Contribution à l'étude sémantique des bases de données : application au système SOCRATE

Rui Barbosa

▶ **To cite this version:**

Rui Barbosa. Contribution à l'étude sémantique des bases de données : application au système SOCRATE. Modélisation et simulation. Institut National Polytechnique de Grenoble - INPG; Université Joseph-Fourier - Grenoble I, 1975. Français. NNT : . tel-00285679

HAL Id: tel-00285679

https://theses.hal.science/tel-00285679

Submitted on 6 Jun 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE présentée à l'

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE

INSTITUT NATIONAL POLYTECHNIQUE

pour obtenir le grade de

docteur-ingénieur

Rui BARBOSA

CONTRIBUTION A L'ETUDE SEMANTIQUE DES BASES DE DONNEES

APPLICATION AU SYSTEME SOCRATE

Soutenue le 11 octobre 1975 devant la Commission d'examen :

Président	:	B. VAUQUOIS
Rapporteur	:	G. VEILLON
Examinateurs		J.C. BOUSSARD
		G. CORAY
		C. DELOBEL
Invité	:	J.R. ABRIAL

UNIVERSITE SCIENTIFIQUE ET MEDICALE DE GRENOBLE		INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE
M. Michel SOUTIF	Présidents	M. Louis NEEL
M. Gabriel CAU	Vice-Présidents	MM. Lucien BONNETAIN Jean BENOIT

MEMBRES DU CORPS ENSEIGNANT DE L'U.S.M.G.

PROFESSEURS TITULAIRES

MM.	ANGLES D'AURIAC Paul	Mécanique des fluides
	ARNAUD Paul	Chimie
	AUBERT Guy	Physique
	AYANT Yves	Physique approfondie
Mme	BARBIER Marie-Jeanne	Electrochimie
MM.	BARBIER Jean-Claude	Physique expérimentale
	BARBIER Reynold	Géologie appliquée
	BARJON Robert	Physique nucléaire
	BARNOUD Fernand	Biosynthèse de la cellulose
	BARRA Jean-René	Statistiques
	BARRIE Joseph	Clinique chirurgicale
	BEAUDOING André	Clinique de Pédiatrie et Puériculture
	BERNARD Alain	Mathématiques Pures
Mme	BERTRANDIAS Françoise	Mathématiques Pures
MM.	BEZES Henri	Pathologie chirurgicale
	BLAMBERT Maurice	Mathématiques Pures
	BOLLIET Louis	Informatique (IUT B)
	BONNET Georges	Electrotechnique
	BONNET Jean-Louis	Clinique ophtalmologique
	BONNET-EYMARD Joseph	Pathologie médicale
	BOUCHERLE André	Chimie et toxicologie
	BOUCHEZ Robert	Physique nucléaire
	BOUSSARD Jean-Claude	Mathématiques appliquées
	BRAVARD Yves	Géographie
	CABANEL Guy	Clinique rhumatologique et hydrologie
	CALAS François	Anatomie
	CARLIER Georges	Biologie végétale
	CARRAZ Gilbert	Biologie animale et pharmacodynamie
	CAU Gabriel	Médecine légale et toxicologie
	CAUQUIS Georges	Chimie organique
	CHABAUTY Claude	Mathématiques Pures
	CHARACHON Robert	Clinique Oto-Rhino-Laryngologique
	CHATEAU Robert	Thérapeutique (Neurologie)
	CHIBON Pierre	Biologie animale
	COEUR André	Pharmacie chimique et chimie analytique
	CONTAMIN Robert	Clinique gynécologique
	COUDERC Pierre	Anatomie pathologique
	CRAYA Antoine	Mécanique
Mme	DEBELMAS Anne-Marie	Matière médicale
MM.	DEBERMAS Jacques	Géologie générale
	DEGRANGE Charles	Zoologie
	DELORMAS Pierre	Pneumo-Phtisiologie
	DEPORTES Charles	Chimie minérale
	DESRE Pierre	Métallurgie
	DESSAUX Georges	Physiologie animale
	DODU Jacques	Mécanique appliquée

MM.	DOLIQUE Jean-Michel	Physique des plasmas
	DREYFUS Bernard	Thermodynamique
	DRUCROS Pierre	Cristallographie
	DUGOIS Pierre	Clinique de dermatologie et syphiligraph
	FAU René	Clinique neuro-psychiatrique
	GAGNAIRE Didier	Chimie physique
	GALLISSOT François	Mathématiques pures
	GALVANI Octave	Mathématiques pures
	GASTINEL Noël	Mathématiques appliquées
	GAVEND Michel	Pharmacologie
	GEINDRE Michel	Electroradiologie
	GERBER Robert	Mathématiques pures
	GERMAIN Jean-Pierre	Mécanique
	GIRAUD Pierre	Géologie
	JANIN Bernard	Géographie
	KAHANE André	Physique Générale
	KLEIN Joseph	Mathématiques pures
	KOSZUL Jean-Louis	Mathématiques pures
	KRAVTCHENKO Julien	Mécanique
	KUNTZMANN Jean	Mathématiques appliquées
	LACAZE Albert	Thermodynamique
	LACHARME Jean	Biologie végétale
	LAJZEROWICZ Joseph	Physique
	LATREILLE René	Chirurgie générale
	LATURAZE Jean	Biochimie pharmaceutique
	LAURENT Pierre-Jean	Mathématiques appliquées
	LEDRU Jean	Clinique médicale B
	LLIBOUTRY Louis	Géophysique
	LONGEQUEUE Jean-Pierre	Physique nucléaire
	LOUP Jean	Géographie
Mle	LUTZ Elisabeth	Mathématiques pures
	MALGRANGE Bernard	Mathématiques pures
	MALINAS Yves	Clinique obstétricale
	MARTIN-NOEL Pierre	Seméiologie médicale
	MAZARE Yves	Clinique médicale A
	MICHEL Robert	Minéralogie et pétrographie
	MICOUD Max	Clinique maladies infectieuses
	MOURIQUAND Claude	Histologie
	MOUSSA André	Chimie nucléaire
	MULLER Jean-Michel	Thérapeutique (néphrologie)
	NEEL Louis	Physique du solide
	OZENDA Paul	Botanique
	PAYAN Jean-Jacques	Mathématiques pures
	PEBAY-PEYROULA Jean-Claude	Physique
	RASSAT André	Chimie systématique
	RENARD Michel	Thermodynamique
	RINALDI Renaud	Physique
	DE ROUGEMONT Jacques	Neuro-chirurgie
	SEIGNEURIN Raymond	Microbiologie et hygiène
	SENGEL Philippe	Zoologie
	SIBILLE Robert	Construction mécanique
	SOUTIF Michel	Physique générale
	TANCHE Maurice	Physiologie
	TRAYNARD Philippe	Chimie générale
	VAILLANT François	Zoologie
	VALENTIN Jacques	Physique nucléaire
	VAUQUOIS Bernard	Calcul électronique
Mme	VERAIN Alice	Pharmacie galénique
MM.	VERAIN André	Physique
	VEYRET Paul	Géographie
	VIGNAIS Pierre	Biochimie médicale
	YOCCOZ Jean	Physique nucléaire théorique

PROFESSEURS ASSOCIES

MM.	CHEEKE John	Thermodynamique
	COPPENS Philip	Physique
	CORCOS Gilles	Mécanique
	CRABBE Pierre	CERMO
	GILLESPIE John	I.S.N.
	ROCKAFELLAR Ralph	Mathématiques appliquées

PROFESSEURS SANS CHAIRE

Mlle	AGNIUS-DELORD Claudine	Physique pharmaceutique
	ALARY Josette	Chimie analytique
MM.	AMBROISE-THOMAS Pierre	Parasitologie
	BELORIZKY Elie	Physique
	BENZAKEN Claude	Mathématiques appliquées
	BERTRANDIAS Jean-Paul	Mathématiques pures
	BIAREZ Jean-Pierre	Mécanique
	BILLET Jean	Géographie
Mme	BONNIER Jane	Chimie générale
MM.	BOUCHET Yves	Anatomie
	BRUGEL Lucien	Energétique
	CONTE René	Physique
	DEPASSEL Roger	Mécanique des fluides
	GAUTHIER Yves	Sciences biologiques
	GAUTRON René	Chimie
	GIDON Paul	Géologie et Minéralogie
	GLENAT René	Chimie organique
	GROULADE Joseph	Biochimie médicale
	HACQUES Gérard	Calcul numérique
	HOLLARD Daniel	Hématologie
	HUGONOT Robert	Hygiène et Méd. Préventive
	IDELMAN Simon	Physiologie animale
	JOLY Jean-René	Mathématiques pures
	JULLIEN Pierre	Mathématiques appliquées
Mme	KAHANE Josette	Physique
MM.	KUHN Gérard	Physique
	LOISEAUX Jean	Physique nucléaire
	LUU-DUC-Cuong	Chimie organique
	MAYNARD Roger	Physique du solide
	PELMONT Jean	Biochimie
	PERRIAUX Jean-Jacques	Géologie et minéralogie
	PFISTER Jean-Claude	Physique du solide
Mlle	PIERY Yvette	Physiologie animale
MM.	RAYNAUD Hervé	Mathématiques appliquées
	REBECQ Jacques	Biologie (CUS)
	REVOL Michel	Urologie
	REYMOND Jean-Charles	Chirurgie générale
	RICHARD Lucien	Biologie végétale
Mme	RINAUDO Marguerite	Chimie macromoléculaire
MM.	ROBERT André	Chimie papetière
	SARRAZIN Roger	Anatomie et chirurgie
	SARROT-REYNAULD Jean	Géologie
	SIROT Louis	Chirurgie générale
Mme	SOUTIF Jeanne	Physique générale
MM.	VIALON Pierre	Géologie
	VAN CUTSEM Bernard	Mathématiques appliquées

MAITRES DE CONFERENCES ET MAITRES DE CONFERENCES AGREGES

MM.	AMBLARD Pierre	Dermatologie
	ARMAND Gilbert	Géographie
	ARMAND Yves	Chimie
	BARGE Michel	Neurochirurgie
	BEGUIN Claude	Chimie organique
Mme	BERIEL Hélène	Pharmacodynamique
M.	BOUCHARLAT Jacques	Psychiatrie adultes
Mme	BOUCHE Liane	Mathématiques (CUS)
MM.	BRODEAU François	Mathématiques (IUT B)
	BUISSON Roger	Physique
	BUTEL Jean	Orthopédie
	CHAMBAZ Edmond	Biochimie médicale
	CHAMPETIER Jean	Anatomie et organogénèse
	CHARDON Michel	Géographie
	CHERADAME Hervé	Chimie papetière
	CHIAVERINA Jean	Biologie appliquée (EFP)
	COHEN-ADDAD Jean-Pierre	Spectrométrie physique
	COLOMB Maurice	Biochimie médicale
	CORDONNIER Daniel	Néphrologie
	COULOMB Max	Radiologie
	CROUZET Guy	Radiologie
	CYROT Michel	Physique du solide
	DELOBEL Claude	M.I.A.G.
	DENIS Bernard	Cardiologie
	DOUCE Roland	Physiologie végétale
	DUSSAUD René	Mathématiques (CUS)
Mme	ETERRADOSSI Jacqueline	Physiologie
MM.	FAURE Jacques	Médecine légale
	FONTAINE Jean-Marc	Mathématiques pures
	GAUTIER Robert	Chirurgie générale
	GENSAC Pierre	Botanique
	GIDON Maurice	Géologie
	GRIFFITHS Michaël	Mathématiques appliquées
	GROS Yves	Physique (stag.)
	GUITTON Jacques	Chimie
	HICTER Pierre	Chimie
	IVANES Marcel	Electricité
	JALBERT Pierre	Histologie
	KOLODIE Lucien	Hématologie
	KRAKOWIAK Sacha	Mathématiques appliquées
Mme	LAJZEROWICZ Jeannine	Physique
MM.	LEROY Philippe	Mathématiques
	MACHE Régis	Physiologie végétale
	MAGNIN Robert	Hygiène et médecine préventive
	MARECHAL Jean	Mécanique
	MARTIN-BOUYER Michel	Chimie (CUS)
	MICHOULIER Jean	Physique (IUT A)
Mme	MINIER Colette	Physique
MM.	NEGRE Robert	Mécanique
	NEMOZ Alain	Thermodynamique
	PARAMELLE Bernard	Pneumologie
	PECCOUD François	Analyse (IUT B)
	PEFFEN René	Métallurgie
	PERRET Jean	Neurologie
	PHELIP Xavier	Rhumatologie
	RACHAIL Michel	Médecine interne
	RACINET Claude	Gynécologie et obstétrique
	RAMBAUD Pierre	Pédiatrie
Mme	RENAUDET Jacqueline	Bactériologie
MM.	ROBERT Jean-Bernard	Chimie-Physique

MM.	ROMIER Guy	Mathématiques (IUT B)
	SHOM Jean-Claude	Chimie générale
	STIEGLITZ Paul	Anesthésiologie
	STOEBNER Pierre	Anatomie pathologique
	VROUSOS Constantin	Radiologie

MAITRES DE CONFERENCES ASSOCIES

MM.	COLE Antony	Sciences nucléaires
	FORELL César	Mécanique
	MOORSANI Kishin	Physique

CHARGES DE FONCTIONS DE MAITRES DE CONFERENCES

MM.	BOST Michel	Pédiatrie
	CONTAMIN Charles	Chirurgie thoracique et cardio-vasculaire
	FAURE Gilbert	Urologie
	MALLION Jean-Michel	Médecine du travail
	ROCHAT Jacques	Hygiène et hydrologie

Fait à Saint Martin d'Hères, OCTOBRE 1974.

"MEMBRES DU CORPS ENSEIGNANT DE L'I.N.P.G."

PROFESSEURS TITULAIRES

MM.	BENOIT Jean	Radioélectricité
	BESSON Jean	ELectrochimie
	BONNETAIN Lucien	Chimie Minérale
	BONNIER Etienne	Electrochimie, Electrométallurgie
	BRISSONNEAU Pierre	Physique du solide
	BUYLE-BODIN Maurice	ELectronique
	COUMES André	Radioélectricité
	FELICI Noël	Electrostatique
	PAUTHENET René	Physique du solide
	PERRET René	Servomécanismes
	SANTON Lucien	Mécanique
	SILBER Robert	Mécanique des fluides

PROFESSEUR ASSOCIE

M.	BOUDOURIS Georges	Radioélectricité

PROFESSEURS SANS CHAIRE

MM.	BLIMAN Samuel	Electronique
	BLOCH Daniel	Physique du solide et cristallographie
	COHEN Joseph	Electrotechnique
	DURAND Françis	Metallurgie
	MOREAU René	Mécanique
	POLOUJADOFF Michel	ELectrotechnique
	VEILLON Gérard	Informatique fondamentale et appliquée
	ZADWORNY François	Electronique

MAITRES DE CONFERENCES

MM.	BOUVARD Maurice	Génie mécanique
	CHARTIER Germain	Electronique
	FOULARD Claude	Automatique
	GUYOT Pierre	Chimie minérale
	JOUBERT Jean Claude	Physique du solide
	LACOUME Jean Louis	Géophisique
	LANCIA Roland	Physique atomique
	LESPINARD Georges	Mécanique
	MORET Roger	Electrotechnique nucléaire
	ROBERT François	Analyse numérique
	SABONNADIERE Jean Claude	Informatique fondamentale et appliquée
Mme	SAUCIER Gabrièle	Informatique fondamentale et appliquée

MAITRE DE CONFERENCES ASSOCIE

M.	LANDAU Ioan Doré	Automatique

CHARGE DE FONCTIONS DE MAITRES DE CONFERENCES

M.	ANCEAU François	Mathématiques appliquées

Fait à Saint Martin d'Hères JANVIER 1974

p.a.f.p.l.p.c.s.p.

Je dois dire que le fait d'écrire une page de remerciements m'enlève toute ma spontanéité même s'ils sont écrits à la main ...

Cependant, je crois qu'il est important de reconnaître publiquement les contributions qui m'ont été apportées.

Je voulais que le lecteur sache que M. B. Vauquois, Professeur à l'Université I de Grenoble, a accepté de présider le jury de cette thèse dans des conditions difficiles. Je lui en suis très reconnaissant.

G. Veillon, Professeur à l'Institut National Polytechnique de Grenoble et Directeur du Laboratoire d'Informatique, a accepté de rapporter mon travail devant l'Uni

versité de Grenoble. Tâche que je comprends être delicate entre autres parcequ'il n'avait pas suivi ses differentes etapes et que ce lui-ci lui a été presenté pratiquement fini. J'ai amelioré des aspects qu'il m'a signalés concernant la structure et la correction des programmes mais je comprends que beaucoup reste encore à faire.

J.C. Boussard, Professeur à l'Universit de Nice, a su m'<u>ecouter</u> (cela veut dire me <u>comprendre</u>), a su orienter mon travail depuis la Maîtrise, et c'est quelqu'un qui, en consequence, m'a <u>beaucoup</u> aidé sur le plan <u>personnel</u> et profession

G. Coray, Professeur à l'Ecole Polytechnique de Lausanne, est pour moi le Professeur aux idées claires et rigoureuses et quelqu'un qui a toujours été disponible pour m'aider et faire com

prendre les limmitations de mon travail et aussi ses qualités.

Je remercie C. Delobel, Maître de Conférences à l'Université I de Grenoble, d'avoir pris le temps et la patience de lire ma Partie 1 et de m'avoir fait des critiques très utiles qui m'ont permis de lui donner une forme nettement plus condensée et rigoureuse.

Tout le travail décrit dans cette thèse a été fait sous la direction de J. R. Abrial, dans l'équipe SOCRATE. J'ai beaucoup appris dans les multiples discussions qu'on a eues où j'ai pu confronter mes rêves de nouveaux systèmes à sa compétence et sens des réalités. Je n'ai pu poursuivre ma recherche que grâce au soutien qu'il m'a toujours accepté de donner même quand il était dif-

ficile de prevoir des resultats positifs.

Avec J.C. Favre j'ai appris beau-coup d'aspects essentiels du langage SOCRA-TE. Son idée de structure universelle est déjà un pas vers l'étude systematique des structures SOCRATE. Il y avait longtemps qu'il avait compris le besoin d'un langage plus symetrique et evolué. En ce qui concerne l'apport de l'Intelligence Artificielle, il a su finalement depasser ses reti-cences initiales et il faut dire que le prototype SOCRATE-P represente un très bon travail d'implementation qu'il vaudrait la peine de poursuivre.

Sans J.C. Profizi je n'aurais pas pu m'apercevoir du decalage qui exis-tait (et qui existe encore!...) entre ce que je voudrais que les autres compren-nent et ce qu'ils comprennent réelle-

ment dû à la mauvaise qualité de ma façon d'exprimer.

M. Trevisan est une virtuose. Sans quoi elle n'aurait pas pu donner l'excellente disposition qu'elle a reussi pour le texte etant donnée la mauvaise qualité du manuscrit et la grande quantité de symboles speciaux. Je lui suis très reconnaissant de sa patience et de tout ce qu'elle a resolu concernant la soutenance.

D. Iglesias a eu la patience de m'attendre, m'attendre indefiniment et resoudre les problèmes de tirage dans des conditions de travail très difficiles. Je lui remercie et à tous ceux qui ont participé au tirage.

Je voulais faire ici etat de la contribution que m'ont donné B. Cassagne

P. Fontanille et A. Lux en me faisant des remarques sur le texte à propos du Français ou d'aspects techniques.

Je n'oublie L. Hocine, Professeur à la Faculté de Droit qui m'a donné un très bon exemple que j'ai implementé.

TABLE DES MATIERES

PARTIE 0 - SITUATION ET OBJECTIFS

0.1. INTRODUCTION

0.2. LANGAGE INDEPENDANT DE LA REPRESENTATION DES DONNEES - DATA INDEPENDANCE
- 0.2.1. Dépendance vis-à-vis de l'organisation des données
- 0.2.2. Dépendance du fait qu'une donnée soit lue ou déduite par programme
  Relations inverses
- 0.2.3. Conclusion

0.3. DESCRIPTION DE DONNEES COMPLETES
- 0.3.1. Données déduites - Données calculées
- 0.3.2. Cohérence des données avec l'environnement réel
  Planification
- 0.3.3. Exemple de modèle sémantique complet
- 0.3.4. Conclusion

0.4. ADEQUATION DU LANGAGE-UTILISATEUR

PARTIE I - SEMANTIQUE DE SOCRATE - CE QUI MANQUE

1.0. INTRODUCTION

1.1. STRUCTURE SOCRATE
- 1.1.1. Exemples
- 1.1.2. Objets et relations

1.2. SEMANTIQUE DE LA STRUCTURE SOCRATE
- 1.2.1. Les "caractéristiques"
  - entités, entités imbriquées, inverses, Mots, Textes, Naturels, listes - conclusion
- 1.2.2. Implantation des relations dans l'espace virtuel- I-relations
- 1.2.3. Modèle sémantique :Obtention d'un I-graphe , obtention d'un modèle sémantique

1.3. SEMANTIQUE DU LANGAGE D'INTERROGATION ET MISE-A-JOUR
1.3.1. Le "ayant" et le "de"
1.3.2. Connecteurs et, ou et __
1.3.3. Composition de questions

1.4. APPLICATIONS IMMEDIATES DE LA FORMALISATION PRECEDENTE
1.4.1. Changement de structure : structures équivalentes, questions équivalentes
1.4.2. Choix d'une structure
1.4.3. Comment améliorer une structure
1.4.4. Structures faiblement équivalentes

1.5. CE QUI MANQUE EN SOCRATE
1.5.1. Modèle Sémantique non implémentable
1.5.2. Contraintes sur les relations n-aires
1.5.3. Les redondances
1.5.4. Différences entre I-relations, P(père) et R(référence)
1.5.5. Structures faiblement équivalentes
1.5.6. Règles sémantiques ad-hoc

1.6. CONCLUSION

PARTIE II - MODELE SEMANTIQUE COMPLET

2.1. INTRODUCTION
2.2. MODELE SEMANTIQUE
2.2.1. Catégories
2.2.2. Propriétés catégorielles : "prpcg"
2.2.3. Relations
2.2.4. Bibliographie

2.2. OPERATEURS ELEMENTAIRES
2.2.1. Opérations sur les c-catégories
2.2.2. Opérations sur les a-catégories
2.2.3. Opérations sur les fonctions d'accès
2.2.4. Tableau récapitulatif des opérateurs élémentaires

2.3. OPERATIONS SUR LES RELATIONS
2.3.1. Relations unaires (prp)
2.3.2. Relations binaires
2.3.3. Relations de degré supérieur à 2

2.4. MODELE SEMANTIQUE COMPLET
2.4.1. Règles sémantiques ad-hoc : METHODES
2.4.2. Exemples.
2.4.3. Changements de méthode

2.5. CONCLUSION

PARTIE III - LANGAGE SEMANTIQUE

3.0. INTRODUCTION
3.1. ENONCE D'UN PROBLEME
3.1.1. Preuves, déductions et transformations
3.1.2. Enoncés composés

3.2. RESOLUTION D'UN PROBLEME
3.2.1. Etat de la résolution
3.2.2. Interruption de la résolution
3.2.3. Processus de résolution d'un problème
3.2.4. Espace

3.3. UTILISATION DU SYSTEME

3.4. CONCLUSION

PARTIE IV - CONCLUSION

4.1. BILAN

4.2. EXEMPLES IMPLEMENTES EN SOCRATE-P

Exemple de gestion

Exemple d'Intelligence Artificielle (cubes)

Exemple de Droit

Exemple des descendants

PARTIE 0

SITUATION ET OBJECTIFS

C.1. INTRODUCTION

Les systèmes de bases de données actuels, comme SOCRATE, se chargent des tâches suivantes :

- stockage organisé de grandes masses de données,
- accès et mise-à-jour sélectifs de ces données au moyen d'analyses logiques pouvant varier entre la sélection booléenne simple, une Algèbre Relationnelle (6,10), ou des formes souvent limitées, du Calcul de Prédicats au 1er ordre (8,11).

Leur rôle est donc de remplacer les supports d'informations classiques et une partie de l'activité humaine de sélection et organisation des informations.

Le premier volet de mon travail sera d'analyser les causes de quelques difficultés graves que pose actuellement l'emploi de SOCRATE et qui sont aussi valables pour la majorité des autres systèmes :

- obligation de tout recommencer depuis zéro (définition de structure, rentrée des données, écriture des programmes) quand l'organisation des données change.
- difficulté, sinon impossibilité, de comprendre et de contrôler la sémantique des programmes utilisateurs.

Le deuxième volet de mon travail consistera à comprendre les limitations de l'emploi des systèmes de bases de données et de là à proposer que plus de tâches actuellement laissées à la charge de l'utilisateur ou de systèmes ad-hoc soient confiées aux nouveaux systèmes. Je pense surtout aux opérations d'accès nécessaires pour déduire une donnée qui n'est pas stockée et aux opérations d'accès et mise-à-jour nécessaires pour garantir la cohérence des données avec l'environnement réel lorsqu'une donnée est changée (par exemple, toutes les questions et mises-à-jour qu'entraîne le changement du prix d'une pièce dans la base de données d'une entreprise).

En Gestion, la prise en charge par le système de ces tâches de déduction de données et le maintien de la cohérence avec l'environnement réel permettra qu'il soit utilisé pour fournir des visions très élaborées des différentes parties de l'entreprise et prévoir les réactions de celle-ci à une prise de décision. La simulation du système devient alors une composante plus importante du processus personnel de réflexion du gestionnaire.

0.1.1. Organisation de la thèse

Cette thèse est organisée en 5 parties.

Chaque partie possède une introduction et une conclusion. L'introduction sert à situer la matière traitée et à la résumer. La conclusion sert à comparer ce qui a été fait dans la partie, avec d'autres solutions existantes qu'on peut trouver dans la bibliographie.

A partir du paragraphe 0.2, je présenterai la situation actuelle des systèmes de bases de données et j'expliquerai les propriétés qui me paraissent nécessaires pour de nouveaux systèmes, satisfaisant aux objectifs que j'ai exposé au paragraphe 0.1. :

1 - langage indépendant de la représentation des données (data-independance)

2 - description de données complète

3 - langage-utilisateur adéquat.

Dans la Partie 1, j'étudie le langage SOCRATE (la structure SOCRATE et le langage utilisateur) et je formalise sa sémantique.

Je montre ensuite comment cette formalisation peut-elle servir à résoudre, à court terme des problèmes que pose l'emploi de SOCRATE tels que le changement de structure, le partage de programmes entre applications structurées différemment (réparties sur un réseau, par exemple) et les choix de structures (1.4).

Enfin, je m'en sers pour définir rigoureusement ce qui manque en SOCRATE pour qu'il permette de décrire complètement la sémantique des environnements réels(1.5).

Dans les Parties 2,3 et 4, je présente le deuxième volet du travail qui est, comme je l'ai dit, la proposition de nouveaux systèmes qui ont les 3 propriétés indiquées et devraient répondre donc aux objectifs consignés dans l'introduction (§ 0.1).

Cette prosition est basée sur le travail effectué sur SOCRATE (partie 1) sur les travaux publiés par E. CODD [6] [7] [8] [9] et sur les nouveaux concepts qu'ont apporté à la Théorie de la Programmation un certain nombre de nouveaux langages d'Intelligence Artificielle: nouveaux types de données symboliques et modes d'accès, mécanismes de déduction et "problem-solving" et nouvelles structures de contrôle.

La description de données est traitée dans la Partie 1 et le langage-utilisateur dans la Partie 3.

En 2.1.5., je compare la description de données avec celle que propose E.W. CODD. Dans les conclusions des deux Parties,je compare les concepts que j'ai introduits, avec ceux du langage d'Intelligence Artificielle cités.

Dans la partie 4, je fais un bilan critique du travail réalisé et je propose des voies de recherche qui à mon avis, en découlent.

Ce bilan comprend, entre autres, l'analyse de trois exemples que j'ai programmé dans une version expérimentale de système - SOCRATE-P - (1) (J.C. Favre) :

- un exemple de Gestion
- un exemple d'Intelligence Artificielle
- un exemple de Droit

---

(1) - qui sont présentés dans l'annexe de la Partie 4.

C.2. LANGAGE INDEPENDANT DE LA REPRÉSENTATION DES DONNÉES - DATA INDÉPENDANCE

La première propriété qui me paraît nécessaire pour les nouveaux systèmes est que le langage doit avoir une syntaxe indépendante de la représentation des données en mémoire, ce que j'appellerai "être data-indépendant". J'explique pourquoi.

Dans un langage de programmation quelconque et pas seulement en bases de données, la codification d'un traitement passe par :

1 - sa définition,
2 - l'analyse de ce dont il aura besoin pour s'exécuter
3 - le choix d'une représentation des données [I] répondant le mieux possible à cette analyse
4 - la codification proprement dite.

Très souvent apparaissent des difficultés quand l'exécution du programme ne se déroule pas de la façon prévue dans la phase d'analyse et le choix fait en 3 ne répond plus aux exigences en espace, temps ou facilité d'utilisation.

Si on veut les résoudre, deux solutions sont possibles :

a) accepter de continuer avec la représentation choisie,
b) changer de représentation.

Or, si le langage a une syntaxe qui dépend de la représentation, la deuxième solution oblige à refaire complètement tous les programmes déjà écri (Malheureusement, ceci ne veut pas dire que les difficultés soient disparues pou toujours et que l'on n'aura pas à recommencer peu de temps après ...)

[I] Je prends "donnée" au sens "information" et non pas au sens "entrée d'un programme". Je considère que parmi les représentations possibles de données, il y a les procédures, les données représentées étant leurs sorties.

Cet effort de reécriture de programmes, qui est forcément précédé d'un effort de déchiffrage des programmes existants, est une des raisons du coût élevé des systèmes actuels.

Pour le domaine des bases de données, ce double effort de déchiffrage et réécriture est évalué en [12] à 50 % de **l'effort** général de programmation !

Un deuxième genre de difficultés causées par la dépendance de la syntaxe des programmes vis-à-vis de la représentation des données est que l'attention des utilisateurs est détournée de l'essentiel, qui est la sémantique de ces mêmes programmes. Celle-ci est souvent très difficile à comprendre et à manipuler (les programmeurs SOCRATE en savent quelque chose ...)

Je vais faire ensuite un exposé des diverses manifestations possibles de la dépendance des programmes vis-à-vis de la représentation des données.

0.2.1. Dépendance vis-à-vis de l'organisation des données

Ce type de dépendance apparaît dans les systèmes où les bases de données sont organisées en arbre ou réseau, cette organisation n'étant pas dûe à la sémantique des données et où l'accès à une donnée s'exprime en explicitant le chemin à parcourir sur l'arbre ou réseau pour l'atteindre. C'est le cas de SOCRATE [3], IMS [14], CODASYL [6], IDS [6], GIS [6], TDMS [6].

La structure SOCRATE a la forme de réseau puisqu'elle est construite en "niveaux hiérarchiques" et qu'on peut ajouter des pointeurs (appelés "références"). L'accès à une donnée s'exprime en indiquant les niveaux successifs qui la séparent du niveau le plus haut.

Par exemple, si la structure est :

```
entité produit
    début
        prix-d-achat naturel 0 à 2000                [I]
        prix-de-vente naturel 0 à 2000
        entité vente
            début
                quantité naturel 0 à 1000
            fin
    fin
```

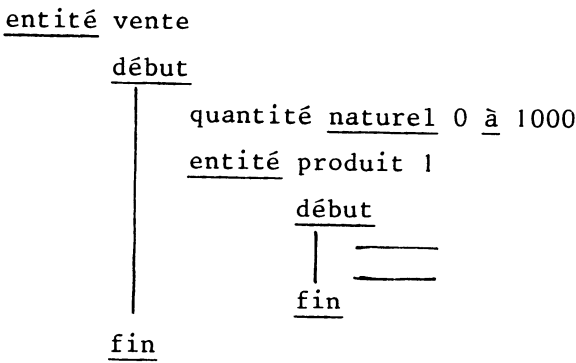
l'accès aux quantités des ventes d'un produit P s'exprime ainsi :

quantité de vente de P

parce que 'quantité' est "fille" de 'vente' elle-même "fille" de 'produit'.

L'inconvénient de la dépendance vis-à-vis du chemin d'accès est que, si on veut changer les niveaux, pour mieux organiser les données, il faut réécrire tous les programmes existants.

Si, par exemple, on voulait changer pour la structure :



L'accès vu s'exprimerait différemment :

quantité de vente ayant produit = P

[I] rigoureusement il fallait écrire de à la place de naturel

0.2.2. Dépendance du fait qu'une donnée soit lue ou déduite par programme

En SOCRATE, si on voulait définir le bénéfice d'un produit x comme étant la différence entre le prix de vente et le prix d'achat de x, il fallait prévoir une procédure :

```
proc bénéfice

    return (prix-de-vente de x - prix-d-achat de x)

fin
```

et si on voulait demander le bénéfice d'un produit P il fallait écrire :

call bénéfice P

alors que, si on prenait la décision d'enregistrer les bénéfices des produits, de façon analogue à ce que l'on fait pour leurs prix de vente et d'achat, il fallait l'écrire autrement :

bénéfice de P

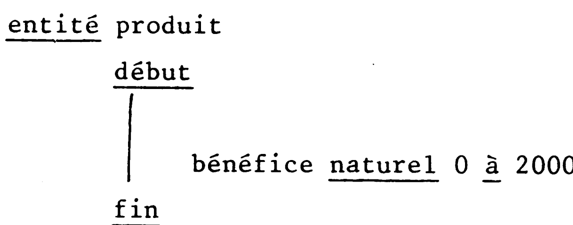
Tous les systèmes de bases de données que je connais, à l'exception de SYNTEX [11], ont cette dépendance du langage vis-à-vis du fait qu'une donnée soit lue ou déduite par programme.

Dans d'autres domaines que les bases de données, ce sont POP-2, DATALESS PROGRAMMING [30] et ensuite PLANNER [35] qui l'on éliminée les premiers.

Relations inverses

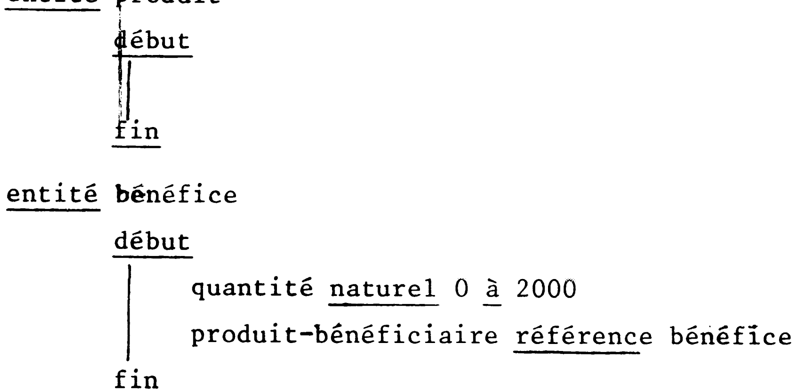
Encore dans ce type de dépendance peut être classée la dépendance du langage vis-à-vis du fait que ce soit une relation R ou son inverse R^{-1} qui est stockée, parce que, évidemment, les réalisations d'une relation peuvent toujours être déduites à partir de celles de son inverse.

