge representation meets information retrieval. In *Proceedings of the 1st IEEE Conference on Artificial Intelligence Applications, Denver (CO), USA*, pages 280–286, 1984.

[Patel-Schneider et Swartout, 1993] P.F. Patel-Schneider et B. Swartout. Description logic specification from the KRSS effort. Working version (draft), June 1993.

[Patel-Schneider, 1989] P.F. Patel-Schneider. Undecidability of subsumption in NIKL. *Artificial Intelligence*, 39(2):263–272, 1989.

[Pavé *et al.*, 1991] A. Pavé, N. Gautier, et C. Bernstein. Object-centered representation and problems related to living systems in nature: Systematics, biogeography and population dynamics. In *Artificial Intelligence in Numerical and Symbolic Simulation*, éditeurs A. Pavé et G.C. Vansteenkiste, pages 51–73. ALEAS, Lyon, 1991.

[Pelenc et Ducournau, 1997] S. Pelenc et R. Ducournau. Contraintes et objets. Rapport Technique 97033, LIRMM, Montpellier, 1997.

[Pelenc, 1994] S. Pelenc. Application de techniques de propagation de contraintes en langage à objets. Mémoire de DEA, Université des Sciences et Techniques du Languedoc, Montpellier, 1994. Également Rapport Technique France Télécom/CNET RP/PAA/ATR/GTR/4153.

[Peltason, 1991] C. Peltason. The BACK system—An overview. *ACM SIGART Bulletin*, 2(3):114–119, 1991.

[Penney et Stein, 1987] P.J. Penney et J. Stein. Class modification in the Gemstone object-oriented DBMS. In *Proceedings of OOPSLA'87, Orlando (FL), USA*, special issue of ACM SIGPLAN Notices, 22(12), pages 111–117, 1987.

[Perec, 1980] G. Perec. Experimental demonstration of the tomatotopic organization in the Soprano (*Cantatrix sopranica L.*). *Banana Split*, 2, 1980. Seconde publication : [Perec, 1991].

[Perec, 1991] G. Perec. *Cantatrix sopranica L. et autres écrits scientifiques*. La Librairie du XX$^{e}$ siècle. Le Seuil, Paris, 1991.

[Perrot, 1995] J.-F. Perrot. *Langages à objets*. Collection Informatique, fascicule H 2 510. Techniques de l'Ingénieur, Paris, 1995.

[Picard, 1972] C.-F. Picard. *Graphes et questionnaires (tome 2)*. Editions Gauthier-Villars, Paris, 1972.

[Pistor et Andersen, 1986] P. Pistor et F. Andersen. Designing a generalized NF2 model with an SQL-type language interface. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB'86), Kyoto, Japan*, pages 278–285. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1986.

[Pivot et Prokop, 1987] B. Pivot et M. Prokop. Définition et réalisation d'une fonctionnalité de classification dans SHIRKA. Mémoire d'année spéciale « Intelligence Artificielle », ENSIMAG, Grenoble, 1987.

[Pope, 1989] éditeur S. Pope. *Computer Music Journal, (13)2, special issue, Object-Oriented Programming*, 1989.

[Pope, 1991a] S. Pope. Introduction to MODE: The Musical Object Development Environment. In *The Well-Tempered Object: Musical Applications of Object-Oriented Programming*, éditeur S. Pope, pages 83–106. MIT Press, Cambridge (MA), USA, 1991.

[Pope, 1991b] éditeur S. Pope. *The Well-Tempered Object: Musical Applications of Object-Oriented Programming*. MIT Press, Cambridge (MA), USA, 1991.

[Projet Sherpa, 1995] Projet Sherpa. TROPES 1.0: Reference manual. Rapport interne, INRIA Rhône-Alpes, Grenoble, 1995. Version révisée : Tropes 1.1 Reference Manual, 1997.
ftp://ftp.inrialpes.fr/pub/sherpa/rapports/tropes-manual.ps.gz.

[Pronk et Tercero, 1994] C. Pronk et C. Tercero. Methods and techniques for object-oriented analysis and design. Technical Report 94-107, Delft University of Technology, The Netherlands, 1994.

[Puig et Oussalah, 1996] V. Puig et C. Oussalah. Constraints and composite objects. In *Proceedings of the 7th International Conference and Workshop on Data Expert Systems Application (DEXA'96), Zurich, Switzerland*, Lecture Notes in Computer Science 1134, pages 521–530, 1996.