Par exemple, il peut être utile de, au lieu de la structure :



```
entité produit
    début
    |
    |    bénéfice naturel 0 à 2000
    fin
```

qui favorise l'accès aux bénéfices des produits, avoir la structure :



```
entité produit
    début
    |
    fin
entité bénéfice
    début
    |    quantité naturel 0 à 2000
    |    produit-bénéficiaire référence bénéfice
    fin
```

afin de favoriser plutôt l'accès aux produits ayant un certain bénéfice. Evidemment ces deux structures sont équivalentes et ne diffèrent que parce qu'avec la première c'est la relation "bénéfice" qui est stockée, alors qu'avec la deuxième c'est son inverse.

La dépendance existe parce que, si l'accès au bénéfice d'un produi P s'exprime dans le premier cas :

bénéfice de P

dans le deuxième cas, s'exprimera autrement :

quantité de bénéfice ayant produit-bénéficiaire = P

Puisque la sémantique est la même dans les deux cas, il faudrait que l'expression soit aussi la même.

0.2.3. Conclusion

Les trois types de dépendance que j'ai analysés, auxquels je pourrai encore ajouter la dépendance vis-à-vis d'une indiciation [6], devront disparaître.

La solution que je propose et que je développerai dans les Parties 2 et 3 est qu'il y ait deux niveaux distincts dans les nouveaux systèmes :

- niveau sémantique, niveau auquel l'utilisateur fait la description de la sémantique des données au moyen d'un Modèle Sémantique, où n'intervient aucune indication pour la représentation en mémoire. Les questions et mises-à-jour sont formulées dans un Langage Sémantique dont la syntaxe ne dépend que du Modèle Sémantique et est donc indépendante de la représentation des données.

- niveau mémoire, niveau auquel doit intervenir un choix de l'organisation des données, et de la lecture ou déduction comme moyen d'accès.

A chacun de ces deux niveaux correspond une compétence différente : l'utilisateur de tous les jours travaillera au niveau sémantique, alors que ce sera au responsable de prendre des décisions au niveau mémoire.

Cette séparation des deux niveaux donnera au système la data-indépendance voulue.

0.3. DESCRIPTION DE DONNÉES COMPLÈTE

La deuxième propriété-clé que je propose pour les nouveaux systèmes est que le Modèle Sémantique (0.2.3)soit complet.

En effet, la description de données dans les systèmes actuels est incomplète. Je vais expliquer pourquoi.

Décrire la sémantique des données consiste à déclarer les types d'objets qui seront stockées dans la base de données (par exemple les produits, les nombres naturels entre 0 et 2000, etc...) et les relations pouvant relier ces objets (par exemple la relation 'bénéfice' entre des produits et des nombres naturels entre 0 et 2000) avec l'indication pour chaque relation des types obligés des arguments (par exemple les deux arguments de 'bénéfice' doivent être un produit et un nombre naturel).

Déjà, en ce qui concerne les types d'objets et les relations, les systèmes actuels sont souvent incomplets : RELATIONAL MODEL [9] et MAC-AIMS [10] ne permettent pas de définir deux relations différentes entre les mêmes objets, ni faire des déclarations récursives de relations (2.1.5.) ; RAND CORPORATION [15] , DBSIA [16] et LEAP [17] ne permettent pas de déclarer une relation n-aire que comme l'imbrication de relations binaires ; SOCRATE [3] ne permet pas de déclarer une relation bijective (1.5.1.) ni d'établir des contraintes sur les réalisations d'une relation n-aire (1.5.2.).

Mais, plus que cà, la sémantique que l'on décrit en déclarant les types et relations ne représente en réalité, qu'une très faible partie de la sémantique des environnements réels : toute sorte de règles sémantiques ad-hoc reste inexprimable.

Je vais expliquer ce que j'entends par là.

0.3.1. Données déduites

Avec les systèmes actuels, on profite très peu d'une base de données, parce que ceux-ci ne fournissent que les données explicitement stockées dans celle-ci.

Il faudrait qu'ils puissent aussi déduire des données en appliquant des règles sémantiques ad-hoc aux données stockées.

Je vais expliquer pourquoi.

Soit, par exemple, la base de données d'une entreprise commerciale.

Supposons qu'elle décrit des produits composés d'autres produits, eux-mêmes composés, et ainsi de suite.

Si le nombre moyen de niveaux de décomposition des produits est 5, et le nombre moyen de composants directs de chaque produit est 2, alors, pour que le système puisse fournir les composants d'un produit, il faut stocker 62 composants par produit, c'est-à-dire 310.000 composants s'il y a 5000 produits.

Cependant, si on pouvait stocker la règle sémantique sur les composants d'un produit x sous l'une des deux formes suivantes :

1 - pour énumérer les composants de x il suffit d'énumérer :
   - les composants directs de x
   - les composants des composants directs de x

2 - pour prouver qu'un produit x est composant d'un autre y il suffit de prouver que x est composant-direct de y ou alors que x est composant d'un composant direct de y.

alors il suffirait de stocker pour chaque produit ses 2 composants directs.
Cela ferait aussi qu'il ne serait plus nécessaire de mettre à jour des composants, chaque fois qu'on mettrait à jour les composants directs.

Données calculées

Un cas particulier très important de données déduites est celui des données obtenues par un calcul sur des données stockées, et que j'appellerai "données calculées".

Reprenons l'exemple présenté en 0.2.2. , des produits et leurs bénéfices.

De façon analogue à ce que j'ai dit pour les composants, si on stockait la règle sémantique qui dit que le bénéfice d'un produit est la différence entre son prix de vente et son prix d'achat, alors il suffit de stocker les prix de vente et achat pour que le système puisse fournir les bénéfices des produits.

La forme de la question sera la même que si le bénéfice était explicitement stocké.

En page II.30 , je présenterai une solution mixte qui consiste à stocker le bénéfice après sa déduction.

0.3.2. Cohérence des données avec l'environnement réel

Un autre aspect des systèmes actuels est que, s'ils permettent de stocker les propriétés d'un environnement réel par contre, ils ne permettent pas de représenter les multiples interdépendances qui existent entre elles.

Par exemple, on peut stocker le prix de vente d'un produit, le prix de ses composants, les prévisions de vente qui le concernent mais on ne peut pas représenter les liens qui existent entre ces quantités dans l'environnement réel.

Conséquence immédiate de ceci, à la suite d'une mise-à-jour, l'état des données n'est plus cohérent avec l'environnement réel.

C'est à l'utilisateur de garantir les mises-à-jour qui les rendent cohérent, ce qui est souvent délicat à faire et accroît considérablement les erreurs.

Planification

Un avantage de ce maintien de cohérence, qui me paraît particulièrement important, est l'utilisation des systèmes de bases de données en prise de décisions (Gestion). Si le maintien de cohérence était fait "instanément" par le système, alors le décideur pourrait mesurer les répercussions de sa décision sur l'ensemble de l'entreprise.

Si, en plus, les mises-à-jour pouvaient être faites dans un espace local, le décideur pourrait planifier sa prise de décision, en essayant différentes mises-à-jour et en mesurant, pour chacune, l'état final des données.

Il reste maintenant à savoir comment représenter les interdépendances dont j'ai parlé.

Reprenons l'exemple du changement de prix d'un produit.

Si on veut représenter la règle qui dit que l'augmentation du prix de vente d'un produit P :

* provoque l'augmentation du prix-de-vente des produits dont P est composant
* provoque la diminution des prévisions de vente de P, d'une quantité F (Δp) où Δp est l'augmentation du prix.

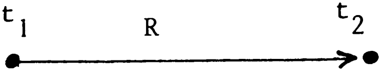
il suffit d'indiquer au système qu'il doit l'appliquer à chaque mise-à-jour du prix d'un produit.

0.3.3. Exemple de Modèle Sémantique Complet

Je vais donner ensuite un exemple de modèle sémantique d'une entreprise, complété par des règles sémantiques ad-hoc.

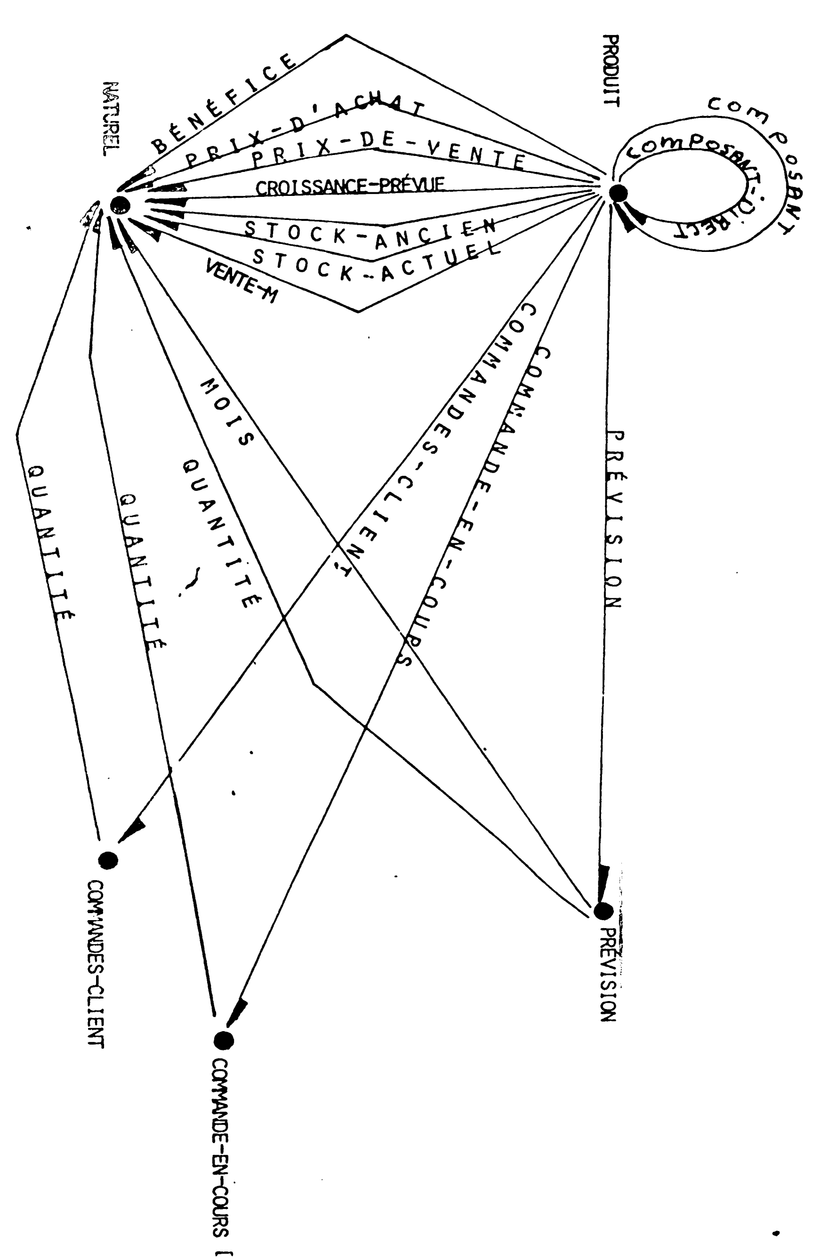
Ce modèle sera défini rigoureusement en 2.5.2.1. En page IV.64 je présente le résultat de l'implémentation que j'ai faite en SOCRATE-P [1]

Le modèle sémantique est celui que je présente dans la page 0.15 sous la forme d'un graphe marqué où les noeuds représentent les types d'objets, et :

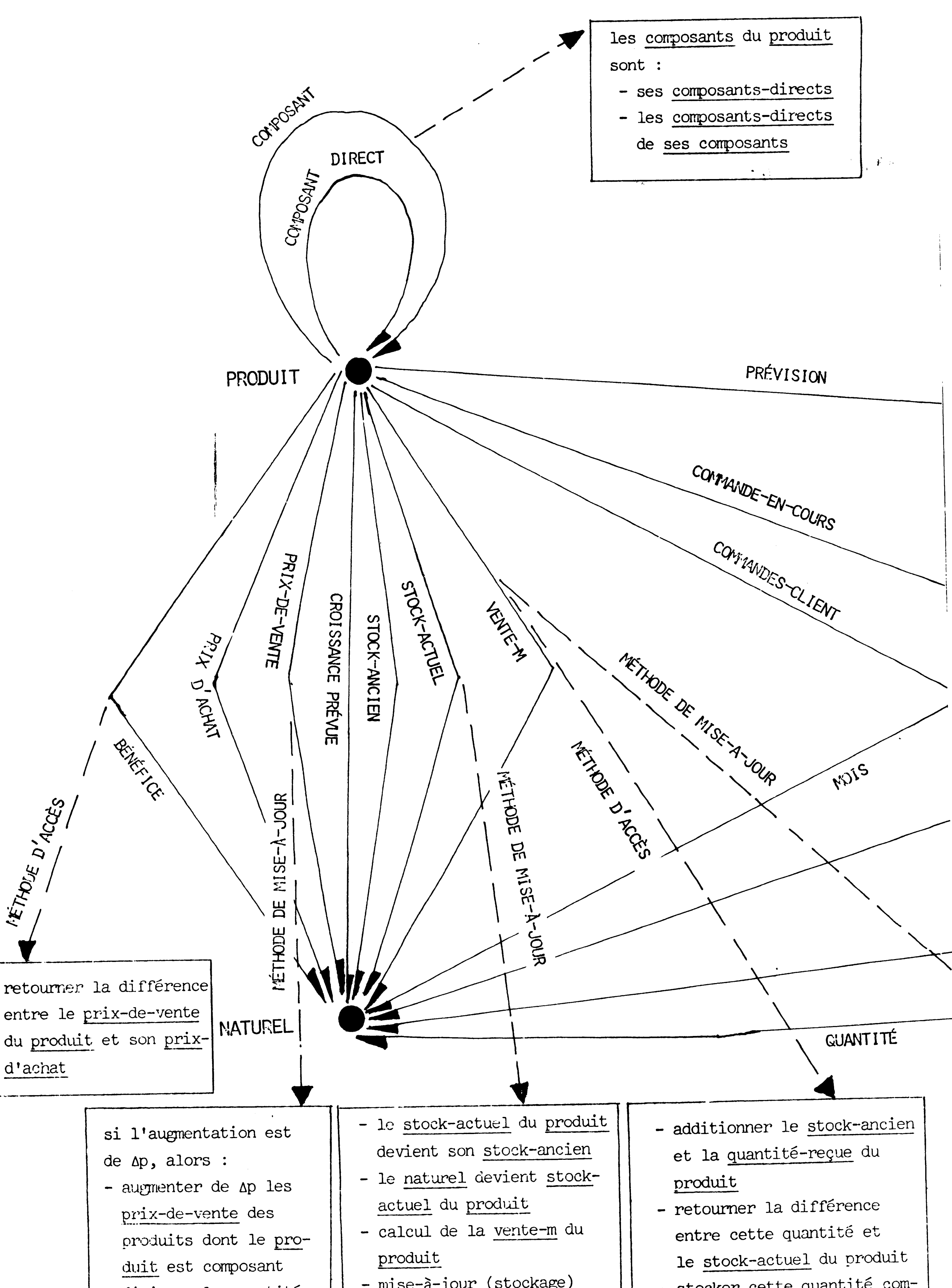


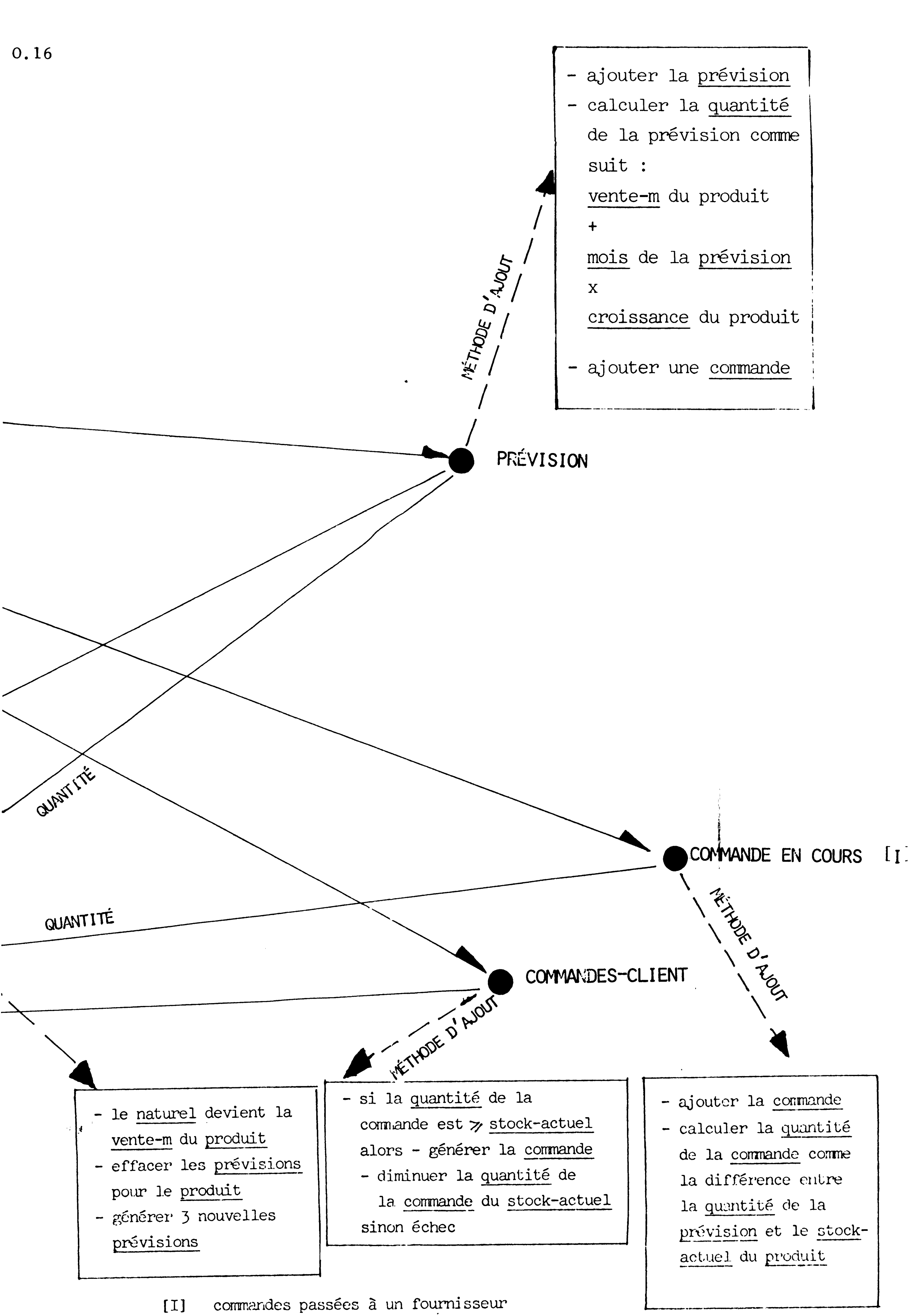
veut dire que R est une relation qui relie des objets qui sont forcément du type t_1 et du type t_2.

Dans la page 0.16, je présente le modèle sémantique complété par des règles sémantiques ad-hoc.



NATUREL
PRODUIT
COMMANDES-CLIENT
COMMANDE-EN-COURS [I]
PRÉVISION
BÉNÉFICE
PRIX-D'ACHAT
PRIX-DE-VENTE
CROISSANCE-PRÉVUE
STOCK-ANCIEN
STOCK-ACTUEL
VENTE-M
COMPOSANT
COMPOSANT-DIRECT
MOIS
QUANTITÉ
QUANTITÉ
QUANTITÉ
COMMANDES-CLIENT
COMMANDE-EN-COURS
PRÉVISION


les composants du produit
sont :
- ses composants-directs
- les composants-directs
de ses composants
COMPOSANT
COMPOSANT
DIRECT
PRODUIT
PRÉVISION
COMMANDE-EN-COURS
COMMANDES-CLIENT
PRIX-DE-VENTE
PRIX D'ACHAT
CROISSANCE PRÉVUE
STOCK-ANCIEN
STOCK-ACTUEL
VENTE-M
MÉTHODE DE MISE-À-JOUR
MÉTHODE D'ACCÈS
MOIS
BÉNÉFICE
MÉTHODE D'ACCÈS
MÉTHODE DE MISE-À-JOUR
MÉTHODE DE MISE-À-JOUR
NATUREL
QUANTITÉ
retourner la différence
entre le prix-de-vente
du produit et son prix-
d'achat
si l'augmentation est
de Δp, alors :
- augmenter de Δp les
prix-de-vente des
produits dont le pro-
duit est composant
- diminuer la quantité
des prévisions de F(Δp)
- le stock-actuel du produit
devient son stock-ancien
- le naturel devient stock-
actuel du produit
- calcul de la vente-m du
produit
- mise-à-jour (stockage)
de la vente-m calculée
- additionner le stock-ancien
et la quantité-reçue du
produit
- retourner la différence
entre cette quantité et
le stock-actuel du produit
- stocker cette quantité com-
me vente-m du produit



[I] commandes passées à un fournisseur

A côté des règles sur les composants (p. 0.11) et le bénéfice j'ai ajouté d'autres règles que j'ai trouvées dans l'ouvrage de MELES

- la mise-à-jour du stock-actuel d'un produit provoque le calcul de la vente du mois écoulé (vente-m) et celui-ci est suivi de celui des prévisions pour les 3 mois à venir. L'ajout des prévisions provoque le calcul de la commande-fournisseur et son ajout à la base de données.

0.3.4. Conclusion

Les règles sémantiques ad-hoc permettent de compléter le Modèle Sémantique des données de telle façon que le système peut se charger de déduire des données pour répondre à des questions, ou de garantir leur cohérence avec l'environnement réel.

La prise en charge par le système d'actions laissées habituellement à l'utilisateur est un concept introduit par de nouveaux langages d'Intelligence Artificielle [34] [35] [39] [40] [41].

Je terminerai la Partie 2 où je définis les Modèles Sémantiques Complets par l'étude comparative de la solution que je présente et de celle qu'on trouve du PLANNER [35].

C.4. ADÉQUATION DU LANGAGE-UTILISATEUR

Le Langage à travers lequel on utilise les données, doit être adapté aux nouvelles possibilités d'emploi que je propose (0.1) (0.3.4)

En effet, les langages actuels permettent à l'utilisateur de :

* initialiser et changer les données. D'un point de vue logique, cela consiste à engendrer des objets, établir des relations entre eux et changer ces relations.

* poser des questions aux données. Ces questions peuvent être de deux sortes :
  - demander si une relation est vraie entre des objets x,y,z, etc...

  ou
  - demander l'énumération de tous les objets, dans une relation donnée avec un objet déterminé x.

Tout d'abord, ces langages sont au plus équivalents au calcul de Prédicats du 1er ordre (I).

(I) SOCRATE ne l'est pas à cause des restrictions sur l'emploi des quantificateurs. Un exemple de langage équivalent est ALPHA [8]

Les études d'Intelligence Artificielle ou de "Semantic Information Retrieval" ont montré qu'en fait, le Calcul de Prédicats au 1er ordre ne suffit pas pour traiter convenablement la sémantique des environnements réels.

Il faudra que le langage soit au moins équivalent à un Calcul de Prédicats au 2nd ordre, ce qui oblige entre autres, à disposer d'un mécanisme plus général d'affectation des variables.

Ensuite, il faut que le Langage soit adapté à la déduction de données et à la planification (0.3.2). Ceci oblige à revoir complètement les notions sur lesquelles sont fondés les langages actues. Dès qu'il y a déduction et dès que les changements d'une donnée entraînent une série de changements en cascade, il faut pouvoir disposer de <u>plusieurs</u> méthodes pour déduire la donnée ou faire le changement. Le langage doit avoir un caractère "non-nécessairement algorithmique" qui va obliger à prévoir des mécanismes qu'on ne trouve pas dans les langages classiques et le rapporche des langages d'Intelligence Artificielle comme PLANNER [35], CONNIVER [40] et POPLER [39], par exemple.

La structure de contrôle doit être plus flexible (en particulier le pseudo-parallèlisme est nécessaire) et il faut pouvoir utiliser des espaces locaux similaires aux "data base contexts" de CONNIVER et QA4 [40] [41].

PARTIE 1

SEMANTIQUE DE SOCRATE

CE QUI MANQUE

1.0. INTRODUCTION

Dans cette partie, j'étudierai d'abord la structure des données SOCRATE [1.1] qui est le moyen par lequel l'utilisateur décrit la sémantique des données (les types d'objets et les relations pouvant exister entre eux) et organise la base de données. Ensuite, j'étudierai le langage d'interrogation et mise-à-jour [1.3].

Le choix de l'organisation des données et la description de leur sémantique étant tout à fait imbriqués, mon premier travail consistera à analyser quelle est la sémantique décrite dans une structure SOCRATE [1.2]. Par exemple, si la structure est :

entité personne

entité voiture

il faudra répondre à des questions comme :

Que veut-on dire en mettant le groupe répétitif 'voiture' à l'intérieur du groupe répétitif 'personne' ? Pourrait-on faire autrement ? [I]

Ou encore

La structure SOCRATE est basée sur des relations binaires (comme [15][16][17])et [18]) ou sur des relations de degré quelconque ? (comme [6][10] et [11])

En 1.2.3., je définirai des graphes marqués qui représentent toute la sémantique contenue dans les structures,et que j'appellerai modèles sémantiques.

Le modèle sémantique d'une structure est obtenu par application de l'algorithme décrit en 1.2.3.1 et 1.2.3.2 à la structure.

[I] On pourrait poser la même question à propos de IMS,CODASYL ou,dans un autre domaine,COBOL

En 1.3 je présente un formalisme pour décrire la sémantique du langage d'interrogation et mise-à-jour, en m'appuyant sur les modèles sémantiques.

Cette formalisation "a posteriori" de SOCRATE est nécessaire parce qu'on ne s'est pas préoccupé de rigueur logique au moment des spécifications [3][4], au contraire de ce qu'a fait, par exemple, E.F. CODD avec le RELATIONAL MODEL (CODD a étudié, entre autre, la sémantique du langage d'interrogation et mise-à-jour, et sa puissance) ou de SYNTEX (dans ce langage on a même abordé des problèmes de décidabilité ...).

Par ailleurs, la formalisation est loin d'avoir un intérêt purement théorique. En 1.4., je prouve qu'on peut l'employer pour résoudre des difficultés graves de l'utilisation de SOCRATE

Je donne d'abord un algorithme qui détermine si deux structures sont équivalentes (notion que je définirai rigoureusement en [1.4.1]). Ensuite un autre, décrit informellement, qui traduit une question ou mise-à-jour écrite en termes d'une structure, dans une question ou mise-à-jour écrite en termes d'une autre structure, équivalente à la première [1.4.1].

Ces deux algorithmes permettent de :

1 - changer de structure en cours d'application sans avoir à réécrire tous les programmes. [I]

2 - partager des programmes correspondant à deux applications SOCRATE structurées différemment (par exemple, deux applications dans un même réseau).

3 - employer une méthode rigoureuse pour choisir la structure [1.4.2]

4 - ajouter des redondances de façon systématique.

[I] propriété analysée dans l'introduction. (cf. § 0.2)

Enfin, d'un point de vue plus "théorique" je me servirai du formalisme développé pour déterminer et caracériser rigoureusement ce qui manque en SOCRATE [1.5] , tant du point de vue de la structure (règles sémantiques qu'on ne peut pas décrire avec elle) que du langage-utilisateur (pouvoir de sélection des données).

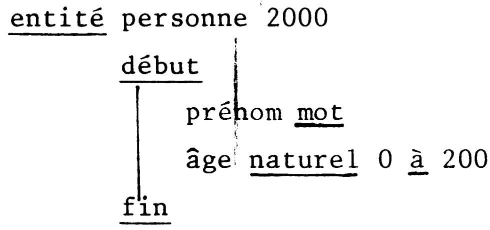
La constatation des limitations de SOCRATE conduit à un deuxième volet de la thèse qui est une proposition pour la définition de langages plus évolués que les actuels (Parties 2 et 3).

1.1. STRUCTURE SOCRATE

1.1.1. Exemples

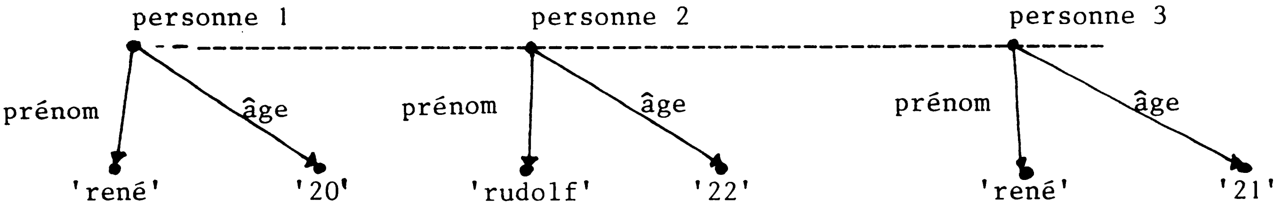
Commençons par étudier quelques exemples de structures SOCRATE (ou "définitions de structure SOCRATE"), pour bien comprendre l'ensemble de mécanismes dont dispose un utilisateur SOCRATE pour faire la description des données.

Un exemple très classique est le suivant :



```
entité personne 2000
       début
       |   prénom mot
       |   âge naturel 0 à 200
       fin
```

Il s'agit, dans la base de données, de stocker pour chaque personne son prénom et son âge. On peut engendrerjusqu'à 2000 personnes. Schématiquement, la base de données peut être représentée :



La structure veut dire :

- on peut engendrer et supprimer des personnes (les personnes sont des entités) (on ne peut engendrerplus de 2000 personnes).
- on peut stocker,pour chaque personne,déjà engendrée (existante) son prénom, qui a comme valeur forcément un mot de 10 caractères au plus.

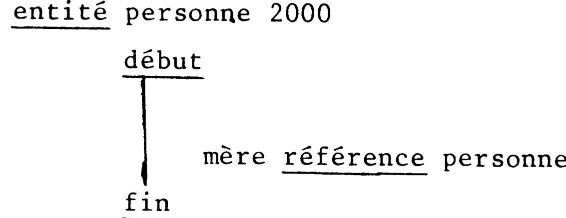
- on peut stocker pour chaque personne engendrée (existante) son âge, qui a comme valeur forcément un nombre naturel entre 0 et 200.

Souvent, dans une structure, on veut mettre en relation des entités . Pour mettre en relation deux entités ("deux enregistrements") on sait qu'il y a classiquement deux moyens

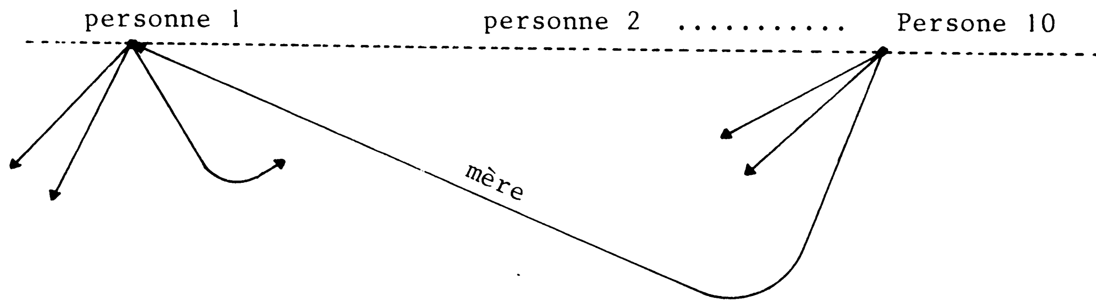
1 - la relation est implicite dans les champs

2 - la relation est explicite.

Dans SOCRATE, on utilise des pointeurs pour expliciter une relation entre deux entités.



Dans la base de données, cela donne, par exemple :



La 1ère personne est la mère de la 10ème personne.

FIGURE 2

Si une personne n'a qu'une mère, en général une personne peut avoir 0 ou plusieurs voitures. Les voitures sont des entités (comme les personnes). Cette relation entre personnes et voitures peut être définie par la structure :

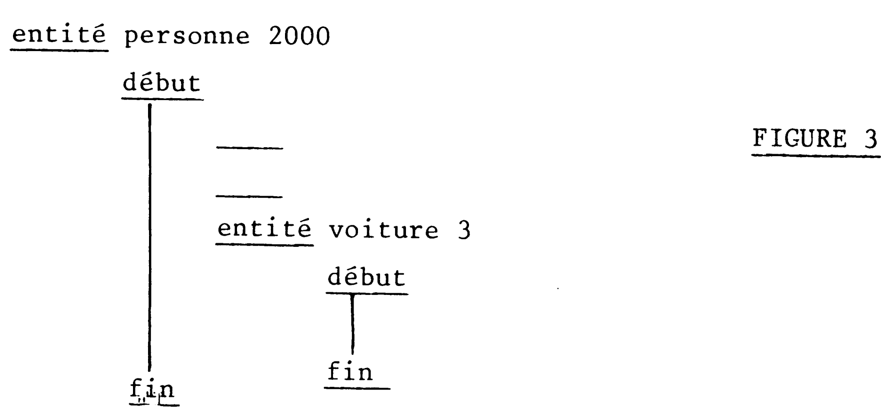


FIGURE 3

Les voitures sont un "groupe répétitif" dans les enregistrements des personnes. Pour chaque personne, il peut y avoir de 0 jusqu'à 3 voitures. La base de données peut avoir la disposition suivante :

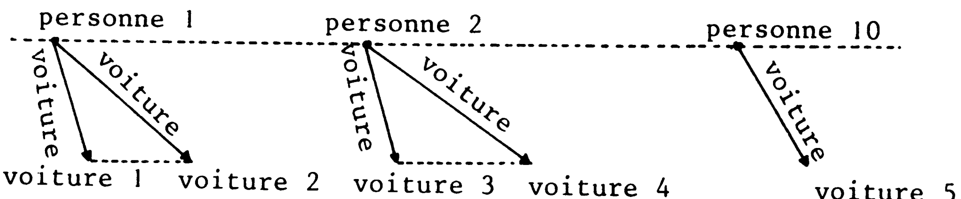


FIGURE 4

On aurait pu définir une autre structure équivalente (fig. 5)(le lecteur peut reconnaître intuitivement cette équivalence. Elle sera définie formellement en 1.4.1).

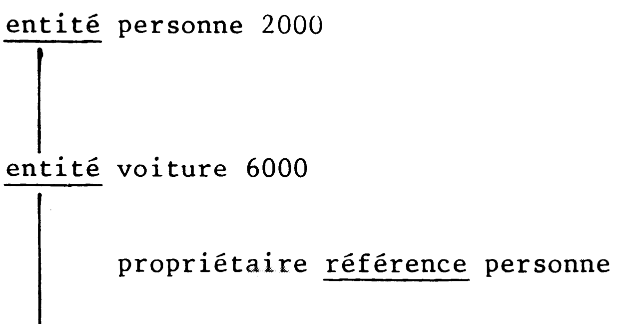


FIGURE 5

dans laquelle on emploie des pointeurs. La base de données aurait alors la disposition suivante :

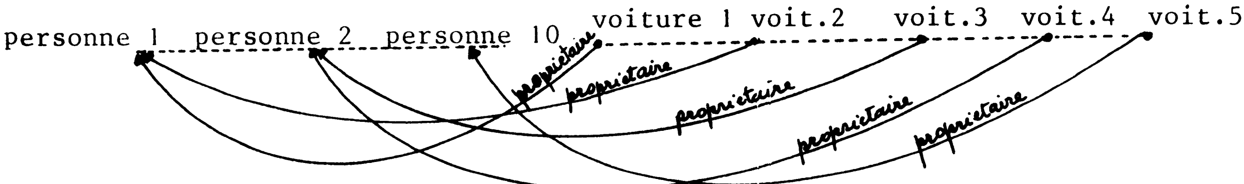


FIGURE 6

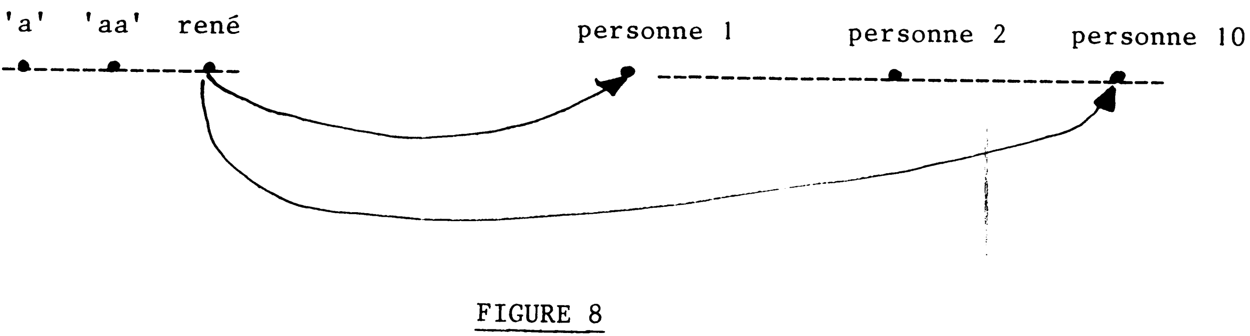
Avec les pointeurs, on exprime les <u>mêmes</u> relations entre les voitures et les personnes que dans la structure précédente, en mettant les voitures à l'intérieur des personnes.

Souvent, au lieu de stocker une relation, on stocke son inverse. Par exemple, au lieu de stocker, pour chaque personne, son prénom, on stocke, pour chaque mot, toutes les personnes qui ont ce mot comme prénom [I] . Dans ce cas-là, on peut utiliser un dictionnaire à adressage dispersé (appelé en SOCRATE "<u>dictionnaire d'accès rapide</u>")

```
entité personne 2000
       début
       |
       |     prénom mot rapide
       |
       fin
```

<u>FIGURE 7</u>

et les données seront disposées comme suit :



<u>FIGURE 8</u>

La relation stockée est l'inverse de la relation 'prénom', qui est celle qui est stockée quand l'organisation est celle de la page I.4.

[I] - On emploie cette organisation afin de rendre plus rapides les accès "quelles sont toutes les personnes qui ont le prénom x" ?

S'il y a des propriétés quantifiables comme le nom, l'âge, etc..., il y en a d'autres qui ne le sont pas : par exemple, la propriété d'être "français" : on est français ou on ne l'est pas (les français sont un sous-ensemble des personnes). Pour définir une telle propriété on fera :

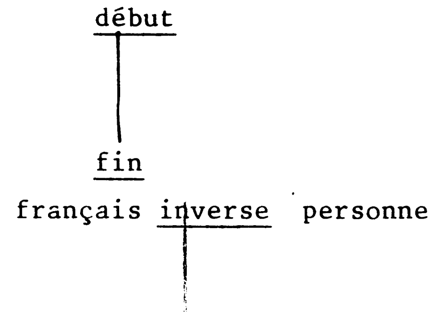


FIGURE 9

ce qui donne dans la base de données, par exemple :

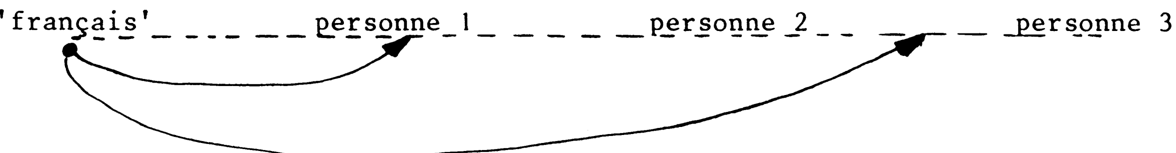


FIGURE 10

L'effet de cet inverse peut être obtenu avec la structure suivante :

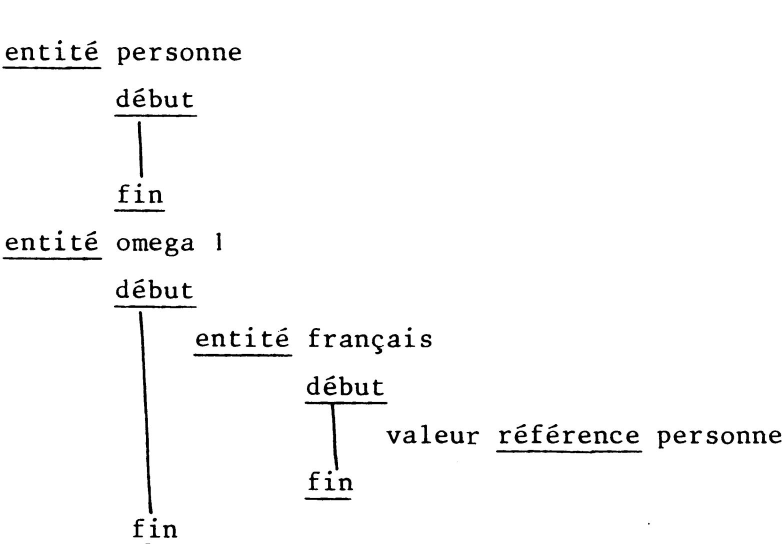


FIGURE 11

Naturellement, on peut utiliser les inverse's pour établir des relations entre entités, comme, par exemple, celle de filiation:

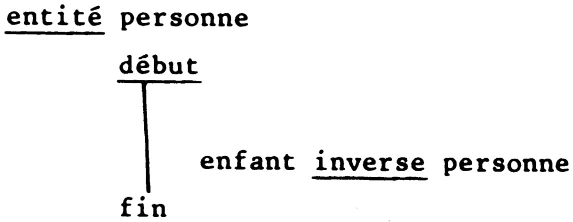


FIGURE 12

Une personne peut avoir 0 ou plusieurs enfants, alors que si on avait mis une référence à la place de l'inverse, une personne aurait toujours un enfant et jamais plus d'un enfant, ce qui serait une erreur sémantique. Les deux organisations ne sont pas équivalentes.

La base de données peut avoir une configuration :

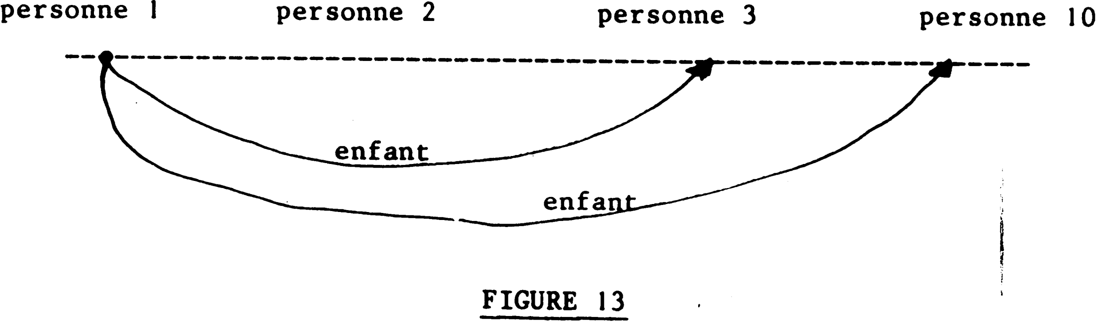


FIGURE 13

L'effet de l'inverse peut être obtenu avec la structure suivante :

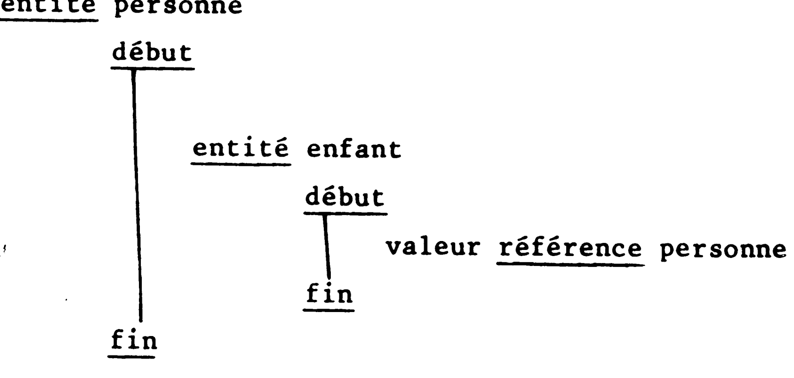
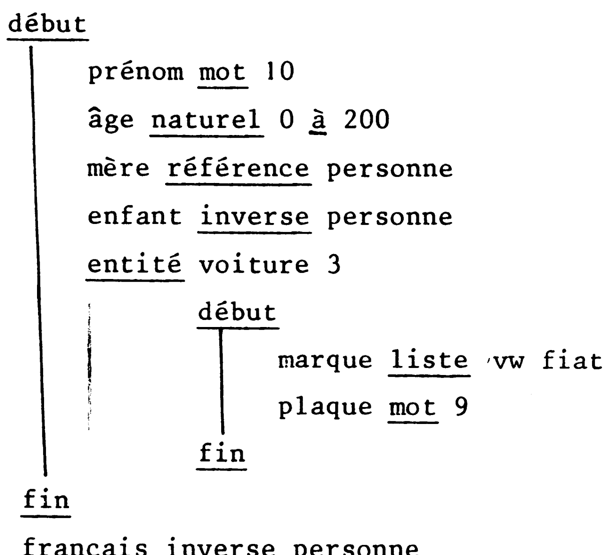


FIGURE 14

Enfin, l'utilisateur peut conjuguer les différentes possibilités de définition - ces différents mécanismes - pour obtenir une structure SOCRATE qui décrit les données. Par exemple :



```
entité personne 2000
    début
        prénom mot 10
        âge naturel 0 à 200
        mère référence personne
        enfant inverse personne
        entité voiture 3
            début
                marque liste vw fiat
                plaque mot 9
            fin
    fin
    français inverse personne
```

FIGURE 15

1.1.2. Objets et relations

Le lecteur peut, dès maintenant, constater que je perçois une base de données SOCRATE comme étant constituée par des objets (la 1ère personne, 2ème personne, ..., 'rudolf', 'rené', 20, etc...), et des relations binaires entre ces objets ('prénom', 'âge', 'mère', 'enfant', etc...)

Dans la suite du texte, j'utiliserai la notion d'objet avec le sens défini dans ce paragraphe, et j'entendrai par "données" à la fois les objets et les relations.

1.2. SÉMANTIQUE DE LA STRUCTURE SOCRATE

J'ai expliqué comment un utilisateur SOCRATE définit la structure et ce qu'elle implique en matière de disposition des objets et relations dans la base de données.

Je vais ensuite étudier, de façon systématique et formelle, la sémantique des structures SOCRATE.

1.2.1. Les "caractéristiques"

La structure SOCRATE peut être vue comme un ensemble de triplets disposés d'une certaine façon. Par exemple :

```
entité personne 2000
     début
     |  prénom mot 10
     |  âge naturel 0 à 200
     fin
```

FIGURE 16

contient 3 triplets :

```
entité personne 2000
prénom mot     10
age    naturel 0 à 200
```

Par la suite, je changerai la notation des "entités" comme suit : personne entité 2000.

Chaque triplet est appelé, en SOCRATE, une "caractéristique", et les formes qu'il peut prendre sont données dans le tableau de la figure 17.

<identificateur>	entité	<naturel>
<identificateur>	référence	<identificateur>
<identificateur>	inverse	<identificateur>
<identificateur>	mot	<naturel>
<identificateur>	texte	<naturel>
<identificateur>	naturel	<naturel>a
		<naturel>
<identificateur>	liste	<identificateur>
		<identificateur>
		etc.

FIGURE 17

La sémantique d'une "caractérisitque" est fonction de sa deuxième composante.

ENTITES

Les caractéristiques entité comme, par exemple :

personne entité 2000

servent à nommer un prédicat (propriété d'objets) dont l'extension sera remplie par la suite en engendrant ses éléments un à un. Ceux-ci (les personnes dans l'ex.) seront codifiés en tant qu'entités, c'est-à-dire que chacun aura un indice (personne 1, personne 2, ..., personne 10, etc...), à partir duquel sera calculée son adresse virtuelle, c'est-à-dire l'adresse de l'espace virtuel (au sens SOCRATE) où se trouvent les relations qui le concernent (prénom, âge, etc...).

J'appellerai catégories les prédicats qui, comme celui-ci, sont déclarés dans la structure. Je pourraisd'ailleurs les appeler 'types' puisque les catégories jouent en SOCRATE le rôle que jouent les types en langages de programmation. Il y a cependant des différences entre les deux concepts. En

particulier les catégories devront être déclarées alors que les types sont en général donnés a priori.

ENTITES IMBRIQUEES

Quand une entité est à l'intérieur d'une autre, elle déclare à la fois :

:: une catégorie

et

:: une relation binaire pouvant relier des objets de cette catégorie aux objets de la catégorie déclarée par l'autre entité.

Par exemple, dans la structure :

```
entité personne 2000
    début
    |
    |   entité voiture 3
    |          début
    |          |
    |          fin
    fin
```

FIGURE 18

la caractéristique 'voiture entité 3' déclare :

:: une catégorie appelée 'voiture'

:: une relation appelée aussi 'voiture' pouvant relier des personnes et des voitures (éléments de la catégorie 'voiture')

Bien que les noms soient identiques, il ne faut pas confondre la catégorie avec la relation.

On peut représenter ainsi cette sémantique.

(personne, entité,2000) o———— voiture ————► o (voiture,entité,6000) (I)

(I) - Le cardinal maximum d'une entité qui est à l'intérieur d'une autre est le produit des 3èmes composantes des deux entités.

Intuitivement, ceci veut dire : "la voiture d'une personne doit être une voiture".

REFERENCES

Regardons maintenant les caractéristiques qui ne sont pas des "entités".

Les caractéristiques référence comme, par exemple, (page I.5)

mère référence personne

servent à déclarer une relation binaire dont le nom est la première composante ('mère'), relation devant relier des objets de la catégorie déclarée par la caractéristique entité à l'intérieur de laquelle elle se trouve (personne), et des objets de la catégorie dont le nom est indiqué dans la troisième composante (dans l'exemple, cette catégorie est aussi 'personne'),

Cette sémantique peut se représenter ainsi :

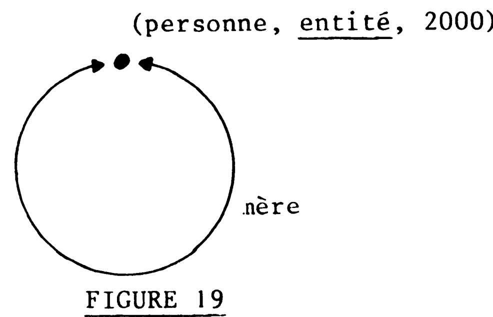
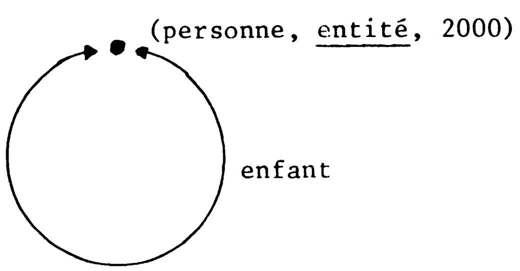


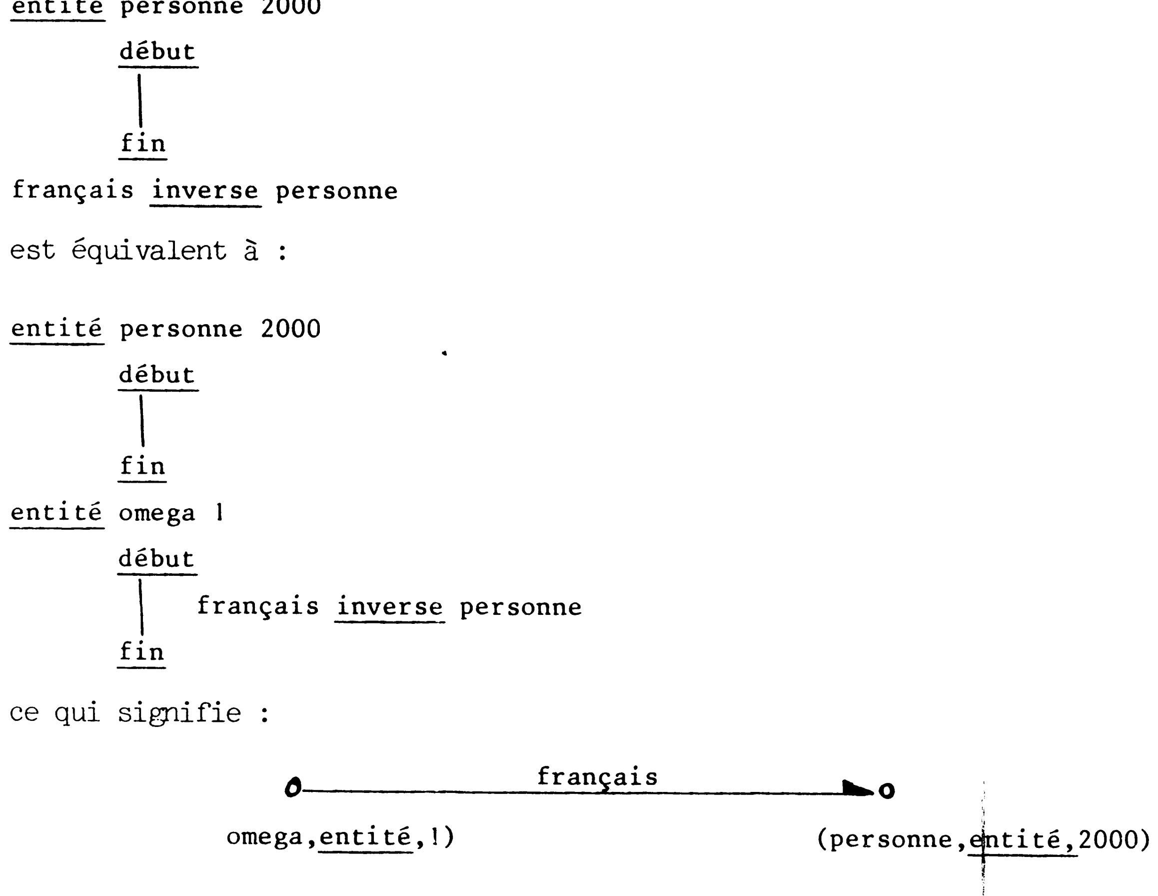
FIGURE 19

INVERSES

Une caractéristique inverse est la déclaration d'une relation de façon analogue aux caractéristiques référence. Dans l'exemple de la page I.8 (figure 11) la relation s'appelle 'enfant' et on peut la représenter ainsi



Une caractéristique inverse peut ne pas être à l'intérieur d'une entité. Tout se passe alors comme si elle était à l'intérieur d'une entité fictive selon ce que l'on a vu en page I.8 . Par exemple



Intuitivement, ceci veut dire "les français sont des personnes".

De telles caractéristiques inverses déclarent des relations unaires (propriétés).

MOTS, TEXTES, NATURELS, LISTES

En ce qui concerne toutes les caractéristiques des autres types (mot, texte, naturel et liste) chacune représente à la fois :

la déclaration d'une catégorie

et

la déclaration d'une relation dont le nom est sa première composante et qui peut relier des objets de la catégorie déclarée par la caractéristique entité à l'intérieur de laquelle elle est définie,et cette deuxième.

Par exemple, dans la structure de la page I.11

prénom mot 10

déclare la catégorie (,mot,10) et la relation prénom qui relie des personnes à des éléments de cette catégorie qui sont des mots de 10 caractères maximum (figure 1). On peut représenter cette sémantique comme suit :

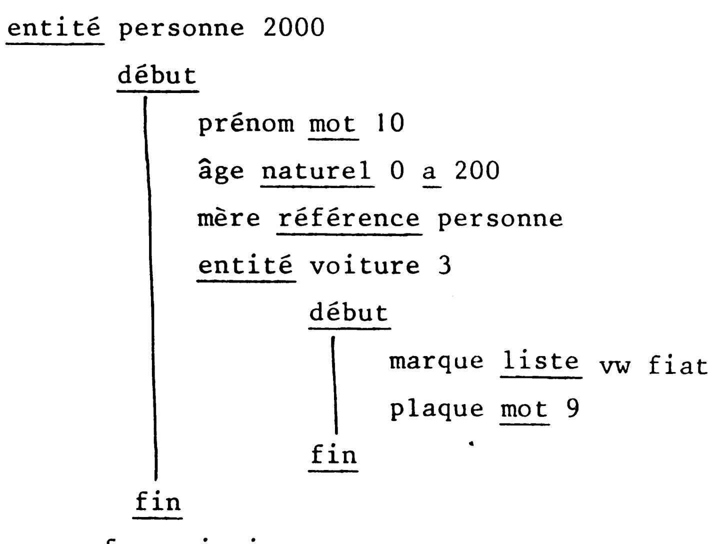
(personne,entité,2000)

o————————prénom————————▶o (,mot,10)

CONCLUSION

En conclusion, les "caractéristiques" SOCRATE signifient des déclarations de catégories d'objets et de relations pouvant relier ces objets, sémantique qu'on peut représenter dans un graphe marqué.

Par exemple, pour la structure :



```
entité personne 2000
      début
         prénom mot 10
         âge naturel 0 a 200
         mère référence personne
         entité voiture 3
               début
                  marque liste vw fiat
                  plaque mot 9
               fin
      fin
   français inverse personne
```

le graphe est :

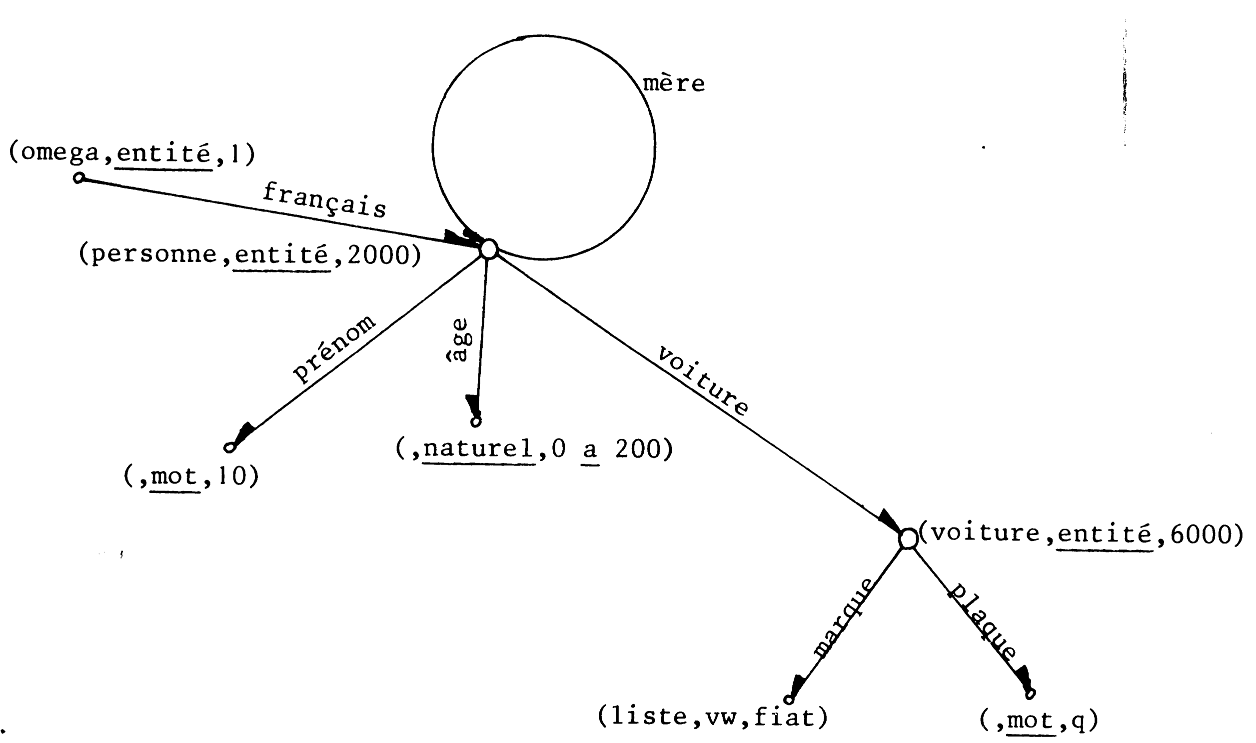


FIGURE 20

Dans ce graphe, une arête

$$C \circ \xrightarrow{\quad R \quad} \circ C'$$

signifie :

$$\forall x \forall y \{x \; R \; y \rightarrow (x \in C) \wedge (y \in C')\}$$

ce qui peut se formuler intuitivement par "la relation R ne peut relier que deux objets des catégories C et C' ".

1.2.2. Implantation des relations dans l'espace virtuel : I-relations

Le graphe que j'ai donné (figure 20) représente toute la sémantique explicitement décrite dans la structure.

Cependant, la manière dont les relations sont implantées dans l'espace virtuel, qui découle de la disposition relative des caractéristiques (figure 15) contient implicitement des règles sémantiques très importantes qu'il faut pouvoir représenter dans le graphe. J'explique.

On pourrait penser que, d'un point de vue simplement sémantique, il serait indifférent de définir la structure de la figure 15 ou alors

entité voiture 6000

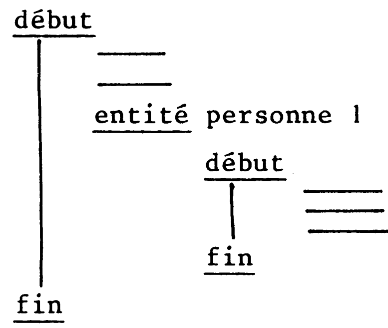


FIGURE 21

ou encore la structure :

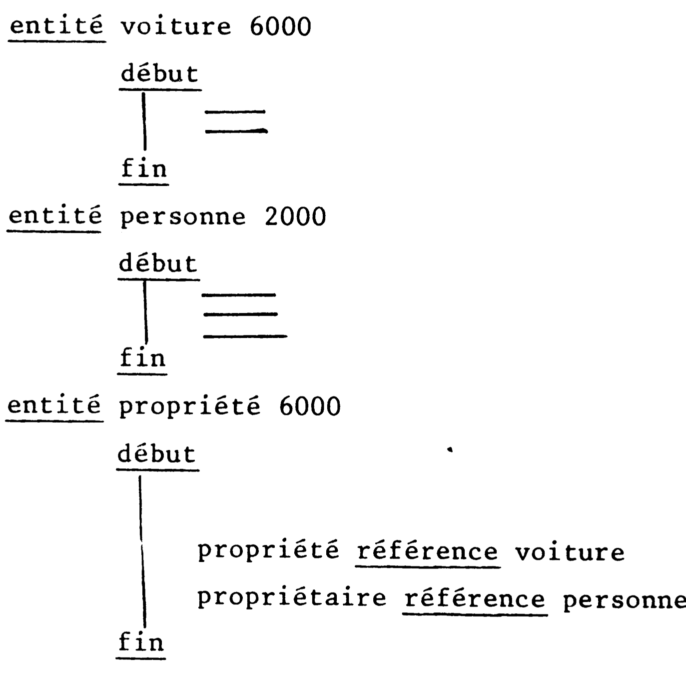


FIGURE 22

à condition, évidemment, que la relation 'voiture' des deux premières structures (figure 15 et 21) veuille dire, pour l'utilisateur, la même chose que la relation 'propriété' de la structure de la figure 22, c'est-à-dire "voiture-qui-est-propriété-de".

Cependant, ces trois structures décrivent des sémantiques très différentes.

C'est ainsi qu'à la première (figure 15) correspond l'implantation suivante dans l'espace virtuel :

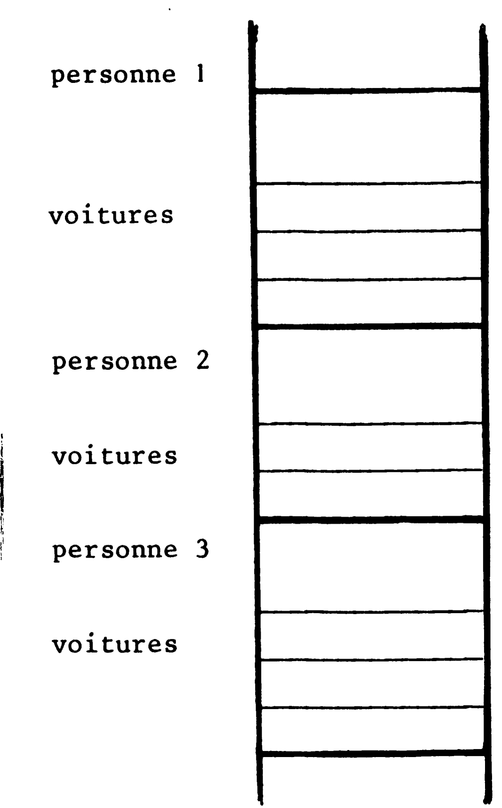


FIGURE 23

ce qui entraîne les règles sémantiques suivantes :

R1 - pour chaque personne, il peut y avoir 0 ou plusieurs voitures qui sont ses voitures

R2 - pour chaque voiture, il y a toujours une personne et une seule dont elle est la voiture (qui est son propriétaire).

Par contre, à la deuxième structure (figure 21) correspond l'implantation :

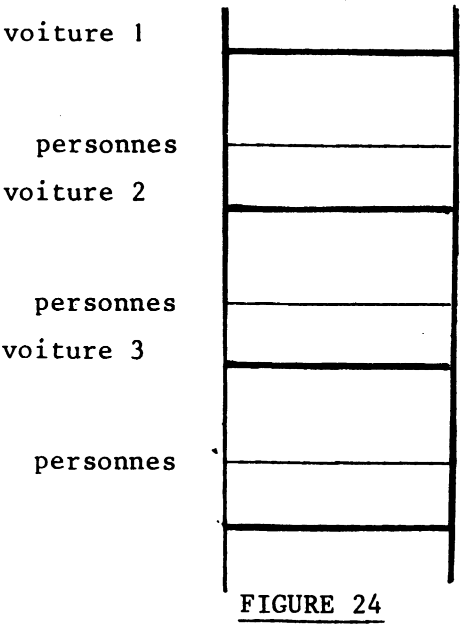


FIGURE 24

qui contient implicitement les règles :

R'1 - pour chaque personne,il y a toujours une et une seule voiture qui est sa voiture

R'2 - pour chaque voiture,il peut y avoir 0 ou alors une personne dont elle est la voiture.

Enfin, à la troisième structure (figure 22) correspond l'implantation :

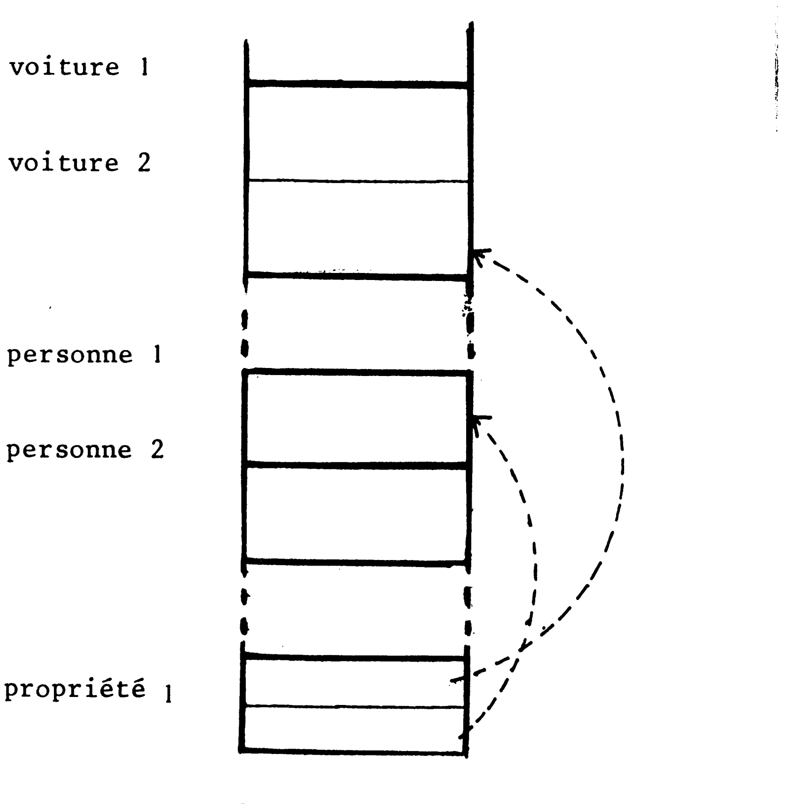


FIGURE 25

qui contient implicitement des règles encore différentes :

R"1 - pour chaque personne, il peut y avoir 0 ou plusieurs voitures qui sont ses voitures

R"2 - pour chaque voiture, il peut y avoir 0 ou plusieurs personnes dont elle est la voiture.

Les trois couples de règles sémantiques (R1, R2, R'1, R'2, R"1, R"2) sont écrits en français et, pour que je leur donne une notation formelle, il faut faire une brève parenthèse logique.

1.2.2.1. Fonctions d'accès - Prédicats Fx

D'un point de vue mathématique, on sait que toute relation binaire R a une relation inverse R^{-1} : si xRy est vrai alors $yR^{-1}x$ est vrai aussi, et inversement.

En bases de données, les relations inverses sont très importantes. Les règles sémantiques qu'on vient d'étudier le prouvent puisque R2, R'2 et R"2 concernent la relation inverse de la relation 'voiture'.
Aussi, au niveau du langage-utilisateur, on emploie souvent les relations inverses.

J'appellerai fonctions d'accès les deux relations inverses l'une de l'autre.
Par exemple, 'voiture' et 'voiture^{-1}' (on pourrait dire 'propriétaire') sont deux fonctions d'accès.

Une fonction d'accès est, d'un point de vue mathématique, une fonction qui, à un objet d'une catégorie C (son domaine) fait correspondre un ensemble d'objets d'une catégorie C' (son co-domaine, qui est éventuellement la même catégorie). Ainsi, par exemple, si les voitures de la première personne sont la troisième et la cinquième et si celle de la deuxième personne est la quatrième, la fonction d'accès 'voiture' est celle qu'on peut représenter symboliquement par la figure 26.