[Quantz et Royer, 1992] J. Quantz et V. Royer. A preference for defaults in terminological logics. In *Proceedings of KR'92, Cambridge (MA), USA*, pages 294–305, 1992.

[Quesneville, 1996] H. Quesneville. *Dynamique des populations d'éléments transposables : modélisation et validation*. Thèse, Université Pierre et Marie Curie (Paris VI), 1996.

[Quillian, 1968] M.R. Quillian. Semantic memory. In *Semantic Information Processing*, éditeur M. Minsky, pages 227–270. MIT Press, Cambridge (MA), USA, 1968.

[Quinlan, 1986] J.R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[Ra et Rundensteiner, 1997] Y.-G. Ra et E.A. Rundensteiner. A transparent schema-evolution system based on object-oriented view technology. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):600–624, 1997.

[Ramalho et Ganascia, 1994] G. Ramalho et J.G. Ganascia. Simulating creativity in jazz performance. In *Proceedings of AAAI'94, Seattle (WA), USA*. AAAI Press, Menlo Park (CA), USA, 1994.

[Ramalho et Pachet, 1994] G. Ramalho et F. Pachet. From real book to real jazz performance. In *Proceedings of the International Conference on Music Perception and Cognition, Lièges, Belgium*, pages 349–350, 1994.

[Rathke et Redmiles, 1993] C. Rathke et D. Redmiles. Multiple representation perspectives for supporting explanation in context. Research Report CU-CS-645, University of Colorado, Boulder (CO), USA, 1993.

[Rathke, 1993] C. Rathke. Object-oriented programming and frame-based knowledge representation. In *Proceedings of the 5th IEEE International Conference on Tools with Artificial Intelligence, Boston (MA), USA*, pages 95–98, 1993.

[Rechenmann, 1985] F. Rechenmann. SHIRKA : mécanismes d'inférences sur une base de connaissances centrée objets. In *Actes du 5e Congrès AFCET Reconnaissance des Formes et Intelligence Artificielle, Grenoble*, pages 1243–1254, 1985.

[Rechenmann, 1988] F. Rechenmann. *SHIRKA : système de gestion de bases de connaissances centrées-objet. Manuel de référence*. INRIA/ARTEMIS, Grenoble, 1988. ftp://ftp.inrialpes.fr/pub/sherpa/rapports/manuel-shirka.ps.gz.

[Rechenmann, 1993] F. Rechenmann. Integrating procedural and declarative knowledge in object-based knowledge models. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Le Touquet, France*, pages 98–101, 1993.
ftp://ftp.inrialpes.fr/pub/sherpa/publications/rechenmann93.ps.gz.

[Régin, 1995] J.-C. Régin. *Développement d'outils algorithmiques pour l'intelligence artificielle. Application à la chimie organique*. Thèse, Université des Sciences et Techniques du Languedoc, Montpellier, 1995.

[Rémy et Vouillon, 1997] D. Rémy et J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages (POPL'97), Paris, France*, pages 40–53, 1997.

[Resnick *et al.*, 1995] L.A. Resnick, A. Borgida, R.J. Brachman, D.L. McGuinness, P.F. Patel-Schneider, et K.C. Zalondek. *CLASSIC. Description and Reference Manual for the Common Lisp Implementation (Version 2.3)*. AT&T Bell Laboratories, Murray Hill (NJ), USA, 1995.

[Revault, 1996] N. Revault. *Principes de méta-modélisation pour l'utilisation de canevas d'applications à objets (MetaGen et les frameworks)*. Thèse d'Informatique, Université Pierre et Marie Curie (Paris VI), LAFORIA, 1996.

[Rich, 1991] éditeur C. Rich. *SIGART Bulletin, (2)3, special issue, Implemented Knowledge Representation and Reasoning Systems*, 1991.

[Ride et Younes, 1986] W.D.L. Ride et T. Younes. *Biological Nomenclature Today*. IRL Press (for International Council of Scientific Unions Press), Oxford, UK, 1986.

[Rist et Terwilliger, 1995] R. Rist et R. Terwilliger. *Object-Oriented Programming in Eiffel*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1995.

[Rivard, 1997] F. Rivard. *Évolution du comportement dans les langages à classes réflexifs dynamiquement typés*. Thèse d'Informatique, École des Mines de Nantes, 1997.