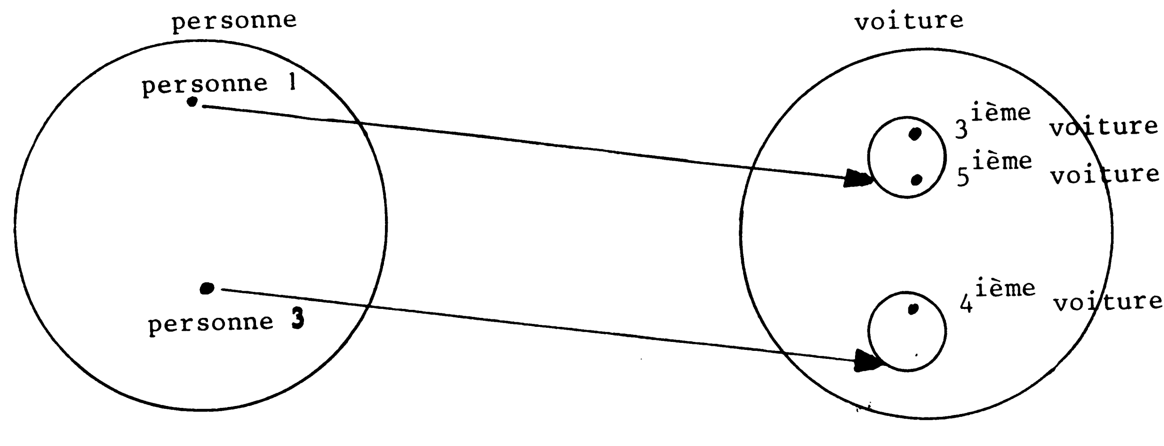


FIGURE 26

Les deux ensembles de voitures peuvent se dénoter comme suit

voiture [personne 1]

voiture [personne 3]

expressions qui représentent les noms de deux prédicats, dont les extensions sont les deux ensembles en question (figure 26).

J'appellerai les prédicats de ce genre des prédicats Fx, pour les distinguer des prédicats déclarés dans la structure que j'ai appelés des catégories (1.2.1.).

Fermons la parenthèse logique et servons-nous de ces notations pour exprimer formellement les règles R1, R2, R'1,R'2,R"1 et R"2.

Je dénoterai "card(P)" le cardinal de l'extension d'un prédicat P quelconque.

Les règles peuvent se formaliser ainsi :

R1 - $\forall x \in$ personne $\{0 \leqslant \text{card}(\text{voiture}[x]) \leqslant \infty\}$

R2 - $\forall x \in$ voiture $\{1 \leqslant \text{card}(\text{voiture}^{-1}[x]) \leqslant 1\}$

ou, plus succinctement,

R1 - $\underline{0\infty}$ (voiture) que je noterai $\underline{0*}$ (voiture)

R2 - $\underline{11}$ (voiture^{-1})

qui veulent dire : la fonction d'accès 'voiture' a la propriété d'être <u>0*</u> (dit autrement : "est de type <u>0*</u>) et la fonction 'voiture^{-1}' a la propriété d'être <u>11</u>.

Avec cette notation, les autres couples de règles ont la forme :

R'1 <u>11</u> (voiture)
R'2 <u>01</u> (voiture^{-1})

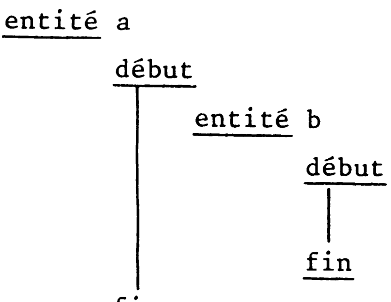
R"1 <u>0*</u> (voiture)
R"2 <u>0*</u> (voiture^{-1})

Chaque couple de règles sémantiques énonce donc les propriétés de la fonction d'accès 'voiture' et de son inverse : 'voiture^{-1}'.

I-relations et I-liens

Je vais exposer ensuite une notation formelle qui permettra de caractériser les implantations dans l'espace virtuel.

Dans le cas de deux entités imbriquées :



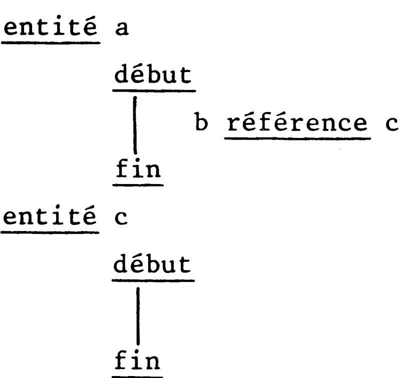
je dirai que la fonction d'accès concernée, dont le nom est 'b' est implantée par la I-relation [I] P, (P pour "père"). En effet, le fait que b est à l'intérieur de a peut être vu aussi comme "a est-père-de b" ce qui est en accord avec le langage utilisé dans la littérature pour caractériser les descriptions de données (cf. "modèles hiérarchiques").

Formellement, je dirai qu'une fonction d'accès est implantée par P quand son co-domaine est déclaré à l'intérieur du bloc correspondant à son domaine.

Pour ces mêmes raisons, dans le cas d'une caractéristique de type mot texte naturel ou liste définie à l'intérieur d'une entité, je dirai que la fonction d'accès correspondante (page I.15) est implantée par P.

[I] I-relation est une abréviation de "relation d'implantation"

Dans le cas des références comme:



je dirai que la fonction d'accès (b) est implantée par **la** I-relation R (R pour "référence") parce que son co-domaine n'est pas déclaré à l'intérieur du bloc correspondant à son domaine.

Si on se sert de ces notations, on peut caractériser formellement les 3 implantations correspondantes aux 3 structures des figures 15, 21 et 22, qui sont représentées dans les figures 23, 24 et 25.

L'implantation de la figure 23 peut se caractériser ainsi : "la fonction d'accès 'voiture' est implantée par la I-relation P".

L'implantation de la figure 24 peut se caractériser ainsi : "la fonction d'accès 'voiture' est implantée par la I-relation P^{-1}"

Enfin, l'implantation de la figure 25 est plus difficile à dénoter parce que la fonction d'accès 'voiture' n'est plus implantée par une seule I-relation mais par <u>deux</u> I-relations (R et R^{-1}) et une caractéristique intermédiaire appelée 'propriété' (figures 22 et 25).

Je dirai alors que 'voiture' est implantée par <u>le I-lien</u>

R^{-1} (propriété,<u>entité</u>,6000) R

On peut considérer que 'voiture' est la composition de deux <u>pseudo-fonctions d'accès</u>, notées respectivement 'propriétaire^{-1}' et 'propriété' et implantées chacune par une I-relation (R^{-1} et R respectivement). La

caractéristique intermédiaire (dans l'exemple,c'est 'propriété') sera dite pseudo-catégorie.

1.2.2.2. Méthode pour déterminer les propriétés d'une fonction d'accès

Quand une fonction d'accès F,dont le domaine est C_1 et le co-domaine C_2, est implantée par une I-relation i, les propriétés de F sont fonction de i et des types de C_1 et C_2 (le type est représenté par la deuxième composante de la caractéristique dans laquelle la catégorie est définie, par exemple entité). Le tableau de la figure 27 donne les propriétés d'une fonction d'accès (3ième colonne) en fonction de i (première colonne) et des types de C_1 et C_2. Je noterai cette fonction par φ .

Quand une fonction d'accès est implantée par un I-lien, alors il faut déterminer les propriétés de chacune des pseudo-fonctions d'accès (page) qui la composent, ce qu'on fait par la méthode vue et enfin appliquer itérativement à ces propriétés les règles du tableau de la figure 28 tableau qui représente une fonction que je noterai Ψ.

i (I-relation)	Types des catégories C_1 et C_2	$\varphi(C_1, i, C_2)$
	type de C_1 : (E,-) type de C_2 : (E,-)	$C_1 \xrightarrow{0*} C_2$
$C_1 \xrightarrow{P} C_2$ (C_1 est père de C_2)	type de C_1 : (E,-) type de C_2 : (E,1)	$C_1 \xrightarrow{01} C_2$
	type de C_1 : (E,-) type de C_2 : (t,-) avec $t \neq E$	$C_1 \xrightarrow{11} C_2$
	type de C_1 : (E,-) type de C_2 : (E,-)	$C_1 \xrightarrow{11} C_2$
$C_1 \xrightarrow{R} C_2$ (C_1 fait référence à C_2)		
	type de C_1 : (t,-) avec $t \neq$ ent. type de C_2 : (E,-)	$C_1 \xrightarrow{0*} C_2$

$C_1 \xrightarrow{P^{-1}} C_2$ (C$_1$ <u>est fils de</u> C_2)	type de C_1 : (t,-) avec t≠entité et discriminant type de C_2 : (E,-)	$C_1 \xrightarrow{01} C_2$
	type de C_1 : (E,-) type de C_2 : (E,-)	$C_1 \xrightarrow{11} C_2$
$C_1 \xrightarrow{R^{-1}} C_2$ (C_1 <u>est référencé par</u> C_2)	type de C_1 : (E,-) type de C_2 : (E,-)	$C_1 \xrightarrow{0*} C_2$
	type de C_1 : (E,-) type de C_2 : (E,1)	$C_1 \xrightarrow{01} C_2$

Fonction Ψ

$C_1 \xrightarrow{r_1} C_2 \xrightarrow{r_2} C_3$		$C_1 \xrightarrow{\psi(r_1, r_2)} C_3$
0*	0*	0*
0*	01	0*
0*	11	0*
01	0*	0*
01	01	01
01	11	01
11	0*	0*
11	01	01
11	11	11

FIGURE 28

1.2.3. Modèle sémantique

Je donne maintenant un algorithme général capable de déterminer, à partir d'une structure SOCRATE quelconque, un graphe où est représentée toute la sémantique que cette structure contient : catégories, fonctions d'accès et propriétés des fonctions d'accès.

L'algorithme est construit en deux étapes :

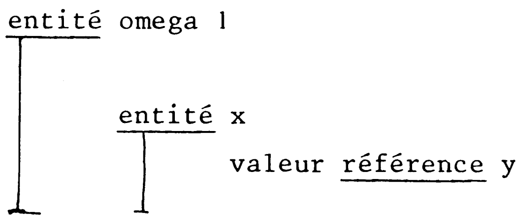
1 - obtention d'un graphe marqué-I-graphe, où sont représentées les catégories, pseudo-catégories (page I.27), fonctions d'accès, pseudo-fonctions d'accès (page I.26) et les I-relations.

2 - élimination des pseudo-catégories et pseudo-fonctions-d'accès avec calcul des propriétés des fonctions d'accès, selon la méthode décrite en 1.2.2.3.

Le graphe ainsi obtenu sera appelé le modèle sémantique.

1) Obtention du I-graphe

Remplacer chaque caractéristique inverse qui n'a pas de père (cf. page I.25) par :



si l'inverse est: "x inverse y"

Tous les autres "inverses" seront remplacés par :

entité x

valeur référence y

Pour toute entité E sans père (cf. page I.25)

faire :

- créer un noeud n_E et le marquer avec
  (x, <u>entité</u>, y)
  où x est le nom de E
  et y est le nombre maximum de réalisations de E
- pour toute caractéristique C fille de E

  faire :

  si C n'est pas <u>référence</u>

  alors - créer un noeud n_C

  - si C n'est pas <u>entité</u>

    alors - marquer n_C avec (x,y,z)
    où x est le mot vide
    y est la deuxième composante de la caractéristique
    z est tout ce qui suit y
    - relier n_E à n_C par une arête marquée (x,P)
      où x est le nom de C
    - relier n_C à n_E par une arête marquée (x^{-1},P^{-1})
      où x a la même signification

    sinon - relier n_E à n_C par une arête marquée (x,P)
    où x est le nom de C
    - Tout recommencer pour cette entité C

    fin

  sinon - relier n_E au noeud n marqué(x,y,z) où
  x est le mot qui suit le mot '<u>référence</u>' dans la caractéristique.

  fin

fin

fin

Exemple :

Le I-graphe de la structure de la figure 15 est le suivant :

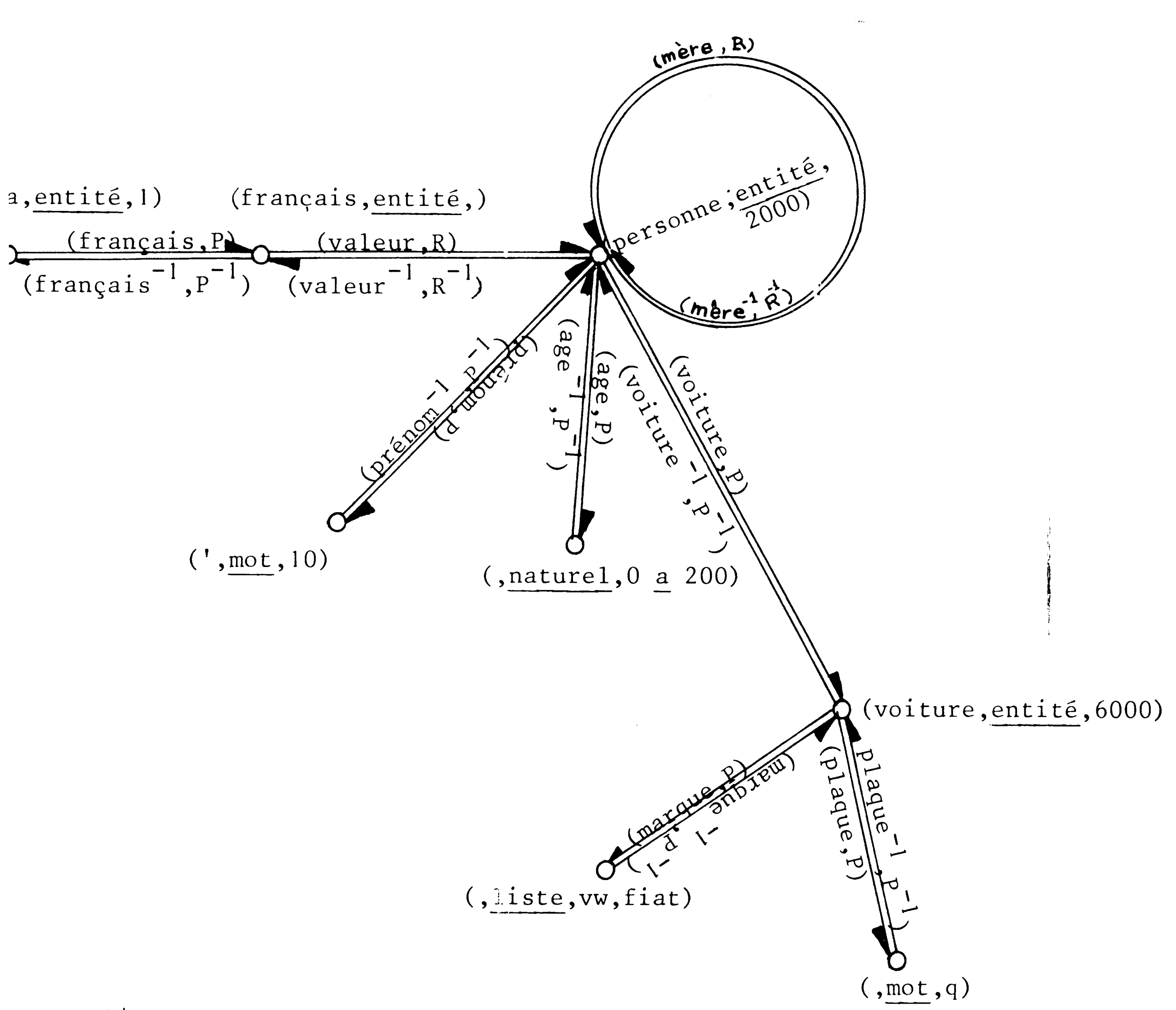


FIGURE 29

2) Obtention du Modèle Sémantique

Enfin, pour obtenir le Modèle Sémantique, il faut exécuter l'algorithme suivant, sur le I-graphe :

- marquer avec 'pc' (pseudo-catégorie) tous les noeuds η qui vérifient les deux conditions :

  a) η est marqué avec (x,entité,y) avec x,y quelconques

  b) le nombre d'arêtes partant de n et marquées avec (x,R), x étant quelconque, est 1.

- soit nulle une propriété de fonctions d'accès telle que $\bar{\Psi}$(x,nulle) = x pour toute propriété x

- pour tout noeud x telque x n'est pas marqué avec pc

```
|
|        élimination (x,x,nulle)
fin
```

où "élimination" est la procédure suivante :

```
élimination
   | pour toute arête x3 partant du noeud x1
   |     | x4 := noeud d'arrivée de x3
   |     | x5 := propriété (x1,x3,x4,x2)
   |     | si x4 est pseudo-catégorie
   |     | alors élimination (x1 x4 x5)
   |     | sinon création-de-relation (x1 x4 x5)
   |     | fin
   |  fin
fproc
```

où "propriété" est la procédure suivante :

```
propriété x1 x2 x3 x4
|    x5 := φ(x1 x2 x3)        [I]
|    x6 := Ψ(x5,x4)           [I]
|    return (x6)
fproc
```

Exemple :

Le Modèle Sémantique de la structure de la figure 25 est :

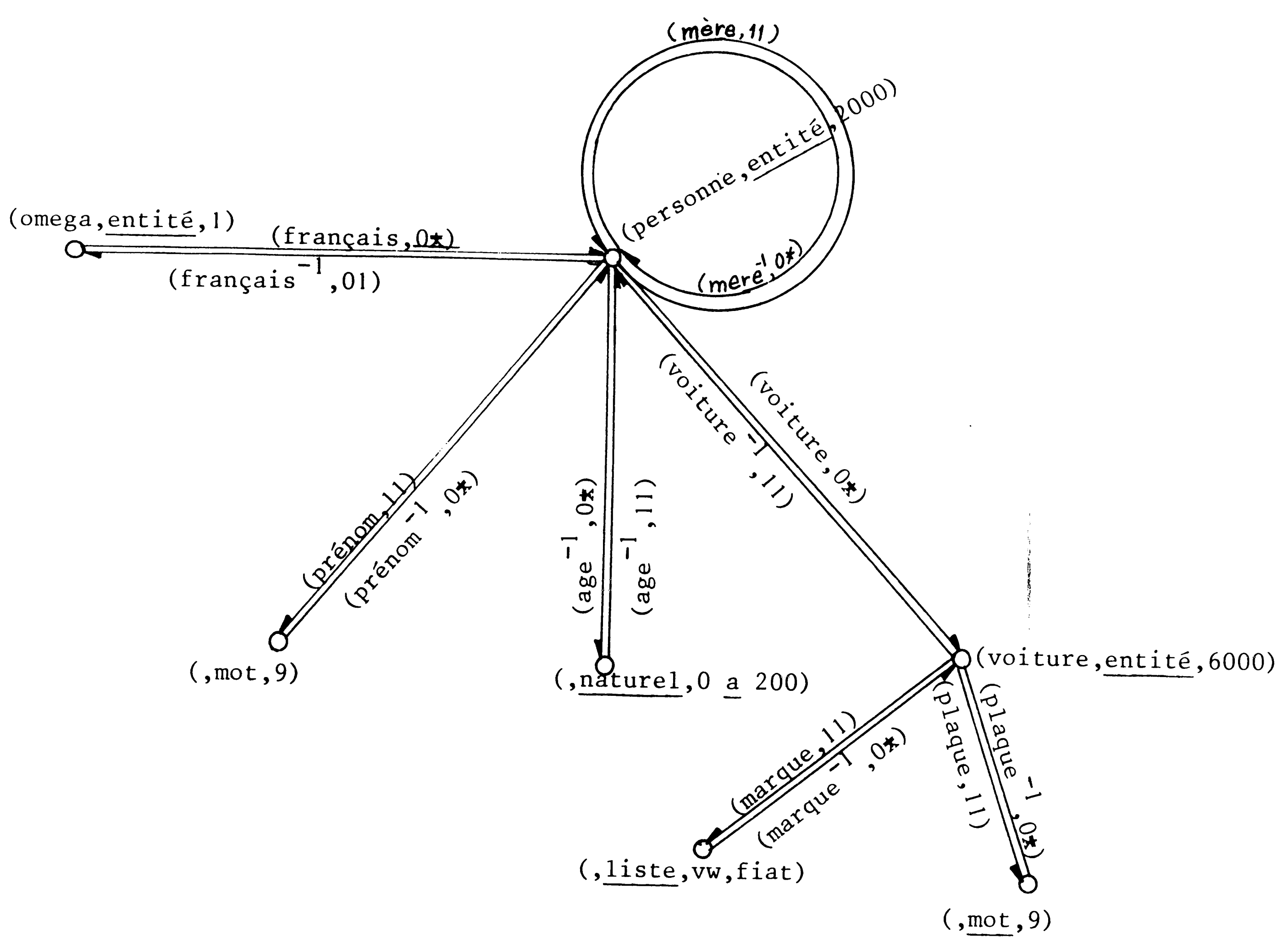


FIGURE 30

[I] φ et Ψ sont les deux fonctions définies par les tableaux respectivement de figure 27 et de figure 28 : $\varphi(c_1, i, c_2)$ est la propriété d'une fonction dont le domaine est c_1, le co-domaine c_2 et implantée par la I-relation i. $\Psi(p_1,p_2)$ est la propriété d'une fonction d'accès composée de deux pseudo-fonctions-d'accès ayant les propriétés p_1 et p_2.

1.3. Sémantique du langage d'interrogation et mise-à-jour

Dans une application de SOCRATE, après avoir défini la structure, l'utilisateur interroge et met à jour la base de données.

Par exemple, il peut faire exécuter :

i mère de personne 1

c'est-à-dire qu'il demande l'impression de la mère de la personne 1, et

m prénom de personne 1 = rudolf

ce qui a pour effet que le prénom de la personne 1 devient "rudolf".

S'il veut demander, par exemple, toutes les personnes de prénom 'rené' et d'âge > 20 il fera :

i personne ayant prénom = rené et âge > 20 [I]

et s'il veut connaître les mères de toutes les personnes de prénom 'rené' et âge > 20 il fera :

i mère de personne ayant prénom = rené et âge > 20

Le langage d'interrogation et mise-à-jour a deux propriétés qu'il faut noter :

1 - il dépend de ce qui est déclaré dans la structure. (par exemple : on ne peut écrire "i mère de personne 1" que parce que dans la structure figurent les deux caractéristiques : 'mère' et 'personne' (figure 15).

2 - les opérateurs utilisés (de, ayant, etc...) sont fonction de la disposition relative des caractéristiques (dans l'exemple, on écrit "mère de personne 1" parce que la caractéristique 'mère' est fille de la caractéristique 'personne' (figure 15).

Avant de pouvoir décrire la sémantique d'une question ou mise-à-jour il faut développer un peu plus les considérations logiques que j'ai faites en page

Si deux objets x et y sont dans une relation R, dont les deux fonctions d'accès sont F et F^{-1}, alors les expressions suivantes sont équivalentes

---

[I] La formulation correcte serait "i tout personne ayant prénom = rené et age > 20" pour distinguer de "i un personne ..."
Je ne considèrerai pas cette possibilité et j'écrirai "i personne ..." au lieu "i tout personne ..."

$$x \ R \ y$$
$$x \in F[y]$$
$$y \in F^{-1}[x]$$

Puisque F y est le nom d'un prédicat (propriété d'être F y), l'expression

$$x \in F[y]$$

veut dire : x appartient à l'extension du prédicat F[y] (formellement, F[y] est vrai pour x).

Je donne un exemple : soit la relation "mariage" qu'on représente par les deux fonctions d'accès 'mari' et 'femme'. Le fait qu'il y ait la relation mariage entre 'jean' et 'marie' peut s'exprimer ainsi :

jean ∈ mari[marie]

('jean' appartient à l'extension du prédicat mari[marie])
ou

marie ∈ femme[jean]

('marie' appartient à l'extension du prédicat femme[jean]).

1.3.1. Le "ayant" et le "de"

Revenons maintenant sur la question SOCRATE :

i mère de personne 1

Si on se remet au Modèle Sémantique de la structure (page I.35) on voit que 'mère' est le nom d'une fonction d'accès et que 'personne 1' est un objet de la catégorie 'personne' d'où part l'arrête qui représente cette fonction d'accès et qui est donc le domaine de celle-ci. La question veut dire :

"énumérer (en l'imprimant) l'extension du prédicat :
mère [personne 1]"

Bien sûr, on a vu qu'une personne n'a qu'une mère et l'extension a un seul élément mais, si on avait la question :

i voiture de personne 1

on aurait l'énumération de plusieurs voitures, en général.

Si, maintenant, on veut connaître toutes les personnes dont la 'personne 5', par exemple, est la mère, on écrira :

i personne ayant mère = personne 5

ce qui veut dire :

"énumérer (et imprimer) l'extension du prédicat :

mère^{-1} [personne 5]"

On voit, donc, que l'opérateur SOCRATE "ayant" joue le rôle de l'exposant , qu'on utilise pour les fonctions inverses.

Quand on utilise, en SOCRATE, les opérateurs ayant ou de, on peut exprimer toutes les questions simples .c'est-à-dire, qui ne concernent qu'une seule fonction d'accès, et seulement si elle est implantée à l'aide d'une I-relation.

Si une fonction d'accès est implantée par un I-lien, (page I.26) les questions auront une forme plus compliquée. Par exemple, soit la structure de la figure 22.

Si on veut demander les voitures de la première personne, c'est-à-dire 'voiture[personne 1] il faut écrire :

i propriété de propriété ayant propriétaire = personne 1

c'est-à-dire qu'il faut "parcourir" le I-lien qui implante la fonction d'accès 'voiture'.

De façon analogue, on formulerait la question inverse ; c'est-à-dire par exemple, voiture^{-1}[voiture 5] se formulerait :

i propriétaire de propriété ayant propriété = voiture 5

1.3.2. Connecteurs et, ou et ¬

En SOCRATE, on peut exprimer des "accès séquentiels filtrés", c'est-à-dire vouloir trier tous les éléments d'une certaine catégorie qui vérifient une propriété simple (c'est le cas de, par exemple, "i personne ayant prénom = rené") ou la combinaison de plusieurs propriétés comme, par exemple :

i personne <u>ayant</u> prénom = rené <u>et</u> âge = 20

ou

i personne <u>ayant</u> prénom = rené <u>ou</u> âge = 20

ou encore

i personne <u>ayant</u> prénom = rené

La **sémantique** de ces trois questions peut s'exprimer à l'aide des opérations ensemblistes (union $\cup$, intersection $\cap$ et complément C).

Enumérer :

$$\text{prénom}^{-1}[\text{rené}] \cap \text{âge}^{-1}[20]$$

$$\text{prénom}^{-1}[\text{rené}] \cup \text{âge}^{-1}[20]$$

$$\underset{\text{personne}}{C} \quad \text{prénom}^{-1}[\text{rené}]$$ [I]

Dans le modèle d'une structure, le fait qu'une arête représentant une fonction d'accès F part d'un noeud C (domaine de F) et arrive à un noeud C' (co-domaine de F) a une grande valeur sémantique. Il veut dire (1.2.1.) :

$$\forall x \ \forall y \quad \{x \in F[y] \rightarrow x \in C' \wedge y \in C\}$$

Une question :

i personne <u>ayant</u> prénom = 'rené' <u>et</u> 'âge' = 20

n'est correcte que <u>parce que le co-domaine des deux fonctions d'accès 'prénom^{-1}'</u> <u>et âge^{-1}' est la même catégorie</u> (on ne peut faire l'intersection d'ensembles que s'ils sont constitués d'objets de la même catégorie).

[I] le complément est évidemment pris par rapport à la catégorie qui est co-domaine de la fonction 'prénom^{-1}', c'est-à-dire 'personne'.

1.3.3. Composition de questions

On peut poser des questions, en SOCRATE, concernant plusieurs fonctions d'accès,pourvu que certaines règles soient observées en ce qui concerne leurs domaines et co-domaines.

Par exemple :

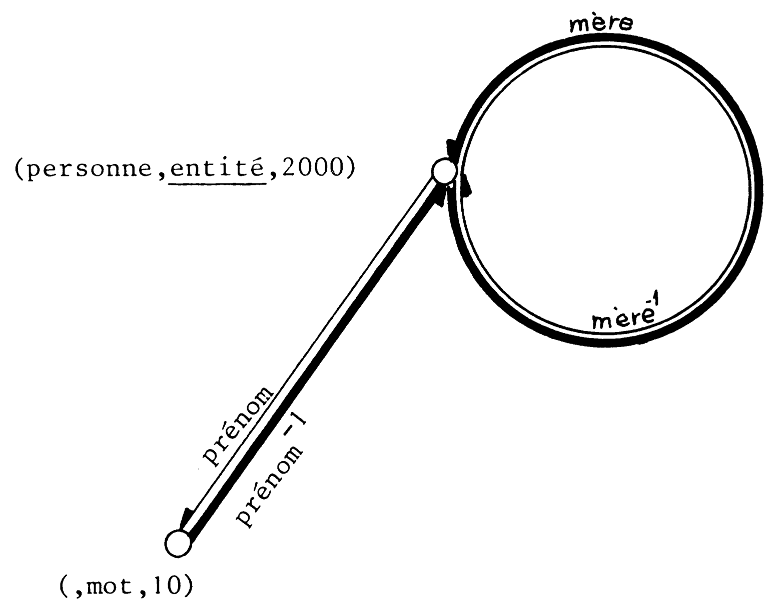
i mère de personne ayant prénom = rené

qui veut dire :

"énumérer l'ensemble :

$\{x \mid \exists y\ (x \in mère[y] \wedge y \in prénom^{-1}\ [rené])\}$

est correcte parce que le co-domaine de 'prénom^{-1}' est le domaine de 'mère'. La question concerne donc la composition des deux fonctions d'accès 'mère' et 'prénom^{-1}' et est représentée dans le modèle (figure 30 de la structure (fig. par le chemin à trait plus gros :



Désormais, au lieu de :

$$\{x \mid \exists y\ (x \in mère[y] \wedge y \in prénom^{-1}[rené])\}$$

j'écrirai :

$$\{x \mid\ x \in mère\ .\ prénom^{-1}[rené]\}$$

ce qui met en évidence le rapport entre la sémantique de la question et le modèle sémantique de la structure.

De manière générale, on peut conclure que : étant donnée une question SOCRATE simple ou composée, sans connecteurs(et ou et ¬) la question est correcte si et seulement si elle correspond, par la méthode vue, à un chemin du modèle sémantique de la structure.

Montrons d'abord, qu'à toute question correcte correspond forcément un chemin.

Dans le paragraphe 1.3.1., j'ai montré que les questions simples peuvent être exprimées à l'aide de :

- l'opérateur de. Elles concernent alors une fonction d'accès représentée par une arête du modèle.

ou

- l'opérateur ayant. Elles concercent une fonction d'accès inverse et puisque le modèle est totalement "inverse",elles sont représentées aussi par une arête.

Les questions contenant plusieurs ayant et de, et qui concernent une fonction d'accès implantée par I-lien, sont elles aussi représentées par une arête, après l'élimination des pseudo-catéogires et pseudo-fonctions d'accès.

La règle qu'on a vue sur la composition de questions SOCRATE (1.3.3.) et la signification du modèle sémantique (1.2.3.) permettent de prouver qu'à toute question composée correspond un chemin du modèle.

Dans l'autre sens, la démonstration peut se faire par induction sur la longueur des chemins :

longueur = 1

Une arête :

(c_1, c_2, c_3) —— F_1 ——▶ (c'_1, c'_2, c'_3)

représente la sémantique :

$$F_1[a]$$

qui se traduit en SOCRATE par :

F_1 de a

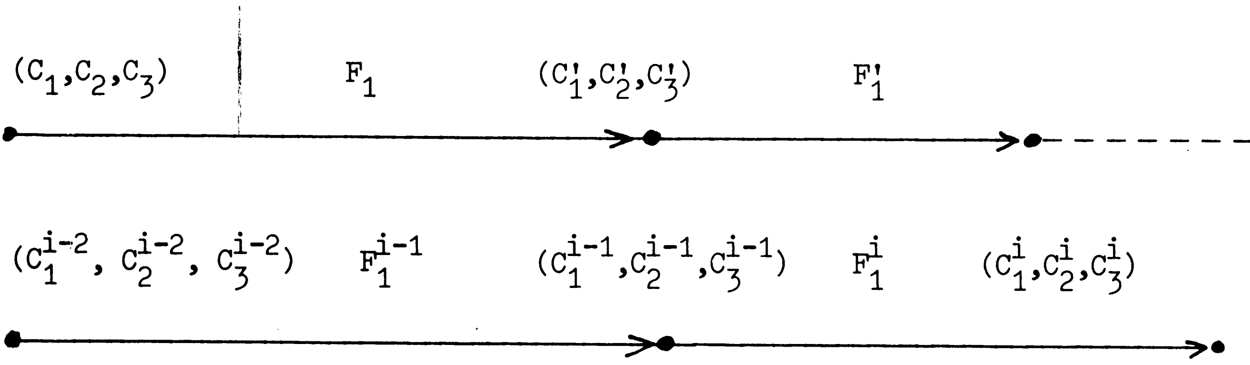
si F_1 est un nom, ou par

$$C_1 \text{ \underline{ayant} } G = a$$

si F_1 est de la forme G^{-1}

Dénotons cette sémantique par S_1.

Si on adopte toujours la notation de la page I.40 alors un chemin de longueur i comme :



sera représenté par :

$$F_1^i \quad S_{i-1}$$

ou par :

$$C^i \text{ \underline{ayant} } G^i = S_{i-1}$$

G^i étant l'inverse de F_1^i

1.4. Applications immédiates de la formalisation précédente

En 1.2 et 1.3 j'ai formalisé la sémantique des structures SOCRATE et du langage d'interrogation et mise-à-jour.

Cette formalisation peut servir immédiatement à la résolution de quelques problèmes graves de l'utilisation de SOCRATE : (a) les changements de structure (changement de l'implantation des données sans changer leur sémantique) en cours d'application et (b) le choix d'une structure.

1.4.1. Changement de structure

En SOCRATE, si l'utilisateur veut changer la structure (par exemple, pour diminuer le coût de l'utilisation), alors il faut qu'il réécrive tous les programmes existants (cf. propriété vue en page 0.5), même si la structure et les programmes conservent leur sémantique.

Je décrirai ici un moyen algorithmique de transformer les programmes existants dans des programmes toujours valables, à condition naturellement que la nouvelle structure ait le même modèle sémantique que l'ancienne.

Structures équivalentes. Questions équivalentes

Définition : Je dirai que deux structures SOCRATE S1 et S2 sont équivalentes si et seulement si elles ont le même modèle sémantique (1.2.3).

Selon cette définition, et d'après la signification des modèles sémantiques, deux structures sont équivalentes si et seulement si elles déclarent les mêmes catégories, les mêmes fonctions d'accès et si celles-ci ont les mêmes propriétés (0*, 01 ou 11).

En d'autres termes, deux structures sont équivalentes si et seulement si elles décrivent la même sémantique des données, abstraction faite de l'implantation des fonctions d'accès. (c'est-à-dire, leurs différences n'existent qu'en ce qui concerne les I-relations et I-liens choisis, pour implanter les fonctions d'accès).

Par exemple :

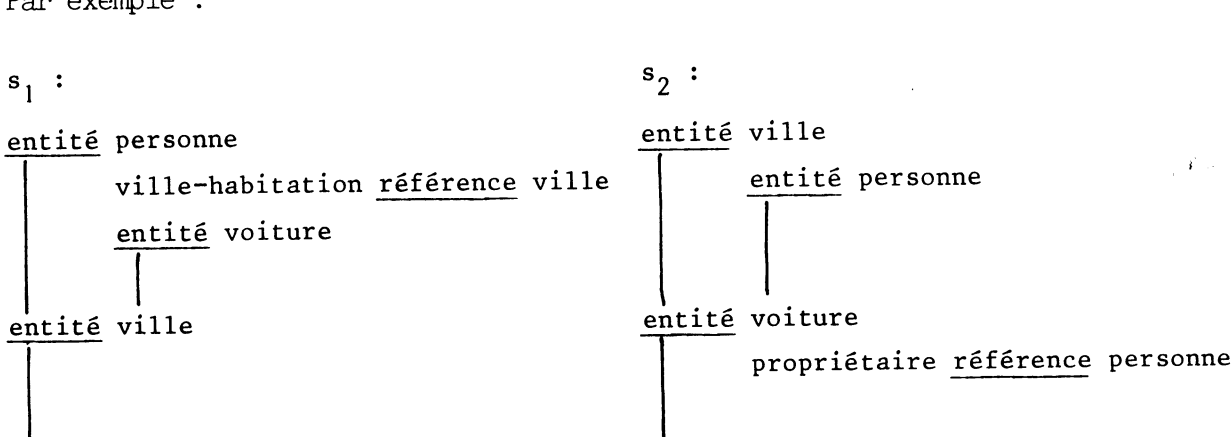


FIGURE 31

sont équivalentes parce que si leurs I-graphes sont :

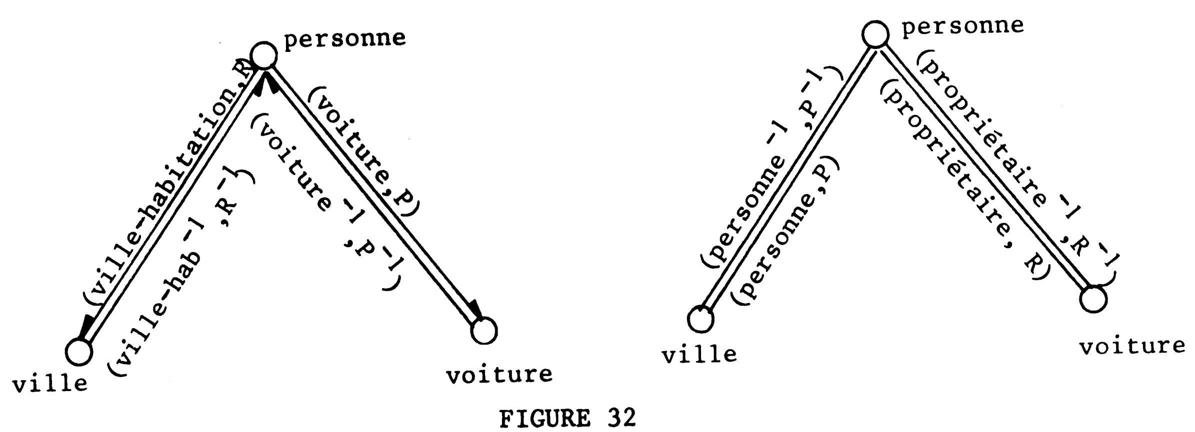


FIGURE 32

leurs modèles sont identiques :

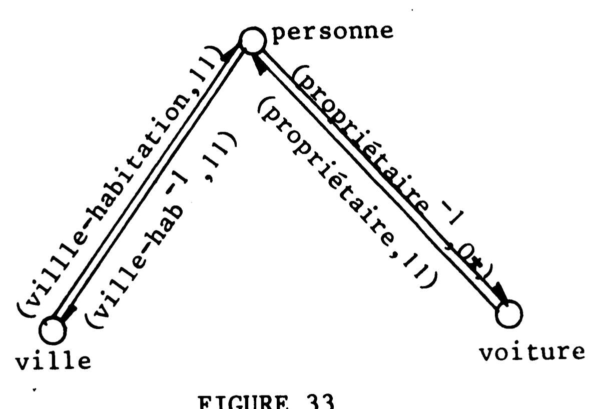


FIGURE 33

Questions équivalentes

La question suivante, écrite en termes de la structure S_1

i ville-habitation de personne ayant voiture = v_1

dont la sémantique est "ville-habitation.propriétaire^{-1}[v_1]"se formulera différemment quand je passe à la structure S_2 (figure 31) :

i ville ayant personne = propriétaire de v_1

Ces deux questions sont équivalentes parce qu'elles ont la même sémantique (figure 33). Ce que je veux c'est qu'en changeant de structure, je puisse faire exécuter un algorithme qui transforme automatiquement l'une dans l'autre.

De manière générale :
Etant données deux structures SOCRATE équivalentes S_1 et S_2, pour toute question q_1 formulée en fonction de S_1
, il existe une question q_2 formulée en fonction de S_2 qui lui est équivalente (possède la même sémantique, au sens de la page I.41) et il y a un algorithme capable de traduire l'une dans l'autre.

Ceci découle immédiatement de la définition de la page I.43 (équivalence de structures) et de ce que j'ai dit dans les pages I.41 et I.39.

De façon informelle, on voit que, donnée q_1 on peut retrouver le chemin du modèle de S_1 qui lui correspond, (méthode décrite dans la page I.41). . Ensuite, à partir de ce chemin, on peut trouver le chemin du I-graphe de S_1 (1.2.3.1.) qui est marqué avec les I-relations correspondantes, ce qui permet d'appliquer la méthode de la page I.41 et trouver la formulation q_2 voulue.

Enfin, on peut dire qu'il y a un algorithme capable de traduire un programme SOCRATE écrit en termes d'une structure S_1 dans un autre écrit en termes d'une structure équivalente S_2.

En effet, dans un programme quelconque, ce qui change quand on passe de S_1 à S_2 c'est la forme des questions, c'est-à-dire, de l'accès aux données. Par conséquent, traduire le programme dans un programme équivalent revient simplement à traduire, à l'intérieur de chaque interrogation ou mise-à-jour, les questions dans les questions équivalentes.

Par exemple le programme :

pour personne x1 **ayant** ville-habitation = ville 4

i voiture **de** x1

fin

qui est écrit en termes de la structure S_1 (figure 31) devient :

pour personne x1 **de** ville 4

i voiture **ayant** propriétaire = x1

fin

quand il est écrit en termes de la structure S_2

1.4.2. Choix d'une structure

L'utilisateur qui veut choisir une structure pour une application, se trouve toujours devant plusieurs structures équivalentes (page I.43),qui sont toutes celles qui vérifient la même sémantique de ses données.

Le choix d'une structure devrait se produire en deux étapes :

1 - définition du modèle sémantique des données
2 - pour chaque fonction d'accès, en respectant les propriétés définies en 1, choix d'une I-relation ou d'un I-lien capable de l'implanter.

Ce choix devra respecter les trois sémantiques suivantes :

C1 - si on choisit une I-relation i pour implanter une fonction d'accès entre deux catégories c et c' marquée (x,y) alors il faut que

$$\varphi(c,i,c') = y$$

où φ est la fonction représentée dans le tableau de la page I.28

C2 - si la fonction inverse de la fonction implantée est marquée à son tour avec (z,w) alors il faut que :

$$\varphi(c\ ,i^{-1},c) = w$$

C3 - D'une catégorie ne peuvent pas partir plusieurs arrêtes marquées avec la I-relation P^{-1}. Ceci traduit le fait que, dans une structure, une catégorie ne peut pas être fille de deux catégories distinctes.

L'énumération des I-relations ou I-liens susceptibles d'implanter une fonction d'accès, doit se faire selon la stratégie suivante :

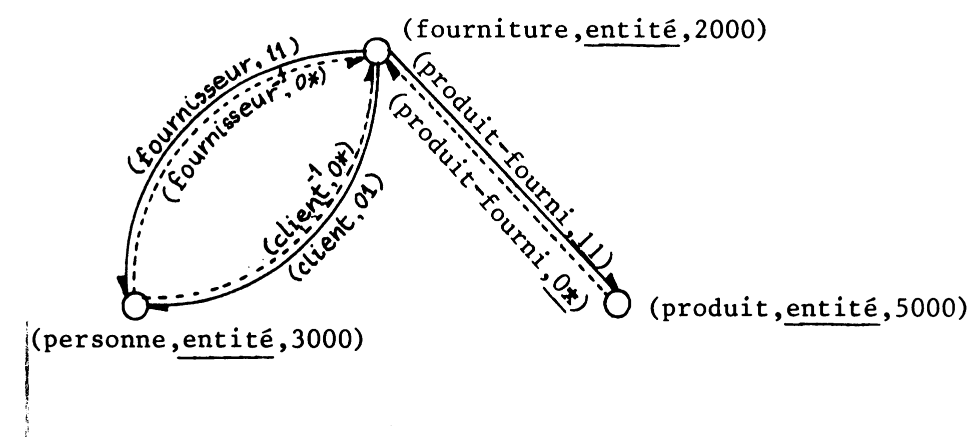
1 - "entrer" dans le tableau de la page I.28 par la colonne de droite et regarder si les types des catégories c_1 et c_2, (deuxième colonne) qui correspondent à y (3ième colonne) sont ceux de c et c'. Si c'est le cas, la I-relation de la 1ère colonne est une implantation possible

2 - quand on a épuisé toutes les I-relations, on passe à l'énumération des I-liens. Pour cela, on "entre" dans le tableau de la page I.30 avec la valeur y (colonne de droite), comme on faisait pour le tableau précédent. On essaye, pour chaque couple de la colonne de gauche, de trouver une pseudo-catégorie et des implantations pour les deux pseudo-fonctions d'accès, en appliquant cet algorithme récursivement à partir de 1.

Je donne ensuite un exemple de choix de structure.

1.4.2.1. Exemple

Soit à implanter le modèle suivant :



qui représente une relation 3-aire appelée "fourniture" entre des fournisseurs, des produits et des clients.

J'énumère, ensuite, les implantations possibles pour chacune des 3 relations binaires qui la constituent.

Relation 'produit-fourni'

'produit-fourni' peut être implanté par :

R
ou
P^{-1}

ou par un des deux I-liens

R (-,entité,n)R
P^{-1}(-,entité,n)P^{-1}

ou encore par un I-lien avec une des formes suivantes :

R(-,entité,n)R (,entité,n) R ... etc.
P^{-1}(-,entité,n)P^{-1}(-,entité,n) P^{-1} ... etc
R(-,entité,n)P^{-1}(-,entité,n)R ... etc.
P^{-1}(-,entité,n)R (-,entité,n)P^{-1} ... etc.

De manière générale, il y a un nombre infini de I-liens pour implanter la relation. On peut symboliser ces I-liens par les éléments du langage suivant :

$L = L_1^q$ où $q \quad 1$

et $L_1 = R^m \quad P^{-1^n} \quad m \quad 0, n \quad 0$

De toutes les solutions possibles pour implanter "produit-fourni" je ne retiendrai que les 3 suivantes :

R

P^{-1}

P^{-1}(-,entité,-)R

puisque toutes les autres conduiraient à des structures dont l'utilisation serait sûrement plus coûteuse que celles qu'on obtient avec ces 3 solutions.

Relation "fournisseur"

Les propriétés de la relation ('fournisseur', 'fournisseur^{-1}') sont les mêmes que celles de la relation 'produit-fourni'. Les solutions sont donc les mêmes.

Relation "client"

On peut constater que la relation 'client' ne peut pas être implantée à l'aide d'une seule I-relation.

Si on dénote ' ' le I-lien

P(-,entité,1) R

alors tout I-lien symboliquement représenté par un élément du langage

$L = \{\ell\} X \ L_1^*$ '[I]

où L_1 = 'ℓ' et e représente

(-,entité,1)

est une solution.

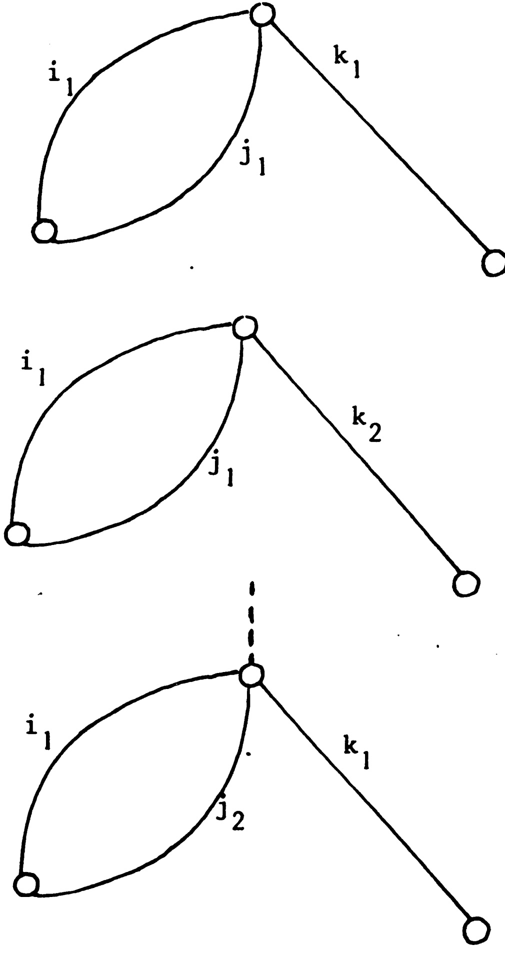
[I] - 'X' représente le produit de langages et '*' l'opération étoile de Kleene

Pour des raisons analogues à celles indiquées dans la page précédente, on ne retiendra que le I-lien :

P(-,entité,1)R

Enfin, si je veux,maintenant,connaître les structures qui implantent le modèle sémantique de la page I.48 , il suffit de combiner trois à trois les solutions trouvées pour implanter les 3 relations qui le constituent (produit-fourni,fournisseur et client) de la façon suivante : dénotons $i_1, i_2, \ldots$ etc. des I-relations ou I-liens par lesquels on peut implanter la relation produit-fourni ______________ ,j_1, j_2, ...etc des I-relations ou I-liens par lequels on peut implanter "fournisseur" , et k_1, k_2,k_3... des I-relations ou I-liens pour implanter la relation "client"

Les structures qui implantent le modèle seront :



et ainsi de suite.

Cependant, il y a des combinaisons qui ne donnent pas de structures : ce sont toutes celles où deux fonctions d'accès ayant le même domaine sont les deux implantées par la I-relation P^{-1}.

Regardons ensuite 5 structures qu'on obtient en combinant les I-relations et I-liens trouvés avant pour les 3 fonctions d'accès.

1 :

```
entité fourniture
|       produit-fourni référence produit
|       fournisseur référence personne
|       entité client 1
|       |       valeur référence personne
```

Cette structure favorise les questions SOCRATE dont la traduction, sous forme de fonctions d'accès, est :

produit-fourni [fourniture i]
fournisseur [fourniture i]
client [fourniture i]

Par ailleurs, on peut rajouter cette structure à une structure ne contenant que les personnes et les pièces, sans avoir à redéfinir la structure.

2 :

```
entité personne
|       entité fourniture
|       |       produit-fourni référence produit
|       |       entité client 1
|       |       |       valeur référence personne
```

Cette structure favorise les questions SOCRATE dont la traduction, sous forme de fonctions d'accès est :

produit-fourni[fourniture i]
fournisseur^{-1} [personne i]
client [fourniture i]

Ici, on a implanté la fonction d'accès inverse de 'fournisseur' au contraire de ce qu'on a fait, par exemple, dans la structure 1

Par ailleurs, la suppression d'une personne est accompagnée de celle de toutes les fournitures dont elle est le fournisseur.

3 :

```
entité e
|   produit-fourni référence produit
|   fournisseur référence personne
|   entité fourniture
|   |   entité client 1
|   |   |   valeur référence personne
```

Cette structure favorise les accès (questions) de forme générale :

$$\text{produit-fourni}^{-1}\ [\text{produit } i] \cap \text{fournisseur}^{-1}\ [\text{personne } j]$$

En plus, si on a, au départ, une structure constituée simplement des "personnes" et "produits", on peut rajouter l' "entité e" et les fournitures <u>dynamiquement</u> sans avoir à redéfinir la structure et les données. Par contre, si la structure choisie était la structure 2 ou la structure 1 <u>cela ne serait pas possible</u>.

4 :

```
entité produit
|   entité fourniture
|   |   fournisseur référence personne
|   |   entité client 1
|   |   |   valeur référence personne
```

Cette structure favorise les question de forme générale :

$$\text{produit-fourni}^{-1}[\text{produit } i]$$

puisqu'on a implanté la fonction d'accès <u>inverse</u> de "produit-fourni"

Enfin, la suppression d'une pièce est accompagnée de celle de toutes les fournitures correspondantes.

5 :

```
entité e
    produit-fourni référence produit
    entité fourniture
        fournisseur référence personne
        entité client 1
            valeur référence personne
```

Cette structure favorise les accès (question) de forme générale :

produit-fourni [fourniture i]
fournisseur [fourniture i]
produit-fourni. fournisseur^{-1} [personne i]
(les produits fournis par la personne i).

Elle peut être rajoutée à une autre structure, comme les structures 1 et 3, mais, au contraire de 2 la suppression d'une fourniture élimine la relation (implicite) entre les fournisseurs et les produits.

1.4.3. Comment améliorer une structure

Un certain temps après avoir choisi une structure, on peut vouloir favoriser des accès (questions) qui, dans cette structure, et en fonction des circonstances au moment du choix, ne sont pas favorisés.

1.4.3.1. Ajout de redondance

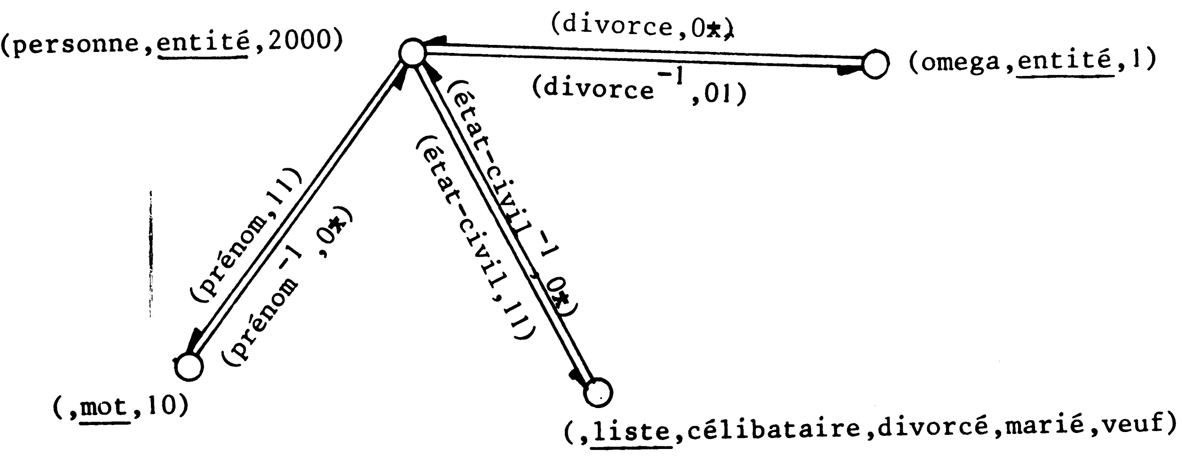
Fichiers inversés

La redondance joue un rôle très important dans le choix d'une structure. Par exemple, on peut définir une structure :

```
entité personne 2000
    début
        prénom mot 10
        état-civil liste célibataire marié veuf divorcé
    fin
divorcé inverse personne
```

qui évite d'avoir à parcourir toutes les personnes pour exécuter un traitement ne concernant que les divorcés.

Le modèle de cette structure est



On est en présence,ici,de ce qui est appelé,en base de données,un fichier inversé. On comprend d'ailleurs que cela s'appelle inversé parce que il ne fait qu'implanter,de façon redondante,la fonction inverse d'une fonction d'accès (état-civil) pour une valeur précise de l'argument ('divorcé').

Le fichier inverse représente le prédicat :

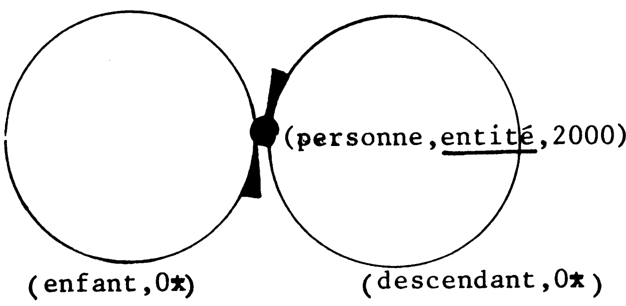
$$\text{état-civil}^{-1}[\text{divorcé}]$$

Composition de relations

Une deuxième sorte de redondances, utiles du point de vue de l'optimisation du coût d'accès, est celle où une relation est, à chaque instant, la composition d'une ou plusieurs relations. Par exemple :

```
entité personne 2000
       début
       |    enfant inverse personne
       |    descendant inverse personne
       fin
```

ce qui donne le modèle :



"descendant" est redondant parce que

$$\{y \mid y \in \text{descendant } [x]\} \equiv \{y \mid y \in \text{enfant.descendant}[x]\}$$

1.4.3.2. Ajout de redondances aux structures de l'exemple du paragraphe 1.4.1.

Je donne ensuite une liste de quelques redondances qu'on peut ajouter aux structures des pages I.51 à I.53 afin de diminuer le coût d'utilisation.

structure 1 (page I.51)

Si on veut favoriser les questions :

produit-fourni . fournisseur^{-1} [personne i]

on peut insérer dans l'entité 'personne' l'inverse suivant :

entité personne

  produit disponible **inverse** produit

Cet inverse maintient les propriétés des fonctions d'accès [I]

[I] 'produit-fourni.fournisseur^{-1}' à la propriété 0* et 'fournisseur.$\text{produit-fourni}^{-1}$' a aussi cette propriété ce qui correspond bien aux propriétés de prdouit-disponible et $\text{produit-disponible}^{-1}$

On peut aussi favoriser l'accès inverse en faisant :

<u>entité</u> produit
    <u>début</u>
        fournisseur-possible <u>inverse</u> personne
    <u>fin</u>

'fournisseur-possible' étant inverse de 'produit-disponible'. Cette structure maintient aussi les propriétés des fonctions d'accès.

structure 2 (page I.51)

A cette structure on peut ajouter, par exemple, la redondance suivante :

<u>entité</u> personne
| achat <u>inverse</u> fourniture

ce qui permet de retrouver rapidement toutes les fournitures concernant un client donné.

On peut aussi ajouter :

<u>entité</u> personne
| produits-achetés <u>inverse</u> produit

afin d'améliorer les questions du type 'produit-fourni .client [personne i]'

structure 3 (page I.52)

On peut ajouter à cette structure l'"inverse" suivant :

produit-4-personne-3 <u>inverse</u> fourniture

ce qui permettra de retrouver rapidement toutes fournitures concernant le produit 4 et la personne 3

1.4.4. Structures faiblement équivalentes

Je dirai que deux structures S_1 et S_2 sont faiblement équivalentes si et seulement si elles ont le même modèle sémantique, sauf en ce qui concerne les propriétés des relations : il suffira que, pour chaque relation, les propriétés d'une fonction d'accès (F ou F^{-1}) soient identiques en S_1 et S_2.

Par exemple, les deux structures suivantes sont faiblement équivalentes :

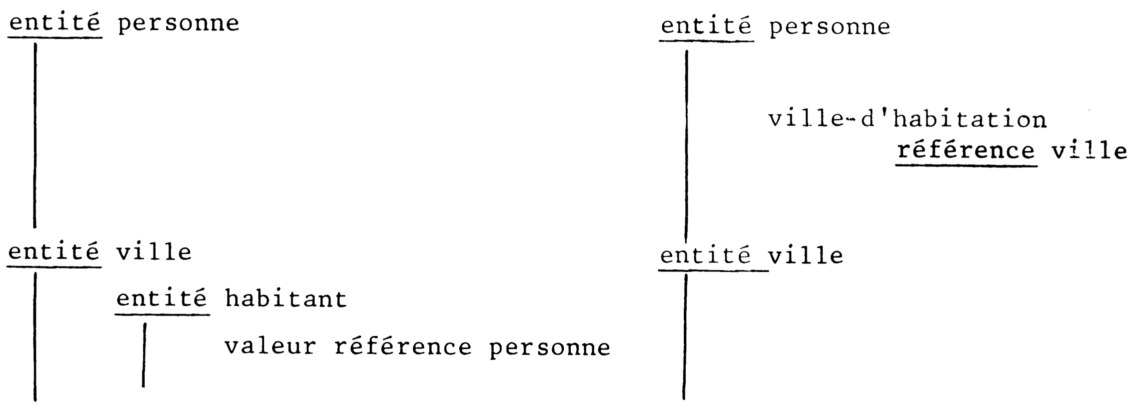
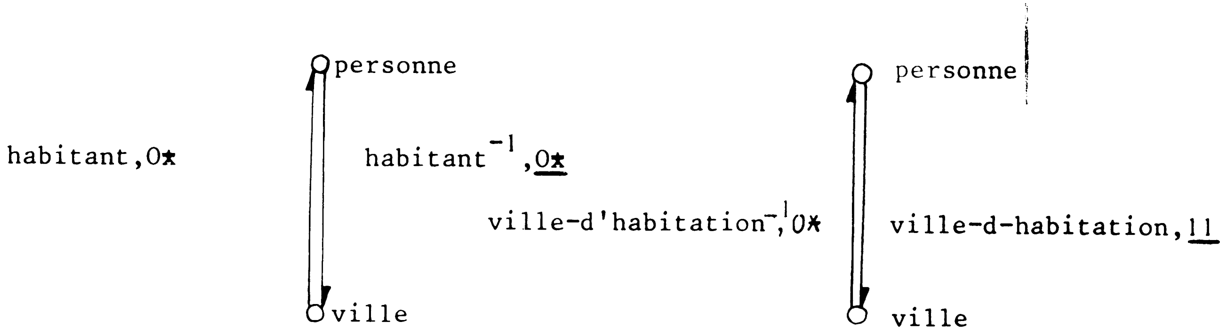


FIGURE 34

parce que leurs modèles sémantiques :



diffèrent en ce qui concerne les propriétés de la fonction d'accès 'ville-d-habitation (ou "habitant^{-1}").

L'intérêt pratique de l'équivalence faible est très grand : souvent l'utilisateur qui veut choisir une structure opère son choix dans un ensemble de structures faiblement équivalentes et non équivalentes comme je l'avais dit au paragraphe 1.4.2.

La raison est que, pour l'utilisation qu'il va en faire, les différences sémantiques ne comptent pas, ou alors, tout simplement, il n'a pas conscience de ces différences, et il ne s'en apercevra que plus tard.

Par exemple, des deux structures de la figure 34, c'est celle de droite qui possède la bonne sémantique. Celle de gauche, qui lui est faiblement équivalente, peut cependant être choisie par l'utilisateur en raison du fait qu'elle favorise les questions sur les habitants d'une ville si ces questions sont très fréquentes.

Evidemment, le choix d'une structure dont le modèle n'est pas exactement le bon, est dangereux du point de vue de la correction des programmes et de la cohérence des données.

Conclusion

Des critères et méthodes exposées on peut déduire que le choix d'une implémentation pour une relation représentée par deux fonctions d'accès F et F^{-1}, vise d'abord à décider d'implanter F ou F^{-1} ou les deux, ensuite choisir des I-relations ou I-liens convenables et, en cas d'échec, opter pour une implantation faiblement équivalente.

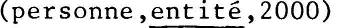
1.5. Ce qui manque en SOCRATE

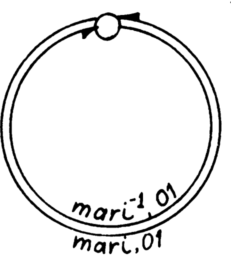
Socrate est incomplet d'un point de vue sémantique (ou logique) parce qu'il ne permet pas à l'utilisateur de déclarer des règles sémantiques importantes pour l'utilisation d'une base de données, ce que je vais justifier dans la suite de ce chapitre.

1.5.1. Modèles sémantiques non-implémentables

On a vu au paragraphe 1.4.2 que la deuxième étape du choix d'une structure consiste dans le choix de I-relations et I-liens capables d'implanter les différentes fonctions d'accès décrites dans le modèle.

Si dans l'exemple du paragraphe I.4 on a pu trouver plusieurs solutions pour implanter le modèle, ceci n'est pas toujours le cas. Par exemple, si on voulait implanter :





et on choisissait le I-lien le plus simple qui implante 'mari', c'est-à-dire :

entité personne 2000

    entité mari 1

        valeur référence personne

la fonction "$mari^{-1}$" aurait la propriété O* ce qui ne serait pas correct. Le lecteur peut constater qu'il n'y a aucune solution pour implanter le modèle [I].

[I] Evidemment, l'utilisateur SOCRATE choisirait la structure indiquée en haut comme "dernier recours". Cependant la cohérence des données est menacée parce que par exemple rien ne l'empêche de faire les deux mises-à-jour suivantes :

    m   mari [personne 1] = personne 2

    m   mari [personne 5] = personne 2

et alors la personne 2 a deux femmes ...

Dans le tableau suivant, je donne les propriétés possible d'un couple de deux fonctions d'accès, et je signale avec 'NI' celles qui <u>ne sont pas pas implantables</u> :

(NI) 01 → / ← 01	(NI) 11 → / ← 01	0* → / ← 01
(NI) 01 → / ← 11	(NI) 11 → / ← 11	0* → / ← 11
01 → / ← 0*	11 → / ← 0*	0* → / ← 0*

FIGURE 35

Il faut remarquer cependant que les couples 01 → / ← 01 , 11 → / ← 01 sont implantables dans le cas particulier où le co-domaine de F (représenté par la flèche de gauche à droite)est de type mot discriminant.

1.5.2. Contraintes sur les relations n-aires

Supposons que l'on veuille déclarer une relation ternaire qu'on appellera "fourniture" entre personnes, produits et clients, voulant dire le suivant :

"fourniture x y z) est vrai si et seulement si la personne x fournit le produit y au client z.

une telle relation se représenterait en SOCRATE par une des 5 structures que j'ai discutées en 1.4.2.1.

Les catégories 'personne' et 'produit' étant définies ailleurs dans la structure.

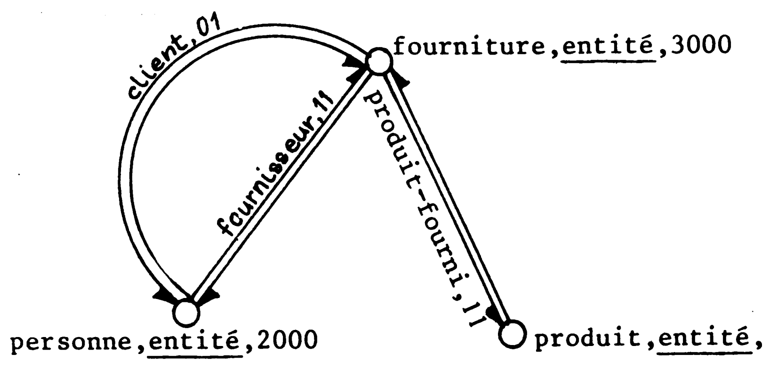
Or, si je veux déclarer la contrainte suivante :

"un produit p n'est pas fourni par plus d'une personne à un même client"

(formellement,

$$\forall x \in \text{produit}(\exists y \in \text{client}(\exists z \in \text{personne}(\text{fourniture } x\ y\ z)) \wedge$$
$$\exists y' \in \text{client}\ (\text{fourniture } x\ y'\ z) \rightarrow (y = y')))$$

je ne peux pas le faire parce que je ne peut pas établir des contraintes que sur les relations indiquées dans le modèle :



alors qu'en réalité la contrainte voulue concerne des relations, implicitement contenues en "fourniture", entre les couples (<produit><client>) et les personnes, et qu'on pourrait représenter informellement par :

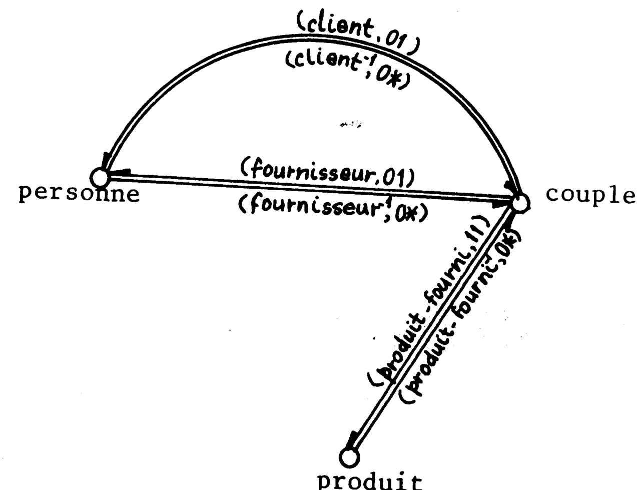
produit client

$$\text{produit} \times \text{client} \xrightarrow{\underline{01}} \text{personne}$$

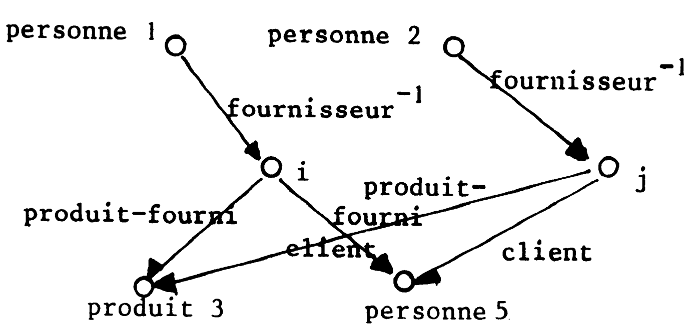
Mais alors, pourquoi ne pas choisir la structure suivante :

```
entité couple
   |  produit-fourni référence produit
   |  entité client 1
   |        valeur référence personne
   |  entité fournisseur 1
   |     |  valeur référence personne
   |     |
```

le modèle de laquelle est :



La raison est que l'on peut avoir des données comme les suivantes :



Une solution serait alors d'ajouter la règle sémantique :

"il ne peut pas exister deux couples (p,a) et (p',a') tels que p = p' et a = a'" mais cela ne peut pas être déclaré en SOCRATE.

E.W. CODD a mis en évidence dans ses travaux l'importance des contraintes ou règles sémantiques de ce genre en Gestion.

1.5.3. Les redondances

On a vu, dans la page qu'on choisit souvent d'implanter, de façon "redondante", la fonction d'accès inverse d'une fonction déjà implantée

(fichiers inversés) ou alors une fonction qui soit la composition, à chaque instant, de deux autres fonctions.