[Robert et Verjus, 1977] P. Robert et J.-P. Verjus. Towards autonomous descriptions of synchronization modules. In *Proceedings of the International Federation of Information Processing Congress, Toronto, Canada*, éditeur B. Gilchrist, pages 981–986. North-Holland, New York (NY), USA, 1977.

[Roberts et Goldstein, 1977] R.B. Roberts et I.P. Goldstein. The FRL manual. AI Memo 409, Artificial Intelligence Laboratory, MIT, Cambridge (MA), USA, 1977.

[Rodet et Cointe, 1991] X. Rodet et P. Cointe. Formes: Composition and scheduling of process. In *The Well-Tempered Object: Musical Applications of Object-Oriented Programming*, éditeur S. Pope, pages 64–82. MIT Press, Cambridge (MA), USA, 1991.

[Rolland et Ganascia, 1996] P.-Y. Rolland et J.G. Ganascia. Automated motive-oriented analysis of musical corpuses: A jazz case study. In *Proceedings of the 20th International Computer Music Conference, Hong-Kong*, pages 240–243, 1996.

[Rolland et Pachet, 1996] P.-Y. Rolland et F. Pachet. A framework for representing knowledge about synthesizer programming. *Computer Music Journal*, 20(3):47–58, 1996.

[Rossazza, 1990] J.-P. Rossazza. *Utilisation de hiérarchies de classes floues pour la représentation de connaissances imprécises et sujettes à exceptions : le système SORCIER*. Thèse d'Informatique, Université Paul Sabatier, Toulouse, 1990.

[Roudier, 1996] Y. Roudier. *Abstractions réactives pour les langages à objets parallèles : modèles et programmation*. Thèse d'Informatique, Université de Nice Sophia Antipolis, 1996.

[Rousseau *et al.*, 1994a] R. Rousseau, P. Brissi, et P. Collet. From Eiffel to a general object model: Our experience and suggestions. In *Proceedings of OOPSLA'94, Portland (OR), USA, Workshop on "The Object Engine: Foundation for Next Generation Architectures"*, éditeur F.A. Cummins, special issue of ACM SIGPLAN Notices, 29(10), pages 29–37, 1994.

[Rousseau *et al.*, 1994b] R. Rousseau, P. Lahire, et P. Brissi. Reuse of software components requires quality and productivity: Our experience with Eiffel. Technical Report RR 94-64, Laboratoire I3S, Sophia Antipolis, 1994.

[Rousseau, 1994] R. Rousseau. Fallait-il concevoir Ada 9X ? *Technique et Science Informatiques*, 13(5):723–729, 1994.

[Rousseau, 1995] R. Rousseau. Vers un modèle à objets commun pour le génie logiciel, les bases de données et la CAO. In *Actes des Journées du GDR Programmation du CNRS, Grenoble, France*, pages 99–109. GDR Programmation/Université de Bordeaux I, 1995.

[Rozier, 1992] M. Rozier. Chorus (overview of the Chorus distributed operating system). In *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures, Seattle (WA), USA*, pages 39–70, 1992.

[Ruiz-Delgado *et al.*, 1995] A. Ruiz-Delgado, D. Pitt, et C. Smythe. A review of object-oriented approaches in formal methods. *The Computer Journal*, 38(10):777–784, 1995.

[Rumbaugh *et al.*, 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, et W. Lorensen. *Object Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1991.

[Rundensteiner, 1992] E.A. Rundensteiner. Multiview: A methodology for supporting multiple views in object-oriented databases. In *Proceedings of the 18th International Conference on Very Large Data Bases (VLDB'92), Vancouver, British Columbia, Canada*, pages 187–198. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1992.

[Rundensteiner, 1994] E.A. Rundensteiner. A classification algorithm for supporting object-oriented views. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94), Gaithersburg (MD), USA*, pages 18–25, 1994.

[Scaletti, 1987] C. Scaletti. Kyma: An object-oriented language for music composition. In *Proceedings of the 11th International Computer Music Conference, San Francisco (CA), USA*, pages 49–56, 1987.

[Schaeffer, 1966] P. Schaeffer. *Traité des objets musicaux*. Le Seuil, Paris, 1966.

[Schaerf, 1994] A. Schaerf. *Query Answering in Concept-Based Knowledge Representation Systems: Algorithms, Complexity, and Semantic Issues*. Dottorato di ricerca in informatica, Università degli Studi di Roma "La Sapienza", Italia, 1994.

[Schafer, 1976] G. Schafer. *A Mathematical Theory of Evidence*. Princeton University Press, Princeton (NJ), USA, 1976.