En réalité, il n'y a pas de vraie redondance. Je dis pourquoi.

Supposons que, dans le cas de la page , on fasse la mise-à-jour suivante :

m état-civil [personne 5] = divorcé

Or, puisque le système se limitera à changer l'extension du prédicat 'état-civil [personne 5]' les deux propositions suivantes seront vraies, après la mise-à-jour :

$$\text{personne } 5 \in \text{état-civil}^{-1} [\text{divorcé}]$$

$$\text{personne } 5 \notin \text{divorcé}$$

ce qui rend la base de données non-cohérente.

Il faudrait pouvoir déclarer la règle sémantique suivante :
"si l'état-civil d'une personne a la valeur 'divorcé' alors la personne a la propriété d'être divorcée"

Dans le deuxième exemple (enfant et descendant) il y a un phénomène analogue , si on voulait maintenir la cohérence des données il faudrait pouvoir déclarer la règle sémantique :

"si la personne x est enfant d'une autre y, qui est elle-même descendante d'une personne z, alors x est descendant de z"

$$x \in \text{enfant} [y] \wedge y \in \text{descendant} [z] \rightarrow x \in \text{descendant} [z]$$

ce qui ne peut pas se faire en SOCRATE.

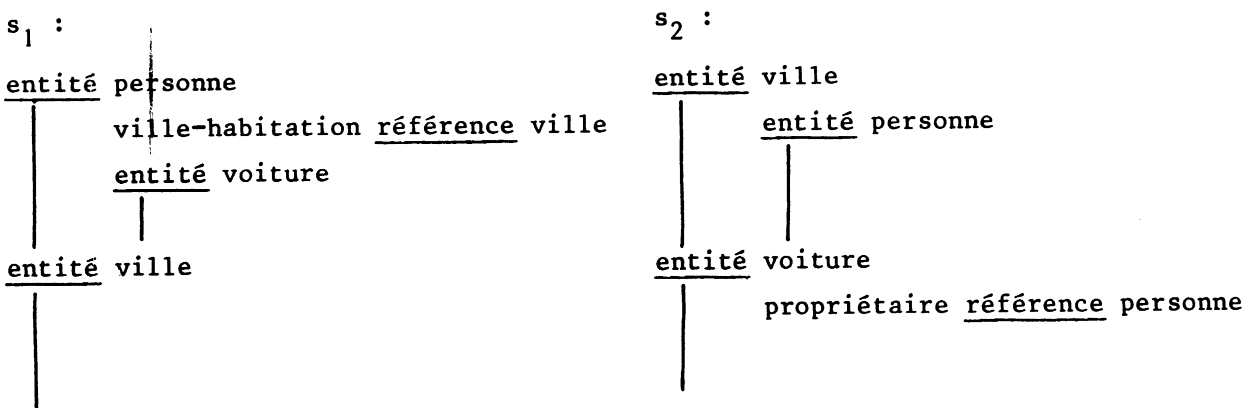
Encore un troisième exemple est celui d'une structure :

```
entité produit 3000
    prix-de-vente naturel 0 a 1000
    prix-d-achat naturel 0 a 1000
    bénéfice naturel 0 a 1000
```

Le bénéfice d'un produit p devant être à chaque instant la différence entre le prix-de-vente de p et son prix-d'achat, et étant donné qu'on ne peut pas déclarer une telle règle, la base de données passera forcément par des états non cohérents.

1.5.4. Différences entre les I-relations P(père) et R(référence)

Regardons, encore une fois, l'exemple des deux structures S_1 et S_2 donnés dans la figure 32 :



Ces deux structures sont équivalentes parce qu'elles déclarent les mêmes catégories et les mêmes fonctions d'accès et celles-ci ont les mêmes propriétés dans l'une et l'autre.

Cependant, leur utilisation devra tenir compte du fait que si, dans le cas de S_1, j'élimine une personne alors seront automatiquement éliminées ses voitures, ce qui n'est pas vrai pour S_2, et que, dans le cas de S_2, si j'élimine une ville, sont éliminés automatiquement ses habitants ce qui n'est pas vrai pour S_1.

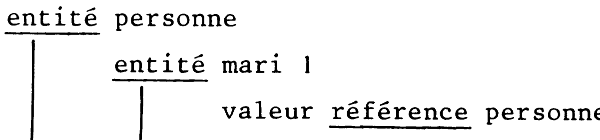
La sémantique contenue dans ces deux structures, qui sont équivalentes au sens de la page , n'est donc pas rigoureusement la même.

Elle ne serait la même qu'à condition de pouvoir ajouter à S_1 la règle : "la suppression d'une ville entraîne celle de tous ses habitants". et à S_2 la règle : "la suppression d'une personne entraîne celle de toutes les voitures dont elle est propriétaire".

Ceci n'est pas possible en SOCRATE.

1.5.5. Structures faiblement équivalentes

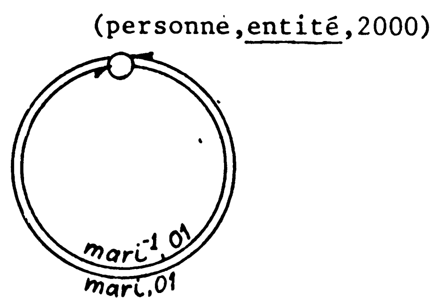
Ce raisonnement à propos des structures S_1 et S_2 peut s'appliquer aux modèles non- implémentables en SOCRATE parce que, si, par exemple, on pouvait ajouter à la structure :



la règle sémantique :

$$\forall x\, \forall y\, \{\{y \in mari^{-1}[x] \;\wedge\; y\, \{z \in mari^{-1}[x]\}\} \rightarrow z = y\}$$

alors on arriverait à implanter effectivement le modèle :



1.5.6. Règles sémantiques ad-hoc

Chaque environnement réel a des règles sémantiques ad-hoc pour la description desquelles les modèles sémantiques ne suffisent pas, et dont j'ai parlé en 0.3.

En fait, si on se penche un peu sur des axiomatisations d'environnements réels telles que, par exemple, décrites en CARNAP [51] on constate qu'il n'y a pas "a priori" une forme standard c'est-à-dire qu'il faut disposer d'un langage aussi étendu que le Calcul de Prédicats ou un langage équivalent pour exprimer les règles sémantiques propres à chaque environnement.

1.6. Conclusion

Parmi les applications immédiates de la formalisation de SOCRATE dont j'ai parlé, je voulais analyser mieux les changements de structure.

L'algorithme qui traduit un programme écrit en termes d'une structure, dans le programme équivalent écrit en termes d'une autre, équivalente à la première, ne résoud qu'une partie des difficultés causées par la dépendance du langage vis-à-vis de l'organisation des données.

Tout d'abord, la traduction peut peser lourd dans le coût, ce qui à la limite peut faire perdre les avantages cherchés en changeant de structure.

Enfin, le langage contient toujours les opérateurs et constructions qui expriment cette dépendance, et le rendent difficile à comprendre.

E.F. CODD a proposé [6] que la description des données soit faite avec des relations n-aires (RELATIONAL MODEL).

Une relation n-aire est vue comme un sous-ensemble du produit cartésien de n types d'objets ("constituants").

La data-indépendance découle du fait que le découpage interne de l'ensemble de relations n'est pas le même que pour l'utilisateur.

Je reviendrai sur le RELATIONAL MODEL au paragraphe 2.1.5.

PARTIE 2

MODELE SEMANTIQUE COMPLET

2.1. INTRODUCTION

J'ai conclu la partie précédente par un double constat d'échec, à propos de SOCRATE : d'abord, l'algorithme de traduction de programmes en programmes équivalents ne suffit pas à rendre le système data-independant et ensuite, la structure et le langage ne permettent de représenter et manipuler qu'une très faible partie de la sémantique des environnements réels.

Or, dans la Partie 0, j'ai expliqué pourquoi il faut que les nouveaux systèmes soient data-indépendants, et complets d'un point de vue sémantique.

Dans la suite, je vais proposer une nouvelle méthodologie pour la description de données (Partie 2) et pour le langage-utilisateur (Partie 3), basée sur le formalisme de catégories et fonctions d'accès développé dans la Partie précédente.

Dans cette Partie 2 je m'occupe de la description de données que j'appellerai MODELE SEMANTIQUE COMPLET, et qui est un Modèle Sémantique qu'on complète par l'ajout de règles sémantiques ad-hoc.[0.3]

Les principaux changements que j'apporte au Modèle Sémantique défini dans la partie précédente sont :

- toute indication pour la codification est reléguée au niveau mémoire (par exemple, les 'mots' et 'textes' sont remplacés par la catégorie unique identificateur et, de même, les indications sur le nombre maximum de caractères ou lignes).
- on peut déclarer des types structurés par les opérations ensemblistes (union $\cup$ intersection $\cap$, complément C, et sous-ensemble) appliquées sur des catégories. Ces types seront appelés des propriétés catégorielles.

Les expressions avec une des 3 formes générales suivantes :

(,mot,x)
(,texte,y)
(,liste,x y z ...)
(,naturel, x a y)

que j'ai appelées 'catégories' dans la partie précédente sont en fait des 'propriétés catégorielles'. Par exemple (,liste, marié veuf célibataire) déclare un sous-ensemble de la catégorie 'identificateur'

Après le Modèle Sémantique, je présente les opérateurs de création,

Enfin, je définis une forme générale pour stocker les règles sémantiques ad-hoc les METHODES.

Je conclurai cette partie par une brève comparaison entre les METHODES et leurs équivalents en PLANNER [35] , - les "theorems" -, et je situerai les deux concepts par rapport à ce qui existe dans les langages classiques.

2.2. MODÈLE SÉMANTIQUE

2.2.1. Catégories

2.2.1.1. Définition et exemples

La définition d'un Modèle Sémantique commence par la déclaration des catégories. Une catégorie est une propriété primitive d'objets.

Il y a deux grandes sortes de catégories : les catégories d'entités (I.12)(dont les objets sont codifiés comme des entités) que j'appellerai désormais, catégories concrètes (c-catégories) et les autres que j'appellerai catégories abstraites (a-catégories).

Les catégories abstraites sont les trois suivantes :

acatg (identificateur)
acatg (naturel)

et ne sont pas déclarées par l'utilisateur parce qu'elles sont définies implicitement ("built-in" dans la littérature anglaise).

Quand on déclare une catégorie concrète on doit simplement indique son nom :

ccatg (personne)
ccatg (voiture)

2.2.1.2. Sémantique des catégories

Les catégories ont une sémantique assez restrictive :

1 - tout objet x est forcément d'une catégorie et d'une seule, à un instant.

2 - un objet x ne peut pas changer de catégorie (si un objet x appartient à une catégorie c il ne pourra pas cesser d'appartenir à c).

Ces deux propriétés 1 et 2 jouent un rôle fondamental dans l'acceptation d'un programme : de même qu'en ALGOL il y a des contraintes sur le(s) type(s) des objets sur lesquels agit un opérateur déterminé, ici aussi un opérateur n'est défini que sur des objets de 'catégories' bien précises

2.2.2. Propriétés catégorielles : 'prpcg'

Une propriété catégorielle est une propriété non-primitive d'objets structurée à partir de catégories et pouvant être :

1 - un sous-ensemble d'une catégorie ou propriété catégorielle,

2 - l'intersection ou le complément de propriétés catégorielles.

Sous-ensemble

Par exemple, si on déclare :

prpcg,, lu (male(0),personne)

ceci veut dire que les males sont un sous-ensemble des personnes. En d'autres termes les males sont forcément des personnes.

Les personnes qui sont males devront être indiquées une à une, et c'est pourquoi j'ai indiqué 'lu' après prpcg.

union et complément

Au contraire de ces 'prpcg' la déclaration de celles qui sont l'intersection ou complément d'autres 'prpcg', se fait en indiquant la façon de déduire leurs éléments.

Par exemple, si on déclare :

prpcg,,lu(adulte(0,),personne)

alors on peut déclarer ensuite :

prpcg,,ded(homme,intersection male,adulte fint)

La déclaration d'une 'prpcg' qui est complément de 'prpcg's se fait par exemple comme suit :

prpcg,,ded(femelle,personne,complément male)

ce qui veut dire que les femelles sont des personnes (et pas n'importe quel objet ...) qui ne sont pas des males.

2.2.2.1. Sémantique des propriétés catégorielles

Les propriétés catégorielles ont une sémantique moins restreinte que les catégories. En effet, la règle 1 n'est pas applicable : un objet peut n'appartenir à aucune 'prpcg', ou appartenir à une ou plusieurs 'prpcg' en même temps. Cependant, la règle 2 est applicable :

- si un objet x appartient à une 'prpcg' alors il ne pourra pas cesser d'appartenir à la 'prpcg'.

  Cette règle implique entre autres que l'on ne peut pas changer la définition d'une 'prpcg'

2.2.3. Relations

Après la déclaration des catégories et 'prpcg', la définition du Modèle Sémantique se poursuit avec la déclaration des relations pouvant exister entre objets de catégories ou 'prpcg' déclarées auparavant.

Si on veut définir une relation, on se sert de fonctions d'accès

Définition : **Une fonction-d'accès (f) est une fonction qui, à chaque objet x d'une catégorie ou 'prpcg' - domaine de f (dom[f]) - fait correspondre un sous-ensemble que je dénoterai f[x], d'une catégorie ou 'prpcg' (pouvant être la même) - co-domaine de F (co-dom[f]).**

Par exemple, 'père' est le nom d'une fonction d'accès qui à chaque élément x de la catégorie 'personne' (chaque personne), fait correspondre un sous-ensemble de la même catégorie dénoté 'père[x]'.

Comme je l'ai dit en page I.23 , une fonction d'accès f peut être vue comme étant, à chaque instant, un ensemble de prédicats avec la notation générale 'f[x]' où x est un élément de dom[f]. Par exemple, la fonction d'accès 'père' peut être considérée comme l'ensemble des prédicats avec la notation générale 'père[x]' où x est une 'personne'.

Une fonction d'accès f a un cardinal minimum - min[f]- et un cardinal maximum - max[f]. Le cardinal minimum (maximum) de f est le nombre minimum (maximum) d'éléments des extensions des prédicats f[x]quand x parcourt le domaine de f. Par exemple min[père] = 1 et max [père] = 1.

Les propriétés d'une fonction d'accès f,telles que définies en 1.2.2.1. peuvent s'exprimer en fonction de min et max de la façon suivante :

01 (f) si et seulement si min[f]= 0 max[f] = 1

11 (f) si et seulement si min[f]= 1 max[f] = 1

0* (f) si et seulement si min[f]= 0 max[f] = ∞

Relations unaires : propriétés

Si on veut déclarer une relation unaire (propriété) il suffit de déclarer une fonction d'accès et indiquer son co-domaine, puisque son domaine est, par définition, une a-catégorie implicite appelée 'omega',réduite à un seul élément - 'ω'-

Par exemple,

prp (français (0,),personne)

veut dire que la 'prp' est constituée par la fonction d'accès 'français' et que dom[français] = omega et co-dom [français] = personne

Les propriétés catégorilles ('prpcg') sont des cas particuliers de 'prp' avec une sémantique particulière qu'on a déjà décrite en 2.2.2.

Relations binaires

On peut déclarer une relation binaire de deux façons :

1 - On déclare un couple de deux fonctions d'accès f_1 et f_2 inverses l'une de l'autre (f_1 = inv[f_2] et f_2 = inv[f_1]). Pour que deux fonctions d'accès soient inverses l'une de l'autre il faut que (mais ce n'est pas suffisant)
dom[f_1] = co-dom [f_2] et dom[f_2] = co-dom [f_1].

Pour déclarer un couple de fonctions d'accès inverses l'une de l'autre, on utilise l'opérateur 'rel' et, pour chacune des fonctions, on indique le co-domaine.

Par exemple,

ccatg (personne)
ccatg (voiture)
rel (propriétaire(1,1),personne,propriété(0,),voiture)

veut dire :

propriétaire = inv [propriété]
propriété = inv [propriétaire]
co-dom [propriétaire] = personne
co-dom [propriété] = voiture
min [propriétaire] = 1
max [propriétaire] = 1
min [propriété] = 0
max [propriété] = ∞

2 - La deuxième façon de délcarer une relation binaire consiste à déclarer : une c-catégorie qui porte, comme nom, le nom de la relation et deux fonctions d'accès. Ceci revient à considérer les liaisons entre les objets comme des objets, eux-mêmes, qui représentent des évènements ou actions du monde réel. Pour l'exemple précédent on ferait :

ccatg (achat)
rel (acheteur(1,1),personne,(0,),achat)
rel (objet-acheté(1,1),voiture,(0,),achat)

On peut remplacer ces 3 déclarations par la déclaration unique suivante :

reln achat,acheteur(1,1) personne,objet-acheté(1,1) voiture nrel

Par définition de 'reln',le co-domaine des fonctions d'accès inv[acheteur] et inv[objet-acheté] est la catégorie 'achat' et les cardinaux min et max pour ces fonctions sont 0 et ∞.

Ces deux façons de déclarer la relation de propriété entre des 'personnes' et des 'voitures' peuvent co-exister. On peut déclarer :

ccatg (personne)
ccatg (voiture)
reln achat,acheteur(1,1),personne,objet-acheté(1,1),voiture nrel

et il faut naturellement ajouter à ces déclarations la sémantique suivante (2.5) :

1 - La personne x est propriétaire de la voiture y si existe un achat dont l'acheteur est x et l'objet-acheté est y.

Relations de degré quelconque

Pour définir une relation de degré n quelconque,on peut continuer à se servir des fonctions d'accès : on déclare une catégorie concrète avec le nom de la relation et n 'rel',ou alors on utilise,plus simplement,'reln' :

Par exemple, on peut vouloir donner plus de précisions sur les achats :

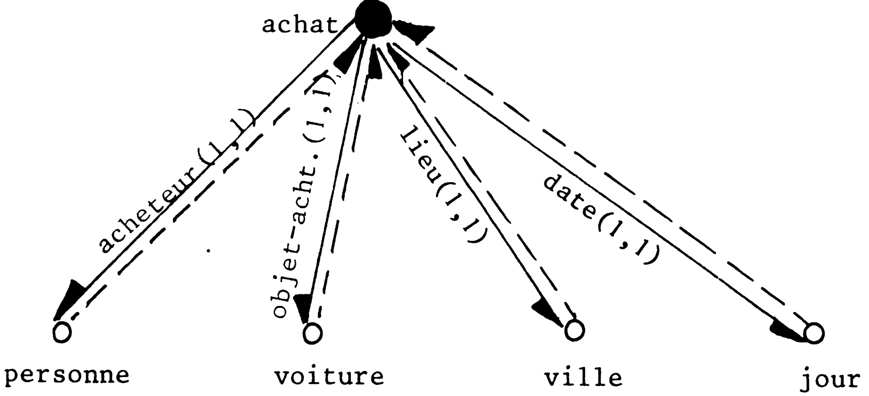
ccatg (achat)
rel (acheteur(1,1),personne,(0,),achat)
rel (objet-acheté (1,1),voiture,(0,),achat)
rel (lieu(1,1),ville,(0,),achat)
rel (date (1,1),jour,(0,),achat)

ou 'ville' et 'jour' sont deux catégories

De façon plus simple :

reln achat,acheteur(1,1) personne objet-acheté(1,1) voiture lieu(1,1) ville date(1,1) jour nrel

Cette même sémantique peut se représenter dans un graphe marqué :



qui exprime clairement le fait qu'on est en présence d'une relation 4-aire, dont chaque "instance" est elle-même un objet.

Le lecteur qui connaît Algol 68 a certainement fait le rapprochement d'une 'reln' avec la notion de 'structure'. En effet si, par exemple, on prend la déclaration de structure :

struct personne (string nom, int âge, ref person épouse)

on la traduit en termes de 'reln' de la façon suivante :

reln personne nom(1,1) identificateur age(1,1) entier époux(0,1) personne nrel

La seconde forme exprime, cependant, plus rigoureusement, la sémantique de 'époux', qui est tout à fait différente de la sémantique de 'nom' : une personne peut ne pas avoir d'époux' alors qu'elle a toujours un nom (même s'il est inconnu).

2.1.5. Conclusion

Le Modèle Sémantique est une méthode de description plus générale que le Relational Model (1.6) et tout d'abord parce qu'il permet de définir :

a) relations entre réalisations de relations

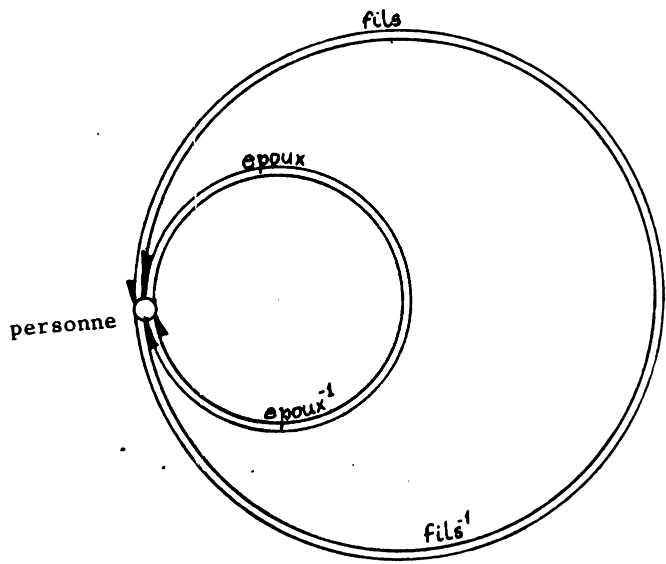
b) descriptions récursives

c) plusieurs relations entre les mêmes objets.

Comme exeple de a), il suffit de prendre la relation 'achat' qui a comme premier argument une personne, elle-même instance de la relation 3-aire 'personne'.

Comme exemple de b), on a le fait que l'époux d'une personne est encore une personne.

Enfin, on pourrait donner un exemple de c) en ajoutant au modèle la fonction d'accès 'fils' entre 'personne' et 'personne'.



Le Relational Model permet la définition de n relations entre 2 objets avec un artifice : on enregistre les objets n fois.

Ceci est, cependant, très coûteux, parce qu'il oblige à faire n-1 opérations supplémentaires de mise-à-jour pour chaque mise-à-jour, n étant le nombre de relations.

Le Relational Model ne donne pas la possibilité de déclarer des propriétés catégorielles, qui jouent le rôle de types de données structurés à partir de types "primitifs" que j'ai appelés "catégories"
Un Relational Model ne contient que des catégories (appelées "constituants".

Dans les travaux de Hoare [22] le lecteur peut trouver une présentation des types structurés. Dans ce travail, cependant, il y a, en plus, le produit cartésien de types, que je n'ai pas introduit pour ne pas avoir résolu tous les problèmes que poserait son introduction.

Enfin, une différence essentielle entre le Modèle Sémantique et le Relational Model est que, dans le premier, toute relation se définit à partir de fonctions d'accès alors que dans le deuxième elle se définit à partir de relations n-aires ($n \geq 2$), une relation n-aire étant définie comme un sous-ensemble du produit cartésien de n constituants.

2.3. OPÉRATEURS ÉLÉMENTAIRES

Je propose ensuite un jeu d'opérateurs basé sur les opérateurs du langage SOCRATE employés pour rentrer, interroger et mettre-à-jour les données.

En présentant ces opérateurs, je formalise aussi la sémantique des opérateurs SOCRATE.

Les erreurs que provoque l'application erronée des opérateurs [3] seront désignés ici des <u>échecs</u>, ou situations d'<u>échec</u>. Par exemple, si l'utilisateur veut faire la mise-à-jour :

m prénom <u>de</u> personne 1 = 20

ceci conduira à ECHEC

Pour chaque opérateur je décrirai les situations d'ECHEC possibles.

Si j'appelle cette sorte d'erreurs des ECHECS c'est parce que les ECHECs provoqués par la mauvaise application d'opérateurs ne sont que des cas particuliers de violation des règles sémantiques des données (0.3).

De façon analogue à Algol 68, par exemple, il y des opérateurs définis implicitement ("built-in") auxquels l'utilisateur peut ajouter des opérateurs définis par lui-même

Parmi les opérateurs implicites, il y a un certain nombre très réduit d'opérateurs, que je dirai élémentaires, et qui sont suffisants pour définir tout autre opérateur, implicite (par exemple : ccatg, acatg, rel,) ou non.

Je présente ensuite des opérateurs élémentaires.

2.3.1. Opérations sur les c-catégories

Génération : g

Si on a, par exemple, déclaré :

ccatg(personne)

et on veut engendrer un nouvel objet de cette catégorie, on se sert de l'opérateur 'g', auquel on passe la catégorie comme paramètre :

g personne

L'objet qui vient d'être "généré" portera, comme nom "historique", [I] un couple constitué par l'indication de la base de données où il a été "généré", et par sa date de naissance dans cette base (Intuitivement, le nom historique représente l'instant de la "connaissance" de l'objet par la base).

Si on veut récupérer, dans une variable, le nom historique d'un objet engendré on fera :

x1 : g personne [II]

et on pourra s'en servir dans le même contexte (par exemple, pour faire entre l'objet dans des relations).

[I] Un élément d'une catégorie concrète possède toujours deux noms : un nom historique et un indice : le nom historique, au contraire de l'indice, ne sera réaffecté à un autre objet, à la suite de la suppression de l'objet.

[II] Le résultat de l'opération "g personne " est affecté, par effet de bord, à la variable x1

Succès ou échec d'une génération

Si l'utilisateur fait, par exemple,

g voiture

et 'voiture' n'est pas une c-catégorie déclarée auparavant, le système répond :

ECHEC

Si la déclaration avait été faite, le système répond SUCCES

Suppression : s

Si, après un certain temps d'existence, on veut supprimer la personne 1 on se sert de l'opérateur s :

s personne 1

Si, pendant son existence, on a attribué un nom externe à la personne, qui est discriminant, on peut faire la suppression d'une autre manière :

s personne(jean,dupont)

notation que j'expliquerai en 2.4.3.

Succès ou échec d'une suppression

Si, pour une raison quelconque (par exemple, l'objet à supprimer n'existe pas, ou l'utilisateur n'est pas qualifié pour supprimer cet objet), la suppression est impossible, le système répondra :

'ECHEC'

suivi de la raison, sinon il répondra SUCCES.

Enumération : pour

Si on veut connaître toutes les personnes existantes à un moment donné, par exemple pour augmenter leur âge d'une unité, on se sert de l'opérateur

pour [I]

```
pour  x1 :∋ personne
      âge[x1] :∋ âge[x1] +1
fin
```

Ceci veut dire : la première personne (1er élément de la catégorie 'personne') est désormais contenue dans (repérée par) X1. On change l'âge de X1 et quand on arrive à 'fin' la boucle se relance automatiquement cherche la deuxième personne,qui devient alors celle repérée par X1, etc... etc... jusqu'à ce que l'on ait énuméré la totalité de la catégorie.

Echec ou Succès d'une énumération

Si, par exemple, la catégorie 'personne' n'existe pas (n'a pas été déclarée) ou si l'utilisateur n'est pas qualifié pour énumérer les 'personnes' ou si l'utilisateur ne peut énumérer qu'une partie des 'personnes', le système répond ECHEC qu'il fait suivre de la raison. Au contraire quand, et seulement quand, l'ensemble des 'personnes' a été énuméré complètement, le système répond SUCCES

Après 'fin', X1 ne contient rien.

Reconnaissance : ∋

Si on veut savoir si un objet x est d'une c-catégorie déterminée on se sert de l'opérateur '∋'. Par exemple :

X1 ∈ personne (intuitivement, "X1 est-il une personne ?")

On se sert souvent de la réponse du système (SUCCES ou ECHEC) pouer déterminer le reste du programme :

[I] La construction 'pour' peut se traduire de façon primitive comme j'indiquerai en page III.23

X1 personne
si échec
alors

etc...
fin

Ce qu'on écrira, souvent, plus simplement :

si X1 personne
alors

fin

En ce point, une remarque s'impose sur l'utilisation des notions de SUCCES et ECHEC comme résultats d'une reconnaissance: l'appartenance d'un objet à une catégorie (comme on verra en 2.3.4.4.) ceci est vrai pour des prédicats quelconques) est le fait des connaissances faillibles d'un utilisateur et, par conséquent, si elle est vraie pour celui-ci elle peut ne pas l'être pour un autre. SUCCES et ECHEC veulent dire, dans ce contexte, CONNU et INCONNU (implictement "du sujet en question"). 38 .

2.3.2. Opérations sur les a-catégories

De la définition même de catégorie abstraite (a-catégorie) découle que des 4 opérateurs vus (g, s, pour, ∈), seuls les deux derniers (pour, ∈) peuvent s'appliquer sur une a-catégorie.

Si l'utilisateur écrit :

ε entier

le système répondra ECHEC.

L'énumération et la reconnaissance seront utilisées de la même façon que pour les c-catégories. Ainsi on peut faire, par exemple,

x1 ∈ entier

2.3.3. Opérations sur les fonctions d'accès

S-affectation : :∋

Pour 's-affecter' (je dirai souvent simplement 'affecter') un objet y à l'extension d'un prédicat f[x] on écrira :

f[x] :∋ y

ce qui veut dire :

"l'extension du prédicat f(x) contient désormais y"

ou encore d'une façon "classique" :

"le prédicat f[x]est désormais vrai pour y".

L'opérateur ':∋' généralise l'affectation classique (:=).
En effet :

x := a

veut dire, dans la terminologie de Algol 68, que "x repère désormais a" et toute la différence entre ':∋' et ':=' est x := a veut dire exactement :

1) "x ne contient plus ce qu'il contenait"

2) "x contient désormais a"

Dans le cas des variables, la destruction implicite du contenu (1) est expliquée par la sémantique même du 'repérage'. Cependant, dans le cas de prédicats autres que les variables, cette sémantique n'est plus forcément valable
Ainsi, si on déclare :

prp(français(0,),personne)

et on fait :

français :∋ jean-dupont
français :∋ rené-durand

à la fin, il y aura deux éléments dans l'extension du prédicat 'français' [I]

[I] Le prédicat devrait être noté 'français [ω]' d'après ce que j'ai explique en page II.7. J'adopterai cependant la notation simplifiée 'français' parce que '[ω]' est implicite.

Les variables sont des 'prp' dont la fonction d'accès unique a max = 1. Par exemple la variable X1 est implicitement déclarée comme suit :

prp (X1 (0,1),objet)

Si par contre, on déclare une 'prp' quelconque qui n'est pas une variable, comme par exemple :

prp(président(1,1),personne)

on peut l'utiliser de façon analogue à une 'variable' :

président : a

mais si on écrit ensuite :

président : b

la réponse du système sera 'ECHEC' et 'a' est toujours le président.

La définition de variable que j'ai donnée recouvre naturellement la notion de variable en Algol 68. On peut en effet déclarer :

var (X,entier)

ou, si on veut, plus simplement :

entier X

Ensuite, on peut faire :

var(XX,prp-sur-entier) (I)

ou plus simplement :

prp-sur-entier XX (ref int en A68)

[I] prp-sur-entier est une 'prpcg' implicite.

Des S-affectations comme :

x :∋ a   où a est un entier

et

xx :∋ x   sont correctes.

L'analyse de la deuxième s-affectation révèle, cependant, une difficulté, quand on la compare avec une expression comme :

fils[y] :∋ x

Parce que, dans le premier cas, c'est la fonction d'accès 'x' qu'on affecte à xx alors que dans celui-ci, c'est le contenu de 'x' ——————————— qu'on affecte à 'fils [y]'. Dans le premier cas c'est un objet du second ordre qu'on affecte à xx, dans le deuxième cas c'est un objet du premier ordre. Or, dans ce langage où il y a des variables généralisées ($x_1, x_2, \ldots$) et où on travaille à un ordre ⩾ 1, une expression comme :

x_1 :∋ x

est ambigüe. Pour lever cette difficulté on peut utiliser juste avant 'x' le mot 'nom' qui indique justement que l'on parle de 'x' en tant que fonction d'accès et non du contenu de x.

L'expression :

xx:∋ x

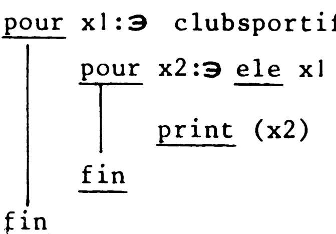
s'écrit donc correctement :

xx:∋ nom x

On utilisera aussi ele pour passer d'une variable à son contenu.
Par exemple :

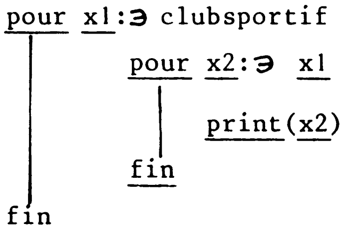
ele xx est la fonction d'accès x.

Si on veut définir un programme à un ordre 2 alors on a besoin de 'ele' :



a comme effet l'impression de toutes les personnes qui appartiennent à tous les clubs sportifs.

Cependant,



```
pour x1:∋ clubsportif
|      pour x2:∋ x1
|      |    print(x2)
|      fin
fin
```

aurait comme effet l'impression de tous les clubs sportifs, ce qu'on aurait pu obtenir plus simplement avec :

```
pour x1:∋  clubsportif
|     print(x1)
fin
```

Succès ou échec d'une s-affectation

L'affectation d'un objet à l'extension d'un prédicat peut conduire à un SUCCES ou ECHEC.

L'affectation :

$$f[x] :\ni y$$

sera un ECHEC si, par exemple :

a) f n'est pas déjà déclarée

b) x n'existe pas ou x $\notin$ dom[f]

c) le cardinal de f[x] est déjà égal à max[f]

d) y n'existe pas ou y $\notin$ co-dom[f]

e) l'utilisateur n'est pas qualifié pour changer f.

S-affectation multiple::⊃

D'après la définition de ':⊃', si on a déclaré :
prp(animal-rationnel(0,),personne)
le programme :

```
pour animal-rationnel :⊃ personne
|
fin
```

a comme effet d'affirmer que toutes les personnes sont 'animal-rationnel's. De façon simplifiée, on écrira :

rationnel :⊃ personne

ce qui veut dire que tout élément de l'extension du prédicat de droite devient élément de l'extension du prédicat de gauche.

Le SUCCES ou ECHEC d'une affectation multiple se définit selon des règles qu'on verra en page III.8 , en fonction du SUCCES ou ECHEC des s-affectations simples qui la composent. L'affectation multiple est une facilité de langage.

2.3.4.2. Désaffectation : :⊅

La désaffectation est l'opération par laquelle on enlève un objet y de l'extension d'un prédicat f[x].

Par exemple,

époux[a] :⊅ b

veut dire que b n'appartient plus à l'extension du prédicat 'époux[a]' ou encore que 'époux[a]' n'est plus vrai pour b.