[Schmidt-Schauß et Smolka, 1991] M. Schmidt-Schauß et G. Smolka. Attributive concept descriptions with complements. *Artificial Intelligence*, 48(1):1–26, 1991.

[Schmiedel, 1990] A. Schmiedel. A temporal terminological logic. In *Proceedings of AAAI'90, Boston (MA), USA*, pages 640–645, 1990.

[Schmolze et Lipkis, 1983] J.G. Schmolze et T.A. Lipkis. Classification in the KL-ONE knowledge representation system. In *Proceedings of IJCAI'83, Karlsruhe, West Germany*, pages 330–332, 1983.

[Schmolze et Mark, 1991] J.G. Schmolze et W.S. Mark. The NIKL experience. *Computational Intelligence*, 6:48–69, 1991.

[Schoman et Ross, 1977] K. Schoman et D.T. Ross. Structured analysis for requirements definition. *IEEE Transactions on Software Engineering*, SE-3(1):6–15, 1977.

[Schreiber *et al.*, 1993] G. Schreiber, B. Wielinga, et J. Breuker. *KADS: A Principled Approach to Knowledge-based System Development*. Academic Press, New York (NY), USA, 1993.

[Shlaer et Mellor, 1988] S. Shlaer et S.J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1988.

[Shlaer et Mellor, 1992] S. Shlaer et S.J. Mellor. *Object Oriented Life Cycles: Modeling the World in States*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1992.

[Siboni, 1994] D. Siboni. OMT-KADS : l'illustration d'une cohabitation sur des applications industrielles. In *Actes des Journées Méthodes Objets et Intelligence Artificielle — Frontières, Ponts et Synergies*, pages 55–61. EC2, Nanterre, 1994.

[Sieffer, 1988] A. Sieffer. Constitution d'une base de connaissances centrée-objet en vue de l'identification des poissons des Kerguelen à l'aide du système SHIRKA. Mémoire de DEA, Université Claude Bernard, Laboratoire de Biométrie, Lyon, 1988.

[Simonet, 1994] G. Simonet. *Héritage non monotone à base de chemins et de graphes partiels*. Thèse d'Informatique, Université des Sciences et Techniques du Languedoc, Montpellier, 1994.

[Simons et Spector, 1987] éditeurs B. Simons et A. Spector. *Fault-Tolerant Distributed Computing*. Lecture Notes in Computer Science 448. Springer-Verlag, Berlin, 1987.

[Skarra et Zdonik, 1986] A.H. Skarra et S.B. Zdonik. The management of changing types in an object-oriented databases. In *Proceedings of OOPSLA'86, Portland*

*(OR), USA*, special issue of ACM SIGPLAN Notices, 21(11), pages 483–495, 1986.

[Smith et Medin, 1981] E.E. Smith et D. Medin. *Categories and Concepts*. Harvard University Press, Cambridge (MA), USA, 1981.

[Smith et Ungar, 1995] R.B. Smith et D. Ungar. Programming as an experience: The inspiration for SELF. In *Proceedings of ECOOP'95, Aarhus, Denmark*, éditeur W. Olthoff, Lecture Notes in Computer Science 952, pages 303–330. Springer-Verlag, Berlin, 1995.

[Smith, 1994] W.R. Smith. The Newton application architecture. In *Proceedings of the 39th IEEE Computer Society International Conference, San Francisco (CA), USA*, pages 156–161, 1994.

[Smith, 1995] W.R. Smith. Using a prototype-based language for user interface: The Newton Project's experience. In *Proceedings of OOPSLA'95, Austin (TX), USA*, special issue of ACM SIGPLAN Notices, 30(10), pages 61–72, 1995.

[Smolka *et al.*, 1993] G. Smolka, M. Hentz, et J. Würtz. Object-oriented constraint programming in Oz. Research Report RR-93-16, DFKI, Saarbrücken Universität, Germany, 1993.

[Snodgrass et Ahn, 1986] R.T. Snodgrass et I. Ahn. Temporal databases. *IEEE Computer Science Technical Committee on Database Engineering*, 19(9):35–42, 1986.

[Sokal et Sneath, 1963] R.R. Sokal et P.H.A. Sneath. *Principles of Numerical Taxonomy*. Freeman, San Francisco (CA), USA, 1963.

[Sommerville, 1985] I. Sommerville. *Software Engineering, Second Edition*. Addison-Wesley, Reading (MA), USA, 1985.