Succès ou échec d'une désaffectation

Le SUCCES ou ECHEC d'une désaffectation se définit de façon analogue à l'affectation (page II.20). Les raisons a)b) et d) d'ECHEC sont toujours valables. La raison c) est remplacée par la règle suivante :

si le cardinal de f[x] devient < min[f] alors il n'y a pas d'ECHEC et f[x] possède désormais des éléments inconnus. Par exemple,

si 'jean' a les 'parent's 'marie' et 'antoine' (le prédicat "parent[jean]" contient les deux éléments 'marie' et 'antoine') et si on fait :

parent[jean] : marie

ceci ne produit pas d'ECHEC, et puisque min[parent] = 2, il y aura désormais dans "parent[jean]" un élément inconnu (u).

2.3.4.3. Enumération : pour

Si on veut énumerer l'extension d'un prédicat f[x] on écrit, comme pour les catégories ou prpcg :

```
pour x1:∋ f[x]
    print(x1)
fin
```

Le SUCCES ou ECHEC est déterminé selon des règles analogues à celles vues dans la page II.15 pour l'énumération des catégories. Naturellement, si f n'a pas été déclarée, x n'existe pas ou x n'est pas du domaine de f, on aura ECHEC.

2.3.4.4. Reconnaissance : '∈'

Pour reconnaître si un objet y appartient à l'extension d'un prédicat f[x] on écrit :

y ∈ f[x]

La réponse est SUCCES ou ECHEC, comme pour toute reconnaissance, et le SUCCES ou ECHEC sont déterminés de façon analogue à la page II.15.

2.3.5. Tableau récapitulatif des opérateurs élémentaires

opérateur	applicable sur
g	c-catégories
s	éléments de c-catégories
:∋	fonctions d'accès
:∌	fonctions d'accès (sauf les prpcg)
pour	catégories ou fonctions d'accès
∈	catégories ou fonctions d'accès

2.4. OPÉRATIONS SUR LES RELATIONS

Toute opération sur une relation de degré n, avec n ≥ 1, se définit comme une suite d'applications d'opérateurs élémentaires (2.3.) sur les fonctions d'accès et, éventuellement, la catégorie (page II.9) qui définissent cette relation.

2.4.1. Opérations sur les relations unaires ('prp')

Une 'prp' est constituée simplement par une fonction d'accès et, par conséquent, toute opération sur la 'prp' consiste simplement en une opération sur la fonction d'accès (:∋, :∌, pour, ∋)(page II.7) On en a déjà vu des exemples au § 2.3.3. Il y a cependant une exception pour les propriétés catégorielles : l'utilisation de ':∌' conduit à ECHEC, pour les raisons vues au § 2.1.2.1.

2.4.2. Opérations sur les relations binaires

Si une relation binaire est déclarée par une 'rel' (page II.7) on peut opérer dessus en opérant indifféremment sur l'une ou l'autre des deux fonctions d'accès, inverses l'une de l'autre. Par exemple, si on a déclaré :

rel(propriétaire(1,1),personne,propriété(0,),voiture)

et on veut mettre en relation une personne x avec une voiture y alors on peut faire indifféremment :

propriété[x] :∋ y (y est propriété de x)

ou :

propriétaire[y]:∋ x (x est propriétaire de y).

Dans un cas ou l'autre, le système se charge de garantir le maintien de cohérence

Pour effacer la relation entre x et y on peut de façon analogue, faire indifféremment :

propriété[x] :∌ y

ou alors :

propriétaire[y] :∌ x

et, comme avant, le système se charge du maintien de la cohérence entre fonctions d'accès inverses.

Si on veut énumérer les voitures qui sont en relation avec une personne x, on fait :

```
pour x1:∋ propriété[x]
    print (x1)
fin
```

Pour reconnaître si une voiture y est en relation avec une personne x on le fera indifféremment de l'une des deux façons :

y ∈ propriété (x)

ou alors :

x ∈ propriétaire(y)

Si, dans l'organisation de données choisie, seule <u>une</u> des deux fonctions d'accès ('propriétaire' ou 'propriété') est stockée (IV), le système ne fera pas le maintien de cohérence au moment de l'affectation sur une des deux fonctions d'accès.

Au lieu de ceci, quand il faut répondre à une question posée sur la fonction qui n'est pas stockée, il va convertir <u>de façon interne</u>, cette question dans une autre posée en termes de l'inverse et ceci de façon transparente à l'utilisateur. Ceci réalise la forme de data-indépendance demandée en 0.2.2.

Par exemple, si 'propriétaire' n'est pas sotckée, au moment de l'affectation :

propriété[x] :∋ y

le système ne fait rien de plus et quand on demandera :

x ∈ propriétaire [y]

le système exécutera :

<u>si</u> y ∈ propriété [x]
<u>alors</u> succès
<u>sinon</u> échec
<u>fin</u>

Supposons maintenant que la relation de propriété avait été déclarée comme suit :

<u>ccatg</u> (achat)
<u>rel</u> (acheteur(1,1),personne,(0,),achat)
<u>rel</u> (objet-acheté(1,1),voiture(0,),achat)

Si maintenant, on veut mettre en relation la personne x et la voiture y il faut faire :

x1 :∋ g achat
acheteur[x1]:∋ x
objet-acheté[x1]:∋ y

Si on utilise '<u>reln</u>' pour déclarer la relation, on peut écrire, plus simplement, <u>g</u> achat (x,y).

Si on veut énumérer toutes les voitures de x, on fera :

```
pour x1:∋ inv[acheteur] [x]
|   print(x1)
fin
```

Pour reconnaître si x est en relation avec y on fera :

existe x1:∋ inv[acheteur][x] ∩ inv[objet-acheté] [y] [I]

ou alors, plus simplement

existe x1:∋ achat(x,y)

Si on veut détruire la relation entre x et y on fera :

s achat(x,y)

2.4.3. Opérations sur des relations de degré supérieur à 2

Les opérations à faire pour établir une nouvelle réalisation ("instance", en anglais) d'une relation de degré > 2 sont les mêmes que pour une relation binaire et seul leur nombre change. Par exemple, dans le cas de la relation "achat"

reln achat,acheteur(1,1) personne,objet-acheté(1,1) voiture,lieu(1,1) ville,
date(1,1) jour nrel

si on veut établir la relation 'achat' entre les objets 'jean','vw','grenoble' et '1er mars', on ferait :

```
x1:∋ g achat
acheteur[x1] :∋ jean
objet-acheté[x1] :∋ vw
lieu[x1] :∋ grenoble
date[x1] :∋ 1er-mars
```

[I] La sémantique de 'existe' sera expliquée en page II.37.
L'expression "existe x1: <ensemble>" veut dire que si existe un élément dans l'ensemble, cet élément sera affecté à x1 par effet de bord.

ou plus simplement :

g achat (jean,vw,grenoble,1-mars)

Pour reconnaître si 'antoine' a acheté une 'fiat' à 'grenoble' le '2-janvier' on fait :

<u>existe</u> **x1** :∋ achat (antoine,fiat,grenoble,2-janvier)

Si on veut supprimer l'achat qu'on a engendré en haut, on fait:

s achat (jean,VW,Grenoble,1-mars)

Si on veut énumérer tous les achats faits par 'jean' on écrira :

```
pour x1:∋ inv[acheteur][jean]
|
|    print(x1)
fin
```

2.5. MODÈLE SÉMANTIQUE COMPLET

Une fois terminée la définition du Modèle Sémantique, c'est-à-dire une fois déclarées les catégories, prpcg et relations, on transforme celui-ci en Modèle Sémantique Complet par l'ajout de règles sémantiques ad-hoc des différentes sortes vue en 0.3.

Pour cela, j'ai besoin de passer de la notation adoptée dans le paragraphe 2.4., pour les opérations élémentaires, à la notation suivante :

notation 2.2	nouvelle notation
g <catégorie>	g (<catégorie>)
s <objet>	s (<objet>)
<prédicat FX> :∋ <objet>	:∋ (<prédicat FX>,<objet>)
<prédicat FX> :∌ <objet>	:∌ (<prédicat FX>,<objet>)
<objet> ∈ <prédicat FX>	(<prédicat FX>, <objet>)
<u>pour</u> x :∋ <prédicat> | <u>fin</u>	<u>pour</u> x :∋ (<prédicat>) | <u>fin</u>

Par exemple,

g personne
s personne (jean, dupont)
<u>pour</u> x1:∋ personne
|
<u>fin</u>

x1 ∈ personne
antoine ∈ époux [marie]
époux[marie] : jean

deviennent, avec la nouvelle notation,

g(personne)
s(personne(jean , dupont))
pour x1 : (personne)

fin
(personne,jean)
(époux[marie], antoine)
(époux[marie],jean)

2.5.1. Règles sémantiques ad-hoc : METHODES

La forme que je donnerai aux règles sémantiques ad-hoc est celle de METHODES. Je vais définir cette notion.

Reprenons l'exemple des produits et bénéfices que j'ai exposé en page 0.12.

J'avais dit alors que la règle sémantique qui dit que le bénéfice d'un produit est la différence entre son prix de vente et son prix d'achat devrait être stockée et utilisée par le système pour répondre aux questions sur les bénéfices des produits.

L'énumération de l'extension d'un prédicat 'bénéfice[x]', où x est un produit, devra donc consister dans les trois opérations suivantes :

- accès au prix-de-vente de x
- accès au prix-d-achat de x
- soustraction des deux valeurs, suivie du retour du résultat.

et ceci s'obtiendra en définissant la METHODE suivante :

```
mth((pour,bénéfice),,,faire
                        | x2 :∋ prix-de-vente[x1]
                        | x3 :∋ prix-d-achat[x1]
                        | return (x2-x3)
                        fin)
```

L'exécution de ce programme sera faite de façon transparente à l'utilisateur, chaque fois que celui-ci écrira :

```
pour y :∋ bénéfice[x]
|    print(y)
fin
```

L'autre manière que j'avais indiquée d'exprimer la même règle sémantique consisterait à définir la méthode (pour,bénéfice) comme suit :

```
mth((pour,bénéfice),,,faire
                        | x2 :∋ prix-de-vente[x1]
                        | x3 :∋ prix-d-achat[x1]
                        | x4 :∋ x2-x3
                        | :∋ (bénéfice[x1],x4)
                        | return(x4)
                        fin
```

Dans ce cas, quand on a demandé une fois le bénéfice d'un produit, les fois d'après il suffit de lire sa valeur puisque celle-ci a été stockée après sa déduction lors de la première fois. Pour cela,on écrira :

```
pour y :∋ (bénéfice[x])
|    print(y)
fin
```

qui représente l'opération élémentaire d'accès au bénéfice.

L'emploi de ces deux formes rend le langage dépendant du fait que le bénéfice est déduit ou lu, ce qui présente les inconvénients exposés en page 0.7.

Une manière d'éliminer ces inconvénients serait de définir la méthode :

```
mth((pour,bénéfice),,,faire
                   | x4 :∋ (bénéfice[x1])
                   | si échec
                   | alors x2 :∋ prix-de-vente[x1]
                   |   |   x3 :∋ prix-d-achat[x1]
                   |   |   x4 :∋ x2-x3
                   |   |   :∋ (bénéfice[x1],x4 )
                   | fin
                   | return(x4)
                 fin)
```

qui veut dire : en cas d'échec de la lecture du bénéfice déduire celui-ci le stocker et le retourner, sinon retourner le bénéfice lu.

Je voudrais que le lecteur remarque la différence entre l'expression :

```
pour  y :∋ bénéfice[x]          et          pour y :∋ (bénéfice[x])
|   print( y)                               |          print(y)
fin                                         fin
```

la première est à un niveau plus haut puisqu'elle consiste dans un certain nombre d'opérations élémentaires du genre des questions SOCRATE alors que la deuxième est au niveau de celles-ci.

D'une manière générale, l'utilisateur peut définir une méthode pour chaque opérateur et chaque catégorie prpcg ou fonction d'accès. (en respectant évidemment les contraintes indiquées dans le tableau 2.3.5.

Si pour l'opérateur et la catégorie, prpcg ou fonction d'accès, il y a déjà une autre méthode définie, alors il faut qu'il donne un nom à la nouvelle méthode. La méthode sans nom sera appelée méthode standard et ce sera la méthode exécutée à moins que l'utilisateur indique un nom de méthode voulue.

Une méthode a deux sortes de paramètres : des paramètres obligés que j'appellerai implicites,qui sont l'objet à supprimer pour les méthodes de suppression, les objets à mettre en relation pour les méthodes de s-affectation et de désaffectation, etc..., et des paramètres explicites dont il faut déclarer les catégories ce qui se fera en 2ème ou 3ème position (cf page II.37) selon qu'il s'agisse de paramètres de retour ou de paramètres d'entrée.

2.5.2. Exemples

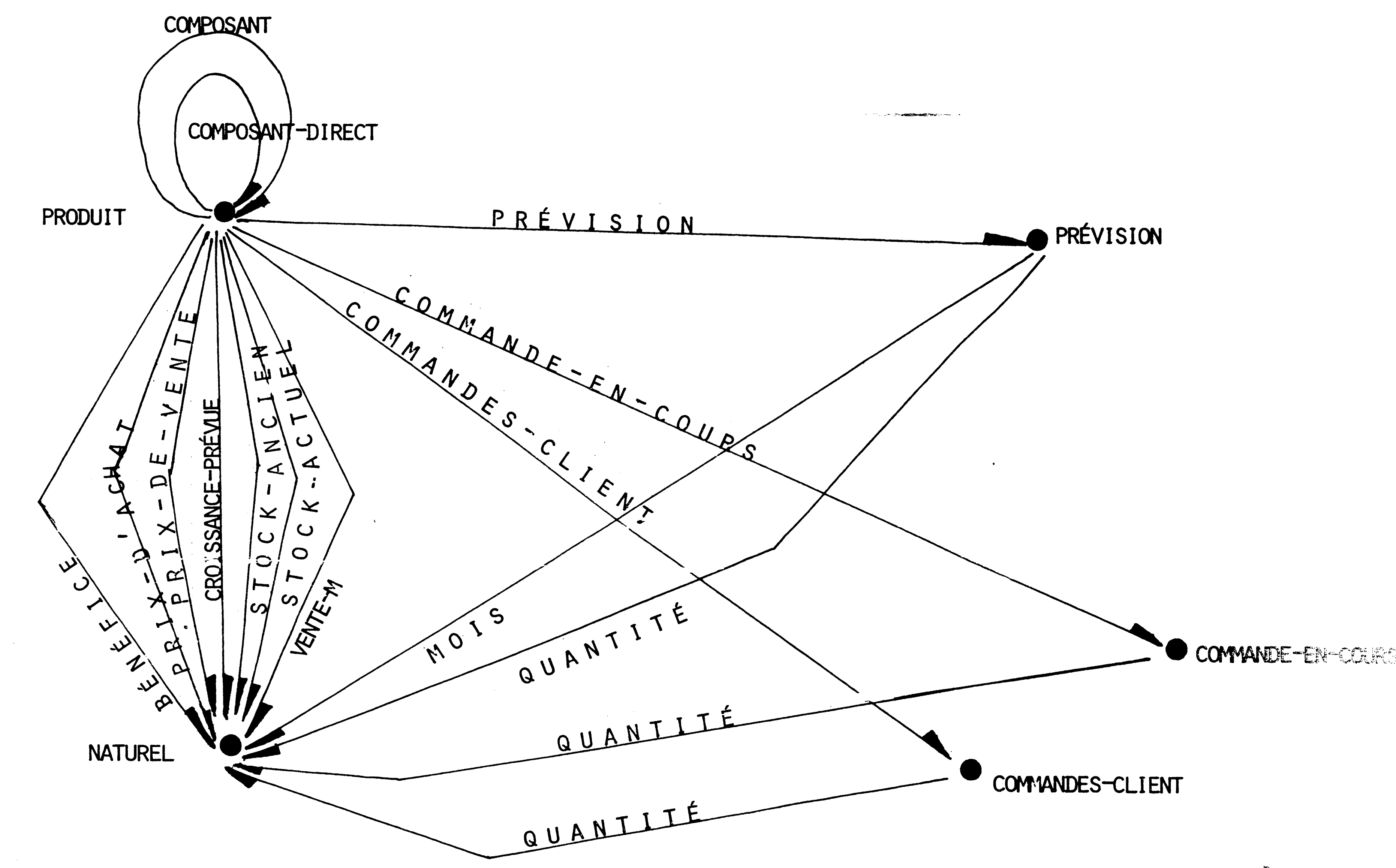
2.5.2.1. Modèle Sémantique Complet de l'exemple de Gestion (paragraphe 0.3.3.)

Le Modèle présenté informellement en page 0.16 se traduit par le Modèle Sémantique que je reproduis en page II.33 et les METHODES que je donne ensuite.

```
mth((pour,vente-m),,,faire
                       x2 :∋ stock-ancien[x1]
                       x3 :∋ quantité-reçue[x1]
                       x4 :∋ stock-actuel[x1]
                       x6 :∋ (x2+x3)-x4
                       :∋ vente-m ([x1],x6)
                       return(x6)
                    fin)
```

La méthode d'accès à la vente-m(vente du mois m) d'un produit consiste à :

- déduire sa valeur à partir des stocks ancien et actuel du produit et de la quantité reçue,
- affecter la valeur à l'extension du prédicat 'vente-m[x]'.



[I] commandes passées à un fournisseur

```
mth((:∋,vente-m),,,faire
          :∋ (vente-m[x1],x2)
          pour x3 :∋ prévision[x1]
               s x3
          fin
          x4 :∋ 1
          faire
               x4 :∋ x4+1
               g prévision (x1,x2,x4)
               si x4=1
               alors g commande (x1)
               fin
               si x4 ≠ 3
               alors up        [I]
               fin
          fin
   fin)
```

L'affectation d'une valeur x à l'extension du prédicat 'vente-m[y]' entraîne, par effet de bord, l'effacement des prévisions concernant le produit y et la "génération" de nouvelles prévisions.

```
mth((g,prévison),,(produit,naturel,naturel),faire
                                    x4 :∋ g(prévision)
                                    x5 :∋ croissance[x1]
                                    x6 :∋ x2 + x5 × x3
                                    :∋ quantité ([x4],x6)
                                fin)
```

[I] 'up' veut dire "recommencer l'exécution du bloc"

Dans cette méthode, en 3ème position on a déclaré les catégories des 3 paramètres explicites (page II.32).

La "génération" d'une prévision entraîne le calcul de la quantité correspondante.

Je pourrais définir de façon analogue la méthode (g,commande) la méthode (:∃,stock-actuel), etc...
La définition de la méthode (g,commande-client) pourrait être

```
mth((g,commande-client),,(produit,naturel),
    faire
          si x2 > stock-actuel[x1]
          alors échec fin
          x3 :  g(commande-client)
          x4 :  stock-actuel[x1] - x2
          :   stock-actuel([x1],x4)
    fin)
```

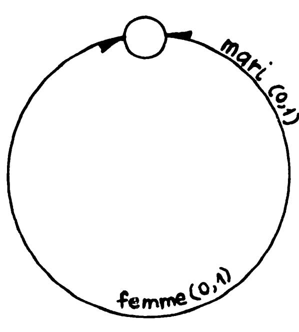
La structure de cette méthode est un bon exemple des notions exposées en 0.3.2 :

- l'opération proprement dite de génération de la commande est précédée d'une validation sémantique ad-hoc (je donnerai en page III.14 une définition plus réaliste de cette méthode).
- comme conséquence logique de la "génération" de la commande, le stock du produit diminue.

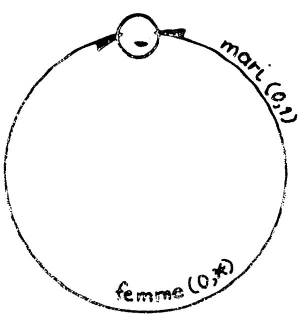
2.5.2.2. Modèles sémantiques non-implémentables en SOCRATE - Redondances - Relatio n-aires

J'ai caractérisé au paragraphe 1.5.1 les Modèles Sémantiques qu'on ne peut pas implanter avec une structure SOCRATE.

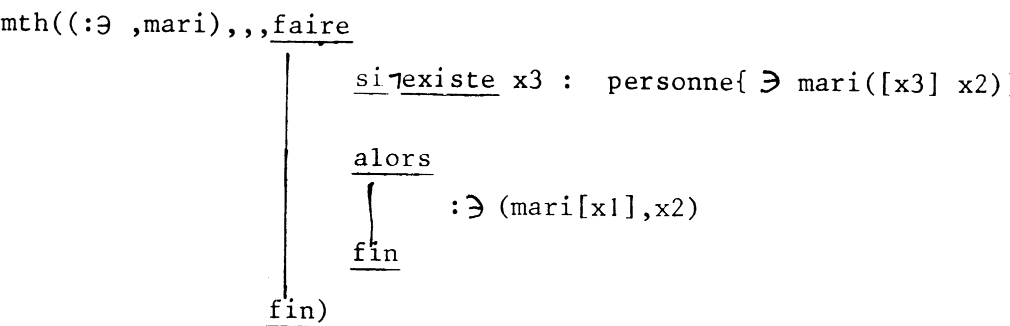
J'avais donné, par exemple, le modèle :



Or, ce modèle peut être implanté si on choisit une organisation de données adéquate qui implante :



et on ajoute au modèle la méthode :



```
mth((:∋ ,mari),,,faire
        si ¬existe x3 :  personne{ ∋ mari([x3] x2)}
        alors
            :∋ (mari[x1],x2)
        fin
fin)
```

ce qui empêche qu'il y ait deux affectations comme suit

mari[personne 1] = personne 2

mari[personne 5] = personne 2

et garantit donc que 'femme' ' est de type (0,1).

De manière générale, tous les modèles non-implémentables en SOCRATE (1.5.1.) peuvent être réalisés de façon analogue.

Redondances

Reprenons l'exemple donné en page I.63.

J'ai alors indiqué que la redondance apparente entre le fichier inversé ('divorcé') et les données sur l'état-civil n'existait pas parce que SOCRATE ne garantit pas la règle:

"l'état-civil d'une personne est 'divorcé' si et seulement si elle est 'divorcée'"

Or cette règle de cohérence s'exprime facilement en ajoutant les deux méthodes suivantes au Modèle Sémantique :

```
mth((:∋, état-civil),,,faire
        :∋ (état-civil [x1],x2)
        si x2 = divorcé
        alors :∋ divorcé(x1)
        fin
fin)
```

```
mth((:∋,divorcé),,,faire
                   | :∋ (divorcé,x1)
                   | :∋ (état-civil[x1] ,divorcé)
                   fin)
```

Elles garantissent que chaque fois qu'une personne est mise dans le fichier inversé, son état-civil sera mis-à-jour automatiquement et inversement.

Pour être rigoureux, il fallait définir aussi deux méthodes de désaffectation exactement similaires.

Contraintes sur les relations n-aires (page I.60)

Les relations n-aires se déclarent à l'aide de 'reln'
Leurs méthodes de "génération" ont, en général, comme paramètres explicites les n arguments de la réalisation ("instance") engendrée.

Ainsi, dans le cas de

```
reln  fourniture,fournisseur(1,1) personne,pièce-finie(1,1) pièce,client(0,1)
      personne nrel
```

la méthode de génération aura 3 paramètres qui sont le fournisseur, le produit fourni et le client. La "génération" d'une fourniture s'écrira, par exemple :

g fourniture (personne 3, moteur, personne 6)

La contrainte exposée en page I.61 (un produit ne peut pas être fourni par deux fournisseurs différents au même client), s'exprime dans la méthode de génération des fournitures.

```
mth((g,fourniture),,(personne, produit,personne)faire
                              | si existe x4 : ∋fourniture
                              |    | produit-fourni [x4] x2 ∧
                              |    | client [x4] = x3 ∧
                              |    | fournisseur [x4] ≠ x1
                              | alors échec
                              | fin
                              | x5 :∋ g(fourniture)
                              | :∋ fournisseur([x5] ,x1)
                              | :∋ produit-fourni([x5] ,x2)
                              | :∋ client([x5] ,x3)
                              fin)
```

Une remarque : la méthode n'empêche pas qu'il y ait plusieurs fournitures différentes d'un produit par un fournisseur à un client.

De manière générale, les contraintes sur des relations n-aires exposées dans les travaux de E. F. CODD [7] ne sont qu'une très faible partie de toutes celles qu'on peut définir à travers les méthodes.

2.5.3. Changements de méthode

Une définition de méthode est constituée par ce que j'appellerai sa déclaration, représentée par le triplet (<opérateur>, <catégorie> ou <fonctions d'accès>, <nom>) les deux listes de catégories des paramètres explicites et enfin par un programme.

Si on veut remplacer une méthode par une autre il suffit de déclarer la deuxième de la même façon que la première.

Tous les programmes sémantiques restent valables après ce remplacement.

Ainsi, si on voulait remplacer la méthode (pour,bénéfice) définie dans le MSC de l'exemple de Gestion par une autre, -------- , il suffit de définir celle-ci de la façon habituelle. Le système se chargera de remplacer l'ancien programme par le nouveau. Tous les programmes contenant l'expression "bénéfice[x]" restent valables. Lors de leurs nouvelles exécutions, l'accès au bénéfice se fera par la nouvelle méthode.

Une autre façon de changer une méthode c'est supprimer ou ajouter des instructions, ce qui se fait avec l'opérateur :∋ parce qu'un programme est considéré à l'intérieur du système comme un cas particulier de 'prp'.

Je ne m'attarderai plus sur ce type de changements de méthodes parce que, à l'heure actuelle, je n'ai pas résolu les problèmes qu'ils posent.

2.6. Conclusion

Les METHODES ressemblent à des procédures avec un mécanisme d'appel particulier : l' "appel" d'une METHODE ne se fait pas comme l'appel d'une procédure.

Si on déclare en Algol 68 [23]

<u>proc</u> (<u>real</u>,<u>int</u>) <u>real</u> toto

l'appel de la procédure se fera par son nom ('toto') alors que si on déclare la méthode :

<u>mth</u>(pour,bénéfice déduction)

son 'appel' sera fait implicitement (par le système-) quand il faudra appliquer l'opérateur 'pour' à un prédicat bénéfice[x].

Je pourrais peut-être caractériser la différence entre les mécanismes d'appel des procédures et méthodes en disant qu'une procédure est appelée "par son nom" et qu'une méthode est appelée "<u>par ce qu'elle fait</u>".

Dans le cas particulier des méthodes dont la déclaration ne contient que le premier paramètre (l'opérateur), la réalisation de l'appel est équivalente. La déclaration d'une méthode équivalente à la procédure ALGOL 'toto' serait :

<u>mth</u> (toto), <u>reel</u>, (<u>reel</u>, <u>entier</u>)

et l'emploi de la méthode est équivalent à celui de la procédure Algol. Toute procédure peut ~~donc~~ être traduite en méthode.[I].

La notion de 'méthode' a été proposée par J.R. ABRIAL après une étude que l'on a faite sur la notion de "theorem" [II], introduite par le langage PLANNER [35] et très utilisée par WINOGRAD [34].

La définition de "theorem" est constituée par :
- sa déclaration qui consiste dans un n-uplet ($n \geqslant 1$) d'objets et variables, qui s'appelle son "<u>pattern</u>"
- un programme écrit en PLANNER

[I] Je profite de l'occasion pour expliquer que les opérateurs g, s, ... ne sont pas autre chose que des méthodes internes déclarées par <u>mth</u>(g), <u>mth</u>(s) etc... déclarations avec la même forme que <u>mth</u>(TOTO).

[II] Il ne faut pas traduire "theorem" par théorème... Le lecteur verra pourquoi.

Le triplet (<opérateur>,<catégorie> ou <fonction d'accès>,<nom>) joue dans une méthode le rôle que joue le "pattern" dans un "theorem". Cependant le mécanisme d' "appel" des "theorems" est plus puissant que celui des 'méthodes'.

J'explique ensuite en quoi.

Reprenons l'exemple des produits et leurs composants traité en

Si je veux demander :

composant (b , a)

(prouver que a est composant de b) après avoir défini la méthode

```
mth((∃,composant),,,faire
                    | ∃composant-direct([x1] x2)
                    |
                    | existe x3 :∃ composant-direct [x1]     { composant([ x3] x2)}
                    fin)
```

qui traduit la 2ème forme de la règle sémantique sur les composants de la page 0.11 cette méthode sera "appelée" pour faire la preuve.

En PLANNER, l'énoncé de la preuve s'écrit

(goal(composant b a))

et le système va chercher d'abord dans la base de données pour essayer de trouver le triplet '(composant b a)' auquel cas la preuve réussi. S'il n'y trouve pas ce triplet alors il "appelle" un "theorem" dont le "pattern" soit conforme au triplet. Un "pattern" est conforme au triplet s'il est :

(composant ?x1 ?x2)

où '?' indique que x1 et x2 sont des variables, ou encore :

(?y ?x1 ?x2)
(composant ?x1 a)
(composant b ?x2)
(?y ?x1 a)
(?y b ?x2)

Or, un "thoerem" avec le premier "pattern" est équivalent à la méthode puisqu'il sert à prouver qu'un produit x2 est composant d'un autre x1.

Par contre, si le "pattern" est un des autres, on ne peut pas donner une définition équivalente de méthode, puisqu'il sert à prouver qu'une relation quelconque y relie x1 à x2, qu'il existe un produit dont a est composant, qu'il existe un composant de b, qu'il existe une relation reliant un bojet quelconque à a, ou enfin qu'il existe une relation quelconque reliant b à un autre objet.

Ce principe d'appel que j'ai présenté dans le cas d'une preuve s'applique à toutes sortes de "theorems", et il est donc plus puissant que le mécanisme d' "appel" des méthodes.

Il introduit cependant une sorte d' "ambiguité" ou non-déterminisme quant au "theorem" employé puisqu'il y a en général plusieurs "theorems" dont les "patterns" sont conformes, ce qui, à mon avis, rend le contrôle de la correction des programmes très difficile à faire (Partie IV).

Données déduites

PLANNER lit toujours la base de données avant de faire une déduction. Au cas où il faut faire celle-ci, il stocke la donnée déduite.

Une solution analogue est aussi employée par SYNTEX [11] sauf qu'il n'y a pas de stockage.

Ceci me paraît mauvais pour deux raisons : trop contraignant vu sous l'angle de la diminution du coût de l'accès parce que, selon l'organisation des données, il peut être plus rapide de déduire que de lire une donnée.

Ensuite, vu sous un angle logique, lire et, en cas d'échec, déduire et stocker équivalent à déduire. Reprenons l'exemple de bénéfice. En effet, si) un instant t_0, on déduit le bénéfice d'un produit et on le stocke et si à un instant t_1 après t_0 on lit la valeur stockée, celle-ci peut ne plus être la différence entre le prix-de-vente et d'achat du produit. Il suffit pour cela qu'une de ces deux valeurs ait été mise-à-jour entre temps. La sémantique n'est donc pas respectée.

Par contre, si on déduit à chaque fois le bénéfice on est sûr de respecter la sémantique.

Les méthodes donnent cependant le moyen de rendre équivalentes ces deux solutions. Il suffit d'ajouter une méthode (:$\not\ni$,prix-d-achat) et une méthode (:$\not\ni$,prix-de-vente) qui mettent à jour le bénéfice chaque fois que ces deux quantités sont mises-à-jour. Evidemment le gain en efficacité finit par se perdre.

En CONNIVER on trouve une solution analogue à celle que j'ai adoptée.

Dans partie 3 (CONCLUSION) j'étudie plus longuement la sémantique des différentes sortes de méthodes comparées avec les différentes sortes de "theorems" disponibles en PLANNER.

PARTIE 3

LANGAGE SEMANTIQUE

3.1. INTRODUCTION

Je présente, dans cette PARTIE, le Langage Sémantique, c'est-à-dire le langage dans lequel l'utilisateur interroge et met à jour les données et dans lequel on définit aussi les METHODES

Reprenons l'étude faite en I. 3 sur le langage d'interrogation et mise-à-jour SOCRATE.

En SOCRATE, interroger ou mettre-à-jour une donnée consiste essentiellement à accèder et soit à l'imprimer, soit à la mettre-à-jour.

On avait vu que l'accès à une donnée ("citation" en SOCRATE) peut être :

- "simple" c'est-à-dire qu'il consiste à énumérer l'extension d'un prédicat (par exemple "i mère de personne 1)
- avec connecteurs (et, ou, ¬,) si le prédicat "énumère" est l'intersection, union ou complément d'autres prédicats (par exemple : "i personne ayant prénom = jean et age = 20)
- "composé" si le prédicat "énuméré" correspond à la composition de plusieurs fonctions d'accès f_1 f_2 ... f_n (par exemple "i mère de personne ayant père = jean")

Le langage SOCRATE possède aussi des mécanismes qu'on trouve dans les langages algorithmiques (ALGOL, FORTRAN, etc...) qui sont les blocs (avec cependant deux commandes - SORTIE et REFAIRE - à la place du GOTO et sans déclaration de variables locales) les conditions (à l'intérieur desquelles on peut faire usage des quantificateurs $\forall$ et $\exists$ bien qu'il soit limité [I]) et les itérations (une itération porte toujours sur les éléments d'une c-catégorie). Les connecteurs et, ou et ¬ employés dans les conditions jouent le rôle des connecteurs $\wedge$ $\vee$ et ¬ de la logique.

[I] Je parle de "limité" vis-à-vis de l'emploi de ces quantificateurs dans un langage formel comme le calcul de prédicats.

En SOCRATE on peut, enfin, définir des procédures analogues aux procédures ALGOL.

Une procédure SOCRATE est une suite d'interrogations et mises-à-jour. ("requêtes")

Dans le Langage Sémantique, au lieu des interrogations et mises-à-jour SOCRATE, il y a des énoncés de problèmes. Les opérations élémentaires, qui sont équivalentes aux "requêtes", sont vues comme des énoncés de problèmes d'un type particulier que j'appellerai "élémentaires".

Ceci se justifie par le fait qu'une expression comme

```
pour x1 :∋ bénéfice[a]
|
|
fin
```

donnera lieu à la recherche et à l'application d'une méthode (p. II.31) et ne se traduira pas par un simple accès aux données.

La méthode appliquée, que je dirai "la méthode de résolution" du problème est elle-même constituée, en général, par des énoncés de problèmes qui ne sont pas élémentaires.

Les programmes sémantiques ne sont pas vraiement des programmes mais plutôt des méthodes de résolution de problèmes : si la résolution d'un sous-problème du problème termine en ECHEC, cet ECHEC peut être récupéré.
peut être récupéré.

La programmation en Langage Sémantique est, par conséquent, une programmation non nécessairement algorithmique

Les caractéristiques du Langage Sémantique induisent entre autres, une structure de contrôle du type de celle des Langages d'Intelligence Artificielle [35][39][40][41] , plus flexible que celle des langages classiques (ALGOL, SOCRATE, etc...).

Elle est caractérisée par deux mécanismes.

Le premier que j'appellerai "pseudo-parallélisme" permet d'activer plusieurs méthodes en même temps et de gérer les "processus" correspondants, qui peuvent être endormis, sans être supprimés, et réveillés tour à tour

Le deuxième (appelé "backup" dans la littérature anglaise) permet de revenir sur un choix fait au cours de la résolution d'un problème, s'il s'est révélé mauvais. Ce mécanisme n'est pas automatique comme celui de PLANNER 35

Puisque la résolution d'un problème change généralement les données revenir sur un choix oblige à effacer les changements intervenus après ce choix. Pour cela, j'introduis le concept d' "espace".