[Souza dos Santos, 1995] C. Souza dos Santos. *Un mécanisme de vues pour les systèmes de gestion de bases de données objet*. Thèse d'Informatique, Université de Paris-Sud, Centre d'Orsay, 1995.

[Sowa, 1984] J.F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, Reading (MA), USA, 1984.

[Steedman, 1984] M.J. Steedman. A generative grammar for jazz chord sequences. *Music Perception*, 2(1):52–77, 1984.

[Steele Jr., 1990] G.L. Steele Jr. *Common Lisp: The Language, Second Edition*. Digital Press, Bedford (MA), USA, 1990.

[Stefik *et al.*, 1986] M.J. Stefik, D.G. Bobrow, et K.M. Kahn. Integrating access-oriented programming into a multi-paradigm environment. *IEEE Software*, 3(1):10–18, 1986.

[Stefik et Bobrow, 1986] M. Stefik et D.G. Bobrow. Object-oriented programming: Themes and variations. *The AI Magazine*, 6(4):40–62, 1986.

[Stein *et al.*, 1989] L.A. Stein, H. Lieberman, et D. Ungar. A shared view of sharing: The Treaty of Orlando. In *Object-Oriented Concepts, Databases and Applications*, éditeurs W. Kim et F.H. Lochovsky, pages 31–48. ACM Press/Addison-Wesley, Reading (MA), USA, 1989.

[Stein, 1987] L.A. Stein. Delegation IS inheritance. In *Proceedings of OOPSLA'87, Orlando (FL), USA*, éditeur N.K. Meyrowitz, special issue of ACM SIGPLAN Notices, 22(12), pages 138–146, 1987.

[Steyaert, 1994] P. Steyaert. *Open Design of Object-Oriented Languages. A Foundation for Specialisable Reflective Language Frameworks*. PhD thesis, Vrije Universiteit Brussel, Belgium, 1994.

[Stonebraker, 1987] M. Stonebraker. The design of Postgres system. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB'87), Brighton, UK*, pages 111–117. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1987.

[Straw *et al.*, 1989] A. Straw, F. Mellender, et S. Riegel. Object management in a persistent Smalltalk system. *Software – Practice and Experience*, 19(8):719–737, 1989.

[Stroustrup, 1986] B. Stroustrup. *The C++ Programming Language*. Addison--Wesley, Reading (MA), USA, 1986.

[Stroustrup, 1992] B. Stroustrup. *Le langage C++, 2^e^ édition*. Addison-Wesley, Paris, 1992.

[Stroustrup, 1994] B. Stroustrup. *The Design and Evolution of C++*. Addison--Wesley, Reading (MA), USA, 1994.

[Sussman et Jr., 1980] G.J. Sussman et G.L. Steele Jr. CONSTRAINTS— A language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, 1980.

[Sutherland, 1963] I. Sutherland. *SKETCHPAD: A Man-Machine Graphical Communication System*. PhD thesis, MIT, Reading (MA), USA, Cambridge (MA), USA, 1963.

[Switzer, 1993] R. Switzer. *Eiffel, an Introduction*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1993. Traduction française : [Switzer, 1995].

[Switzer, 1995] R. Switzer. *Introduction à Eiffel*. Masson, Paris, 1995. Traduction française de [Switzer, 1993].

[Taivalsaari, 1991] A. Taivalsaari. Cloning is inheritance. Computer Science Report WP-18, University of Jyväskylä, Finland, 1991.

[Taivalsaari, 1993] A. Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming*. PhD thesis, University of Jyväskylä, Finland, 1993.

[Tan et Katayama, 1989] L. Tan et T. Katayama. Meta operations for type management in object-oriented databases. In *Proceedings of the 1st Conference on Deductive and Object-Oriented Databases (DOOD'89), Kyoto, Japan*, pages 58–75, 1989.

[Tanaka *et al.*, 1988] K. Tanaka, M. Yoshikawa, et I. Koso. Schema virtualization in object-oriented databases. In *Proceedings of the 4th IEEE International Conference on Data Engineering (ICDE'88), Los Angeles (CA), USA*, pages 31–38, 1988.

[Tansel *et al.*, 1993] A.U. Tansel, J. Clifford, S. Gadia, S. Jajopia, A. Segev, et D.R. Snodgrass. *Temporal Databases*. Series on Database Systems and Applications. Benjamin/Cummings Publishing, Redwood City (CA), USA, 1993.

[Tassy, 1986] P. Tassy. Construction systématique et soumission au test : une forme de connaissance objective. In *L'ordre et la diversité du vivant*, éditeur P. Tassy, pages 83–98. Fondation Diderot, Fayard, Paris, 1986.

[Thomas et Weedon, 1995] P. Thomas et R. Weedon. *Object-Oriented Programming in Eiffel*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1995.

[Thomas, 1992] L. Thomas. Extensibility and reuse of object-oriented synchronization components. In *Proceedings of the 4th International Parallel Architectures and Languages Europe Conference (PARLE'92), Paris, France*, éditeurs D. Etiemble et J.-C. Syre, Lecture Notes in Computer Science 605, pages 261–278. Springer-Verlag, Berlin, 1992.

[Tomlinson et Singh, 1989] C. Tomlinson et V. Singh. Inheritance and synchronization with enabled-sets. In *Proceedings of OOPSLA'89, New Orleans (LA), USA*, special issue of ACM SIGPLAN Notices, 24(10), pages 103–112, 1989.

[Touretzky, 1986] D.S. Touretzky. *The Mathematics of Inheritance*. Morgan Kaufmann Publishers, Los Altos (CA), USA, 1986.

[Tresch et Scholl, 1993] M. Tresch et M.H. Scholl. Schema transformation without database reorganization. *ACM SIGMOD Record*, 22(1):21–27, 1993.

[Trombettoni, 1992] G. Trombettoni. Conception d'un algorithme de maintien de solutions dans un réseau de contraintes. Rapport de Recherche 1784, INRIA Sophia Antipolis, 1992.

[Trombini, 1987] C. Trombini. Planning organic synthesis I–IV. *Da la Chimica e l'Industria*, 69:(5):82–86, (6):83–86, (9):99–104, (12):129–133, 1987.

[Tsang et Aitken, 1991] C.P. Tsang et M. Aitken. Harmonizing music as a discipline of constraint logic programming. In *Proceedings of the 15th International Computer Music Conference, Montréal, Canada*, pages 61–64, 1991.

[Tsang, 1987] E.P. Tsang. Times structures for artificial intelligence. In *Proceedings of IJCAI'87, Milano, Italy*, pages 456–461, 1987.

[Tsang, 1993] E. Tsang. *Foundations of Constraint Satisfaction*. Computation in Cognitive Science. Academic Press, New York (NY), USA, 1993.

[Tsichritzis et Klug, 1978] D. Tsichritzis et A.C. Klug. The ANSI/X3/SPARC DBMS framework report of the study group on database management systems. *Information Systems*, 3(3):173–191, 1978.

[Ullman, 1989] J.D. Ullman. *Principles of Database Systems and Knowledge--Based Systems, volumes 1–2*. Computer Science Press, Rockville (MD), USA, 1989.

[Ulrich, 1977] W. Ulrich. The analysis and synthesis of jazz by computer. In *Proceedings of IJCAI'77, Cambridge (MA), USA*, pages 865–872, 1977.

[Ungar *et al.*, 1991] D. Ungar, C. Chambers, B.W. Chang, et U. Hølzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3):223–242, 1991.

[Ungar et Smith, 1987] D. Ungar et R.B. Smith. SELF: The power of simplicity. In *Proceedings of OOPSLA'87, Orlando (FL), USA*, éditeur N.K. Meyrowitz, special issue of ACM SIGPLAN Notices, 22(12), pages 227–242, 1987. Également publié dans *Lisp and Symbolic Computation*, 4(3), Kluwer Academic Publishers, pages 187–205, 1991.

[Valtchev et Euzenat, 1996] P. Valtchev et J. Euzenat. Classification of concepts through products of concepts and abstract data types. In *Ordinal and Symbolic Data Analysis*, éditeurs E. Diday, Y. Lechevallier, et O. Opitz, Studies in Classification, Data Analysis, and Knowledge Organization, pages 3–12. Springer-Verlag, Berlin, 1996.
ftp://ftp.inrialpes.fr/pub/sherpa/publications/valtchev96a.ps.gz.

[van den Bos et Laffra, 1991] J. van den Bos et C. Laffra. Procol: A concurrent object-language with protocols, delegation and persistence. *Acta Informatica*, 28(6):511–538, 1991.

[Van Hentenryck, 1989] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge (MA), USA, 1989.

[Vanwormhoudt *et al.*, 1997] G. Vanwormhoudt, B. Carré, et L. Debrauwer. Programmation par objets et contextes fonctionnels. Application de Crome à Smalltalk. In *Actes du Colloque Langages et Modèles à Objets (LMO'97), Roscoff, France*, éditeurs R. Ducournau et S. Garlatti, pages 223–239. Hermès, Paris, 1997.

[Vanwormhoudt, 1998] G. Vanwormhoudt. Programmation par contextes en Smalltalk. *L'Objet, numéro spécial Smalltalk*, 3(4):429–444, 1998.

[Varma, 1987] T.R. Varma. Clinical algorithms: Infertility. *British Medical Journal*, 294:887–890, 1987.

[Verfaillie et Schiex, 1994] G. Verfaillie et T. Schiex. Dynamic backtracking for dynamic constraint satisfaction problems. In *Proceedings of the ECAI'94 Workshop on Constraint Satisfaction Issues Raised by Practical Applications, Amsterdam, The Netherlands*, pages 1–8, 1994.

[Vignes, 1991] R. Vignes. *Caractérisation automatique de groupes biologiques*. Thèse, Université Pierre et Marie Curie (Paris VI), 1991.

[Vignes, 1996] R. Vignes. Construction de clés d'identification et stratégies d'identification. *Biosystema*, 14:91–108, 1996. Société Française de Systématique, Paris.

[Vismara *et al.*, 1992] P. Vismara, J.-C. Régin, J. Quinqueton, M. Py, C. Laurenço, et L. Lapied. RESYN — Un système d'aide à la conception de plans de synthèse en chimie organique. In *Actes des 12$^{es}$ Journeés Internationales Intelligence Artificielle, Systèmes Experts et Langage Naturel, Avignon*, pages 305–318, 1992.

[Vismara, 1995] P. Vismara. *Reconnaissance et représentation d'éléments structuraux pour la description d'objets complexes. Application à l'élaboration de stratégies de synthèse en chimie organique*. Thèse, Université des Sciences et Techniques du Languedoc, Montpellier, 1995.

[Vismara, 1996] P. Vismara. Appariements dirigés pour le raisonnement par classification sur des hiérarchies de graphes. In *Actes du Colloque Langages et*

*Modèles à Objets (LMO'96), Leysin, Suisse*, éditeur Y. Dennebouy, pages 201–214. École Polytechnique Fédérale de Lausanne, 1996.

[Vlissides et Linton, 1990] J.M. Vlissides et M.A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, 1990.

[Vogel, 1988] C. Vogel. *Génie cognitif.* Masson, Paris, 1988.

[Waldén et Nerson, 1995] K. Waldén et J.-M. Nerson. *Seamless Object-Oriented Software Architecture — Analysis and Design of Reliable Systems*. The Object-Oriented series. Prentice-Hall, Englewood Cliffs (NJ), USA, 1995.

[Walker *et al.*, 1992] W. Walker, K. Hebel, S. Martirano, et C. Scaletti. ImprovisationBuilder: Improvisation as conversation. In *Proceedings of the 16th International Computer Music Conference, San Jose (CA), USA*, pages 190–193, 1992.

[Waller, 1991] E. Waller. Schema updates and consistency. In *Proceedings of the 2nd Conference on Deductive and Object-Oriented Databases (DOOD'91), Munich, Germany*, pages 167–188, 1991.

[Ward, 1986] P. Ward. The transformational schema: An extension of the data flow diagram to represent control and timing. *IEEE Transactions on Software Engineering*, SE-12(2):198–210, 1986.

[Wegner, 1987] P. Wegner. Dimensions of object-based language design. In *Proceedings of OOPSLA'87, Orlando (FL), USA*, special issue of ACM SIGPLAN Notices, 22(12), pages 168–182, 1987.

[Weihl, 1989] W. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–283, 1989.

[Wiener, 1995] R. Wiener. *Software Development Using Eiffel: There Can Be Life After C++*. Prentice Hall Object-Oriented Series. Prentice Hall International, Hemel Hempstead, UK, 1995.

[Wiener, 1996] R. Wiener. *An Object-Oriented Introduction to Computer Science Using Eiffel*. Prentice Hall Object-Oriented Series. Prentice Hall International, Hemel Hempstead, UK, 1996.

[Wiley, 1981] E.O. Wiley. *Phylogenetics: The Theory and Practice of Phylogenetic Systematics*. John Wiley & Sons, Chichester (West Sussex), UK, 1981.

[Williams et Lambert, 1960] W.T. Williams et J.-M. Lambert. Multivariate methods in plant ecology II: The use of an electronic digital computer for association analysis. *Journal of Ecology*, 48:689–710, 1960.

[Williams, 1984] C. Williams. *ART: The Advanced Reasoning Tool. Conceptual Overview*. Inference Corporation, Los Angeles (CA), USA, 1984.

[Wing, 1994] J. Wing. Decomposing and recomposing transaction concepts. In *Object-Based Distributed Programming*, éditeurs R. Guerraoui, O. Nierstrasz, et M. Riveill, Lecture Notes in Computer Science 791, pages 111–122. Springer--Verlag, Berlin, 1994.

[Winograd, 1968] T. Winograd. Linguistic and computer analysis of tonal harmony. *Journal of Music Theory*, 12(1):2–49, 1968.

[Winograd, 1975] T. Winograd. Frame representation and the declarative/procedural controversy. In *Representation and Understanding: Studies in Cognitive Science*, éditeurs D.G. Bobrow et A. Collins, pages 185–210. Academic Press, New York (NY), USA, 1975.

[Woods et Schmolze, 1992] W.A. Woods et J.G. Schmolze. The KL-ONE family. *Computers & Mathematics with Applications*, 23(2–5):133–177, 1992.

[Woods, 1975] W.A. Woods. What's in a link: Foundations for semantic networks. In *Representation and Understanding: Studies in Cognitive Science*, éditeurs D.G. Bobrow et A. Collins, pages 35–82. Academic Press, New York (NY), USA, 1975.

[Woods, 1991] W.A. Woods. Understanding subsumption and taxonomy: A framework for progress. In *Principles of Semantic Networks: Explorations in the Representation of Knowledge*, éditeur J.F. Sowa, pages 45–94. Morgan Kaufmann Publishers, San Mateo (CA), USA, 1991.

[Woodward et Bond, 1974] P.M. Woodward et S.G. Bond. *Algol 68-R User's Guide*. Her Majesty's Stationery Office, London, 1974.

[Wright et Fox, 1983] J.M. Wright et M.S. Fox. *SRL/1.5 User's Manual*. Robotics Institute, Carnegie-Mellon University, Pittsburgh (PA), USA, 1983.

[Yokote et Tokoro, 1987] Y. Yokote et M. Tokoro. Concurrent programming in ConcurrentSmalltalk. In *Object-Oriented Concurrent Programming*, éditeurs A. Yonezawa et M. Tokoro, pages 129–158. MIT Press, Cambridge (MA), USA, 1987.

[Yokote, 1992] Y. Yokote. The Apertos reflective operating system: The concept and its implementation. In *Proceedings of OOPSLA'92, Vancouver, Canada*, éditeur A. Paepcke, special issue of ACM SIGPLAN Notices, 27(10), pages 414–434, 1992.

[Yonezawa *et al.*, 1987] A. Yonezawa, E. Shibayama, T. Takada, et Y. Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In *Object-Oriented Concurrent Programming*, éditeurs A. Yonezawa et M. Tokoro, pages 55–89. MIT Press, Cambridge (MA), USA, 1987.

[Yonezawa et Tokoro, 1987] éditeurs A. Yonezawa et M. Tokoro. *Object-Oriented Concurrent Programming*. Computer System Series. MIT Press, Cambridge (MA), USA, 1987.

[Yonezawa, 1990] éditeur A. Yonezawa. *ABCL, an Object-Oriented Concurrent System*. Computer System Series. MIT Press, Cambridge (MA), USA, 1990.

[Yourdon, 1975] E. Yourdon. *Techniques of Program Structure and Design*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1975.

[Zendra *et al.*, 1997] O. Zendra, D. Colnet, et S. Collin. Efficient dynamic dispatch without virtual function tables: The SmallEiffel compiler. In *Proceedings of OOPSLA'97, Atlanta (GA), USA*, special issue of ACM SIGPLAN Notices, 32(10), pages 125–141, 1997.

[Zicari, 1991] R. Zicari. A framework for schema updates in an object-oriented database system. In *Proceedings of the 7th IEEE International Conference on Data Engineering (ICDE'91), Kobe, Japan*, pages 2–13, 1991.

# Index des références bibliographiques

# Index des langages, méthodes et systèmes cités

# Index des termes

### – E –



### – F –

**– I –**

**– J –**



**– K –**



**– L –**

## – N –



## – O –

**– P –**