Ce concept est similaire à celui de "data base context" employé en QA4 [41] et CONNIVER [40]

3.2. ÉNONCÉ D'UN PROBLÈME

La forme de l'énoncé d'un problème se distingue de celle d'un problème élémentaire (page II-28) par la position des parenthèses.

Ainsi, l'exemple sur le bénéfice (page II.31)

Un problème élémentaire est un problème dont la méthode de résolution n'est pas spécifique d'une catégorie ou fonction d'accès [I] par conséquent il ne faut pas indiquer celle-ci. Par contre elle figure comme paramètre, entre parenthèses, puisque justement la méthode est uniforme.

[I] C'est pour cela que la déclaration de cette méthode ne contient que l'indication de l'opérateur (pour) et fait partie du système.

S'il y a plus d'une méthode pour résoudre un problème, on a le droit d'indiquer son nom dans l'énoncé. dans l'énoncé.

Par exemple,

<u>pour</u> x : bénéfice,déduit y
<u>fin</u>

et le problème sera résolu par cette méthode là et non pas par la standard.

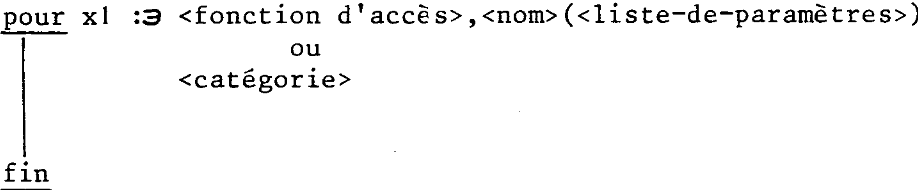
Si la résolution d'un problème est accompagnée de passage de paramètres (c'est le cas des "énumérations", par exemple) alors on ajoute un signe : ∋ et la (les) variable(s) de retour à gauche.

x1 :∋ g commande-client(moteur,20)

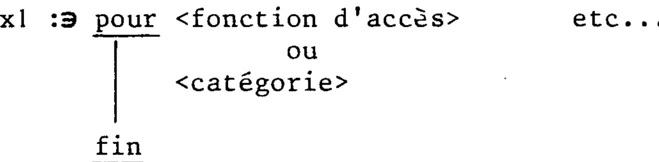
ce qui veut dire que la commande-client engendrée est retournée et affectée par effet de bord, à x1.

Dans le tableau de la page suivante, je présente toutes les formes possibles d'énoncés de problèmes non élémentaires.

Il faut remarquer que :



est une écriture plus facile de



D'ailleurs la boucle 'pour' est une écriture simplifiée. La traduction dans des constructions primitives est expliquée en

FORME DES ENONCES

g<catégorie>,<nom> (<liste-de-paramètres>)

s<catégorie>,<nom> (<liste-de-paramètres>)

:∋<fonction d'accès>,<nom> (<liste-de-paramètres>)

:∌<fonction d'accès>,<nom> (<liste-de-paramètres>)

∋ <fonction d'accès>,<nom> (<liste-de-paramètres>)
ou
<catégorie>

---

pour <fonction d'accès>,<nom> (<liste-de-paramètres>)
ou
<catégorie>

3.2.1. **Preuves**, énumérations et transformations

Pour des raisons d'exposé, j'appellerai désormais preuves aux problèmes dont les énoncés commencent par '∍' ou existe (cf. 'reln'page II.27), énumération si l'énoncé commence par 'pour' et enfin, transformations à ceux dont l'énoncé commence par g, s, :∍ ou :∌. J'explique pourquoi.

Une preuve consiste à démontrer une proposition qui représente, pour les preuves vues, jusqu'ici, l'appartenance d'un objet à l'extension d'un prédicat (∍) ou l'existence d'une relation entre n objets (existe) (en 3.2.2.2.) on verra des formes plus compliquées).

Une énumération consiste dans la déduction des éléments de l'extension d'un prédicat.

Les preuves et énumérations, en général, ne changent pas les données.

Une transformation est un problème dont la résolution consiste à changer les données afin de rendre une proposition vraie, changement pouvant intervenir au niveau de la validation de la transformation (page III.14) et au niveau des effets de bord qu'elle provoque.

Je vais donner un exemple pris en Intelligence Artif. [34]que je présenterai complètement plus tard.

Il s'agit d'une base de données qui décrit un environnement réel constitué par des cubes sur une table

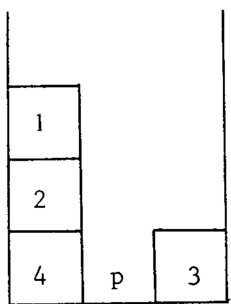
Le fait qu'un cube x est sur un cube y est représentée par "x ∊ sur[y]".

L'affectation suivante :

:∍ sur([cube 1] cube 2)

est une transformation : elle consiste à changer les données afin de rendre vraie la proposition "cube 2 ∊ sur [cube 1]", ceci étant fait de telle façon que l'état de ces données soit, à chaque instant, la reproduction d'états cohérents de l'environnement constitué par les cubes.

Supposons que, à un certain moment, les données représentent l'état des cubes :



La transformation :

:∋ sur ([cube 1] cube 2)

doit être précédée des trois transformations plus simples :

:∋ lieu ([cube 1] p) [I]
:∋ sur ([cube 3] cube 2) [II]
:∋ sur ([cube 4] cube 1) [III]

pour que l'état final soit atteint.

I :∋ lieu ([cube 1],p) consistera elle-même dans les deux transformations plus simples :

:∌ sur ([cube 2],cube 1)

et :∋ (lieu [cube 1],p) qui est élémentaire

II ":∋ sur ([cube 3]cube 2)" consiste elle-même dans les deux transformations plus simples :

:∌ sur ([cube 4], cube 2)

:∋ (sur[cube 3], cube 2), qui est élémentaire.

III :∋ sur ([cube 4] cube 1) consiste elle-même dans les deux transformations

:∌ lieu ([cube 1]p)

:∋ (sur[cube 4], cube 1)

3.2.2. Enoncés composés

Souvent, au lieu d'énoncer un seul problème, on a besoin d'énoncer une suite de problèmes.

Une suite de problèmes constitue un bloc. Un bloc peut avoir un nom.

Un bloc a la forme générale suivante :

```
faire
|    E1
|    E2
|    :
|    Ei
|    :
|    En
fin
```

où E_1, E_2,...,E_i sont des énoncés, éventuellement des blocs d'énoncées, eux aussi.

3.2.2.1. Transformations composées

J'appellerai "transformation composée" un bloc constituée par des énoncés de transformations.

Ainsi, par exemple :

```
faire (toto)
|     :∋ sur table (cube 2)
|     :∋ lieu ([cube 1], p)
fin
```

est une transformation composée par les deux transformations indiquées, ce qui veut dire :

- faire en sorte que "cube 2 ∈ sur-table" soit vrai
- faire en sorte que "p ∈ lieu [cube 1] soit vrai

Il faut remarquer que cela ne veut pas dire :

- faire en sorte que "cube 1 ∈ sur-table" et "p ∈ lieu [cube 1]" soient les deux vraies.

La raison est la suivante : les transformations indiquées dans un bloc doivent être résolues dans l'ordre indiqué ———————— et puisque une résolution change les données, rien n'empêche que pour résoudre la ième transformation, les résultats des transformations précédentes n'aient été partiellement ou totalement détruits.

D'ailleurs, dans l'exemple, la deuxième transformation ne peut se faire que si le résultat de la première est détruit : le cube 1 ne pourra être placé en p que si le cube 2 est enlevé de p, place où il a été mis par la première transformation.

Evidemment, il y a la possibilité de faire en sorte que les résultats des transformations partielles ne soient pas détruits. Il suffit de définir des méthodes adéquates.

Dans l'exemple, il suffirait de définir une méthode (:∍ , sur s/occuper-place-indiquée) et écrire :

```
faire (toto)
|    :∃ sur-table,s/occu-pl-indiquée (cube 1)
|    :∍ lieu ([cube 1] p)
fin
```

3.2.2.2. Preuves composées : opérateurs ∧, ∨, ¬, et quantificateurs existe(∃) et pourtout (∀)

On veut, souvent, énoncer des preuves de propositions plus compliquées que celles de la forme : " prédicat (objet)" que j'appellerai preuves "simples". Pour cela, on se sert des opérateurs logiques (∧,∨,¬) et des quantificateurs (existe et pourtout). Les quantificateurs peuvent porter sur des prédicats, prédicats de prédicats etc..., ce qui rend les analyses logiques (pouvoir de sélection) équivalentes au Calcul de Prédicats, sans restrictions au 1er ordre

Les quantificateurs s'emploient de façon analogue à la Logique, avec cependant une différence fondamentale : la quantification est toujours bornée à l'extension d'un prédicat.

Aussi, la quantification est accompagnée de l'affectation des valeurs à une variable.

Je vais donner un exemple d'énoncé de preuve composée :

Supposons qu'on a déclaré dans la base de données d'un hôpital :

prp (section 1 (0,-),malade)
prp (section 2 (0,-),malade)
prp (section 3 (0,-),malade)
prp (section(0,-), propriété)
ccatg (médecin)
rel (médecin-traitant(1,1),médecin,malade-traité(0,-),malade)

alors on peut énoncer la preuve :

existe x1 :∋ section { pourtout x2 :∋ ele x1 {existe x3 :∋ medecin
{∋médecin-traitant([x2],x3)}}}

qui veut dire : "prouver qu'il existe une section x1 telle que pas pourtout élément de la section x1, x2, il soit vrai qu'il existe un médecin x3 tel que x3 est médecin-traitant de x2".

L'affectation à x1 de la section où les malades n'ont pas tous un médecin traitant, peut servir, par la suite, dans le même programme (par exemple pour chercher des médecins pour la section

En Annexe I , je traduirai cette expression de façon primitive, en termes de blocs de preuves simples.

Je montrerai alors que, d'une manière générale, toute preuve composée peut se traduire en blocs de preuves simples ce qui justifie pleinement la désignation de preuve composée.

3.2.2.3. Enumérations composées : opérations ensemblistes ∩, ∪ et ∁

Si on veut énumérer des ensembles qui sont obtenus à partir d'extensions de prédicats par les opérations ensemblistes classiques d'union, d'intersection et complément, on se sert des opérateurs ∩ ∪ et ∁

Si, par exemple, on a fait les déclarations :

```
prp (enseignant (0,-),personne)
prp (élève-3ème cycle (0,-),personne)
```

l'énumération composée :

```
pour x1 :∋ (élève 3ème cycle ∪ enseignant)
|    print (x1)
fin
```

donne, comme résultat, la réunion, sans répétition, des élèves de 3ème cycle et des enseignants, et l'énumération :

```
pour x1 :∋ ∁enseignant
|    print (x1)
fin
```

donne comme résultat toutes les 'personne's (et non pas tous les objets ...) qui ne sont pas 'enseignant's.

Le lecteur trouvera l'application d'énumérations composées dans l'exemple de Droit exécuté sur SOCRATE-P

De façon analogue aux preuves, les énumérations composées se traduisent de façon primitive dans des blocs d'énumérations simples ce qui justifie la désignation d'énumérations composées.

La mise des preuves et des énumérations composées sous la forme de blocs de preuves et d'énumérations simples permet de rendre uniforme la notion d'énoncé composé : tout énoncé de problème composé se traduit en blocs d'énoncés de problèmes simples.

3.3. Résolution d'un problème

Je vais exposer, ensuite, la notion de résolution d'un problème.

La résolution d'un problème d'énoncé ayant la forme générale A,B,C (a,b,c) s'accomplit en deux étapes :

1 - sélection d'une méthode de résolution. S'il n'y a pas une méthode (A,B,C) et C = ∅ alors ce sera la méthode (A)

2 - après l'affectation de a,b,... à x1,x2,..., résolution du bloc (suite) de problèmes que j'appellerai "sous-problèmes" énoncés dans la méthode.

Quand le problème est élémentaire ——— sa résolution est faite au niveau mémoire et non plus au niveau sémantique. Elle consiste dans l'éxécution d'opérations de lecture, écriture et calcul qui ne nous intéressent pas dans ce travail [I].

3.3.1. Etat de la résolution

L'état de la résolution d'un problème est indiqué par le contenu d'une variable globale état, qui fait partie du système ("built-in") et est mise à-jour par celui-ci de la façon suivante :

1 - sa valeur est ECHEC tant que tous les sous-problèmes n'ont pas été résolus.
2 - sa valeur à la fin de la résolution - que j'appellerai "état final de la résolution" - est SUCCES si et seulement si l'état final de la résolution de chacun des sous-rpbolèmes a été SUCCES.
3 - si l'état final de la résolution d'un sous-problème est ECHEC cela provoque l'interruption de la résolution du problème, l'état final de celle-ci étant donc ECHEC.

[I] Par exemple la résolution de :∋(vente-m[b],a) consistera à trouver l'adresse (si on adopte l'espace virtuel SOCRATE, ce sera l'adresse virtuelle) de la portion mémoire réservée à l'extension de 'vente-m[b]', et y écrire le code interne de a.

Quand il s'agit d'un problème élémentaire si l'une des conditions d'ECHEC indiqués en 2.3. est vérifiée, sa résolution se termine avec l'état final ECHEC.

Il est équivalent de dire qu'un "problème est (n'est pas) résolu" ou dire que "l'état final de la résolution du problème est SUCCES (ECHEC)".

Je vais ensuite illustrer ces notions en m'appuyant sur l'exemple de Gestion

La méthode (g,commande-client) était :

```
mth (g,commande-client),(produit,naturel)
    faire
    |   stock-actuel[x1] > x2
    |   x3 :∋ g(commande-client)
    |   :∋ stock-actuel([x1], stock-actuel[x1] -x2)
    fin)
```

La résolution du problème :

g commande-client (moteur,20)

consistera à :

* <u>prouver</u> stock-actuel[moteur] >20

ce qui reviendra, en général, à appliquer une méthode (pour, stock-actuel) et faire la comparaison (>) si la preuve se termine en SUCCES.

* faire la <u>transformation</u>

x3 :∋ g(commande-client)

qui peut se terminer en ECHEC dans les conditions indiquées en page 0.16
Si SUCCES,

* faire la <u>transformation</u> :

:∋ stock-actuel ([moteur], (stock-actuel[moteur]- 20))

qui correspondra, en général, à appliquer une méthode (:∋ , stock-actuel) Si SUCCES, alors l'état final de la résolution est SUCCES.

3.3.1.1. Récupération d'un ECHEC

Le programmeur a le droit d'introduire des exceptions aux règles 2 et 3 : il peut ajouter, derrière un énoncé de sous-problème, une séquence qui servira à récupérer un éventuel ECHEC de la résolution de celui-ci.

Il emploiera, pour cela, la construction :

si echec alors ... fin

et insèrera la séquence entre alors et fin.

Par exemple, la méthode (g,commande-client) pourrait être :

```
mth ((g,commande-client),,(produit,naturel),
    faire
        stock-actuel[x1] > x2
        si échec
        alors g  montage-en-attente (x1,x2 - stock-actuel[x1])

        fin
        x3 :  g(commande-client)
        :  stock-actuel ([x1],stock-actuel[x1]-x2)
    fin)
```

forme qui correspond à une satisfaction de commande plus "définie" (I) que celle traduite par la méthode précédente.

Le lecteur trouvera des exemples d'emploi de cette construction dans l'exemple de Gestion   et dans l'exemple des cubes exécutés en SOCRATE-P.

si échec alors ... sinon ... fin

La construction de programmes peut être simplifiée par l'emploi

---

[I] plus "définie" au sens de la Gestion : capable d'être appliquée avec SUCCES à plus de situations.

de si échec alors ... sinon ... fin

Entre sinon et fin on insère la séquence à exécuter en cas de SUCCES.

Il faut, cependant, remarquer que ceci peut obscurcir la sémantique du programme.

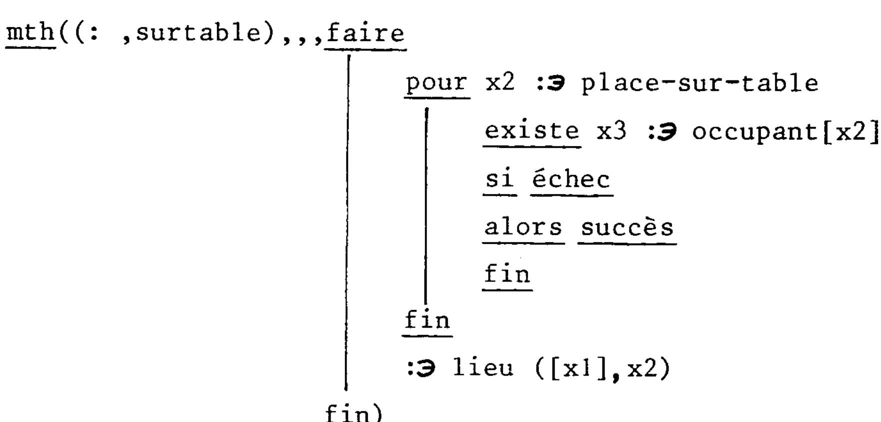
3.3.2. Interruptions de la résolution

Le programmeur a le droit de mettre-à-jour la variable état

S'il lui donne la valeur SUCCES, il provoque, indirectement, l'arrêt de la résolution, et l'état final est SUCCES.

Reprenons l'exemple des cubes

On peut définir une méthode (:∋ , surtable), pour "mettre des cubes sur la table" :



```
mth((: ,surtable),,,faire
                  |  pour x2 :∋ place-sur-table
                  |  |  existe x3 :∋ occupant[x2]
                  |  |  si échec
                  |  |  alors succès
                  |  |  fin
                  |  fin
                  |  :∋ lieu ([x1],x2)
                  fin)
```

Cette méthode consiste à chercher une place sur table et dès qu'on en trouve une libre, la résolution de la boucle se termine en SUCCES. Cette sortie de la boucle est une interruption parce que sa sortie normale correspond comme on verra en page III.23 à l'épuisement de l'ensemble.

3.3.2.1. Raison d'une interruption

Dans le cas où la résolution d'un problème peut être interrompue (en ECHEC ou SUCCES) en différents points de la méthode, il est important de pou- voir connaître le point d'interruption.

Pour cela, le programmeur a le droit de mettre-à-jour une deuxième variable globale appelée raison en même temps que état. Par exemple, il écrira 'échec,1' 'échec,2', etc... (la raison est un nombre naturel).

Un peu plus loin, je vais en donner un exemple.

3.3.2.2. "Down" et "up"

La mise-à-jour d'état avec la valeur SUCCES provoque indirectement la fin de la résolution. Cependant, l'état final est SUCCES, alors qu'il y a des cas où l'on veut que l'état final reste ECHEC.

Le programmeur peut le faire en se servant de 'down' : à l'intérieur d'un bloc, down veut dire "abandonner la résolution du bloc". L'état n'est pas changé.

Je vais montrer comment l'on utilise la raison et down.

Reprenons l'exemple des cubes.

On peut définir une nouvelle méthode (:∃ , sur) comme suit :

```
mth((:∋,sur),,,faire
          ¬ existe x3 :∋ sur[x2]
          si échec
          alors échec,1
          |     down
          fin
          ¬ existe x3 :∋ sur [x1]
          si échec
          alors échec,2
          |     down
          fin
          :∋ (sur[x1] x2)
      fin)
```

Elle a 3 sorties possibles :

* échec,1 , au cas où il y a un cube sur le cube x2
* échec, 2, au cas où il y a un cube sur le cube x1

qui sont des interruptions, et

* succès qui est la sortie normale, après que le cube x2 soit mis effectivement sur le cube x1.

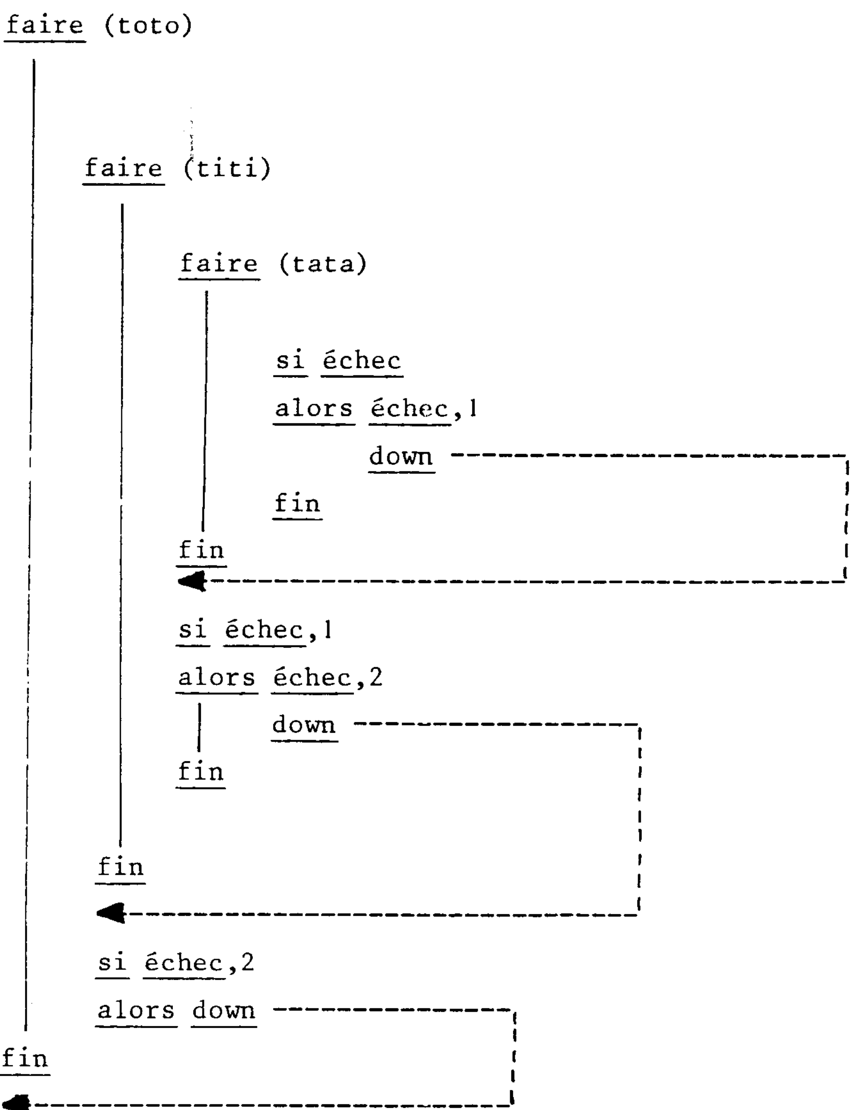
Les deux échecs peuvent être récupérés, après la résolution. Par exemple, comme suit :

```
:∋ sur ([x1], x2)
si échec, 1
alors :∌ sur ([cube 2], sur [cube 2])
sinon si échec,2
|     alors :∌ sur ([cube 1], sur [cube 1])
|     fin
fin
:∋ sur ([cube 1], cube 2)
```

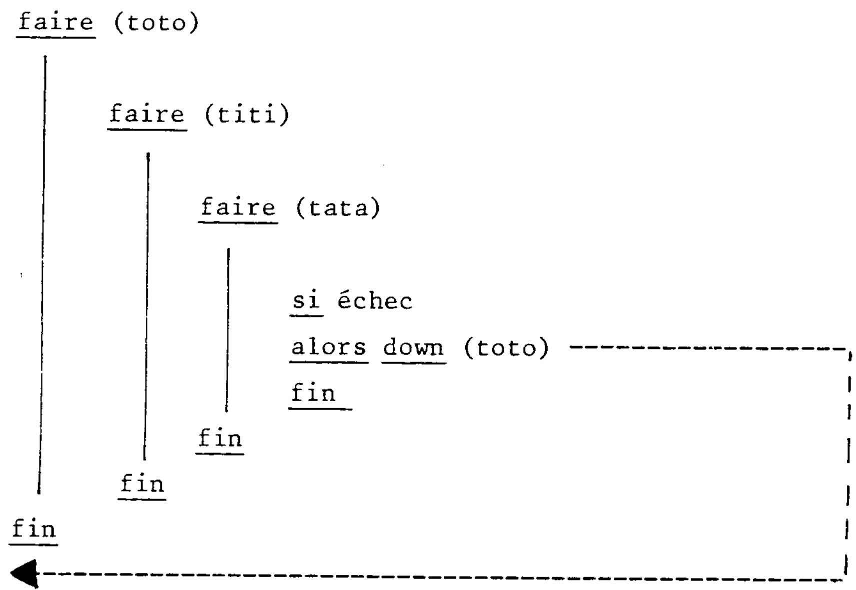
Down multiple

Quand on a des blocs imbriqués, si l'on veut abandonner plusieurs blocs "d'un coup" on peut se servir d'un down "multiple" : down auquel on ajoute le nom ———— du dernier bloc que l'on veut abandonner.

Par exemple, la structure de contrôle suivante :



s'exprime plus simplement à l'aide d'un "down" multiple :



L'emploi de down, multiple ou non, conjugué avec la manipulation d'état et raison sert à définir des blocs et boucles à sorties multiples avec des structures très générales. Cependant, l'utilisation du down multiple est à éviter parce qu'elle rend les contrôles de correction plus difficiles à faire.

up

up sert à interrompre la résolution d'un bloc pour la relancer à partir du début.

Il n'y a pas de up multiple parce que le down multiple et le up suffisent à le simuler.

Le up est important entre autre, pour définir des algorithmes de backtrack

Dans les exemples de et dans ceux exécutés en SOCRATE-P le lecteur peut trouver beaucoup de cas où l'on emploie le down et le up.

3.3.3. Processus de résolution d'un problème

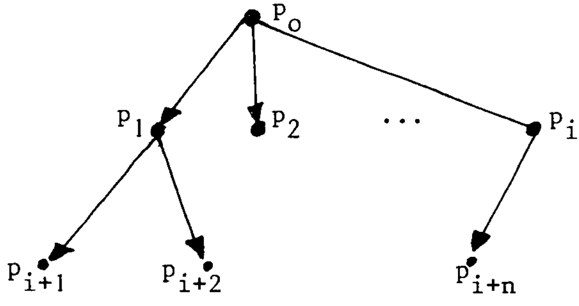
J'appellerai processus de résolution d'un problème (ou, simplement, "processus") l'application d'une méthode afin de résoudre celui-ci.

J'emploie le terme "processus" par analogie avec les processus d' exécution de procédures (appelés dans la littérature anglaise "instances of execution" ou "copies")

Il y a deux sortes de processus : ceux qui sont engendrés et gérés de façon interne (par le système) et ceux qui sont engendrés et gérés par l'utilisateur.

L'utilisateur peut définir une méthode de résolution qui consiste à gérer plusieurs processus.

Donc, a un instant donné, la résolution sera représentée par un arbre de processus vivants



Les processus sont tous en état d' "attente" - je dirai endormis - sauf un, qui est en train de s' "exécuter" - je dirai qu'il est réveillé -

La gestion des processus se programme à l'aide des opérateurs suivants :

x1 :∋ open (<méthode>)

sert à engendre un processus appliquant la méthode dont le nom historique [I]

[I] 'open' inclut l'opération : g(processus), 'processus' étant une c-catégorie implicite. En page II.13., j'ai défini "nom historique".

sera affecté, par effet de bord, à x1.

Je dirai que le processus x1 est alors vivant.
On peut ajouter une liste de paramètres à droite de l'indication de la méthode .

* get (<processus>)

réveille le <processus>.
Je dirai que le processus est alors en état de réveillé (analogue du terme "actif" employé dans la littérature des systèmes d'exploitation

* resume

indique un point où un processus en état de réveillé, s'endort, le contrôle étant transféré vers celui qui l'avait réveillé par get.

Le processus "se supprime" (ce qui veut dire "cesse d'être vivant"), au lieu de s'endormir, si résume est remplacé par return.

Tant le resume comme le return peuvent être accompagnés d'un passage de paramètres. A ce moment-là, le get correspondant possède, à sa gauche, une liste de paramètres de retour

* close (<processus>)

C'est l'opération duale de open, c'est-à-dire qu'elle sert à supprimer le processus s'il ne s'est pas déjà supprimé lui-même, par return.

Je vais illustrer ceci en présentant rapidement un exemple pris en Droit, que j'ai programmé en SOCRATE-P et je discuterai, aussi, en détail, un peu plus tard IV- 3

Il s'agit de résoudre une preuve (prouver qu'une personne x a violé un domicile y) par la gestion en pseudo-parallèle de deux processus qui appliquent deux méthodes différentes de résolution de la preuve.

Les deux méthodes sont appelées "violence-concrète" qui cherche à prouver que x a brisé quelque chose en y et "violence-abstraite" qui essaie de prouver que x est entré, en faute, dans y.

J'emploierai le terme "violence" pour me rapprocher du langage utilisé en Droit.

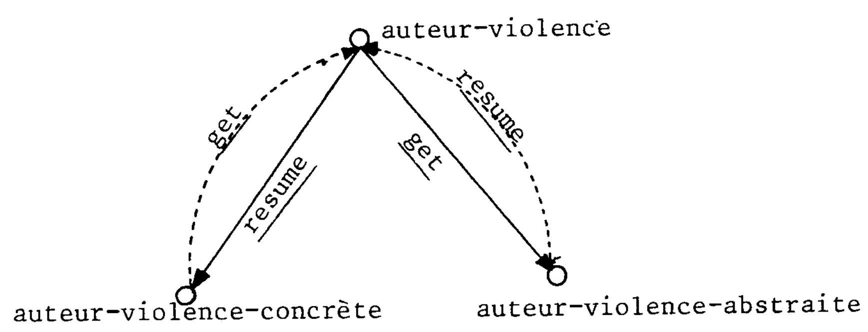
La preuve s'énonce :

auteur-violence ([maison 1] jacques)

et la méthode (∋ ,auteur-violence) est la suivante :

```
mth((∋,,auteur-violence)
   faire
   |    x1 :∋ open(méthode( ∋,auteur-violence-cte))
   |    x2 :∋ open(méthode( ∋,auteur-violence-abste))
   |    faire
   |    |    get (x1)
   |    |    si échec
   |    |    alors get (x2)
   |    |    |     si échec
   |    |    |     alors up
   |    |    |     up
   |    |    fin
   |    fin
   |    close (x1)
        close (x2)
   fin)
```

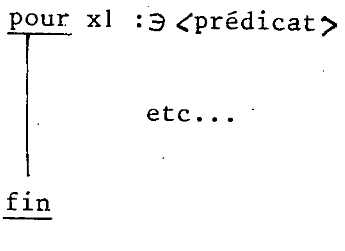
ce qui donne l'arbre :



3.3.3.2. Traduction des boucles 'pour' dans des constructions primitives

Comme je l'ai déjà dit, les boucles 'pour' sont des facilités d'écriture.

Une boucle avec la forme générale :



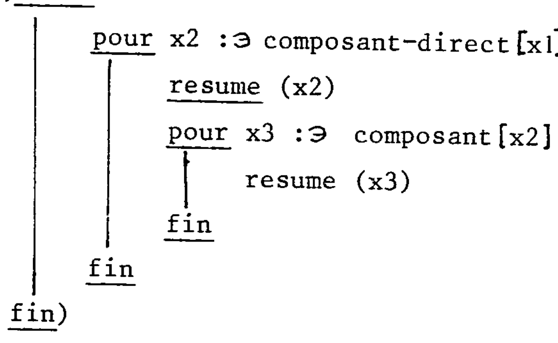
```
pour x1 :∋ <prédicat>
|
|       etc...
|
fin
```

se traduit, de façon primitive, dans :

```
x2 :∋ méthode (pour,<prédicat>)
x3 :∋ open (x2)
faire
|    x1 :  get(x3)
|    si x1 = ∅
|    alors succès
|    fin
|
|    etc...
fin
```

Dans une méthode (pour, prédicat), le passage de paramètres se fait par resume, sauf si le cardinal maximum du prédicat est 1 auquel cas, il se fait par return.

Par exemple, dans l'exemple de Gestion la méthode (pour,composant) est :



```
mth((pour,composant),,,faire
                        |  pour x2 :∋ composant-direct[x1]
                        |  |  resume (x2)
                        |  |  pour x3 :∋ composant[x2]
                        |  |  |  resume (x3)
                        |  |  fin
                        |  fin
                        fin)
```

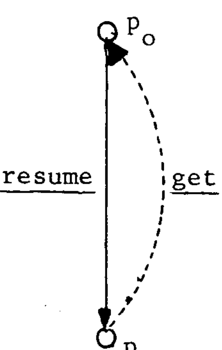
Si l'on écrit dans un programme :

```
pour x4 :∋ composant [moteur]
|    print(x4)
|
fin
```

cela va se traduire, de façon interne, comme suit :

```
p :∋ open (méthode(pour,composant)
faire
|    x4 :∋ get(p)
|    si x4 = ∅
|    alors succès
|    fin
|    print(x4)
|    up
fin
```

ce qui correspond aux processus suivants :

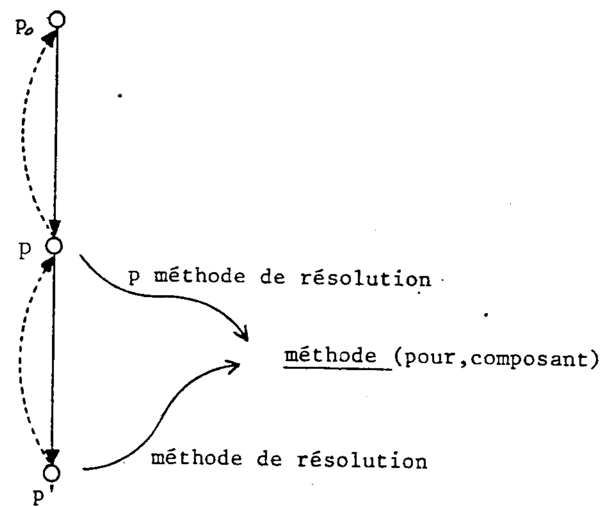


Le processus p_o passe le contrôle à p (get(p))qui s'"exécute" sur la méthode (pour,composant) jusqu'au premier resume et retourne alors le contrôle à p_o en lui transmettant simultanément le paramètre (qui est le premier composant-direct). Celui-ci est affecté à x4.

Le processus p continue à s'exécuter en faisant print(x4), et up qui relance les opérations.

Au deuxième réveil dû au processus p_o, le processus p engendre lui-même un autre processus p', sur la méthode (pour,composant) qui s'exécute jusqu'au premier resume et retourne alors le contrôle à p en lui passant le paramètre (qui est le premier composant-direct du premier composant-direct du moteur) et ainsi de suite.

L'arbre des processus est le suivant :



On pourrait appeler ce type de traitements des résolutions "par application récursive d'une méthode" ceci par analogie aux appels récursifs de procédures en ALGOL.

3.3.4. Espaces

Un problème se résoud dans un espace que j'appellerai sous espace de résolution [I].

Quand l'espace de résolution n'est pas la base de données, il doit être indiqué avec l'énoncé derrière dans. Par exemple :

:∋ sur ([cube 1] cube 2) dans toto

veut dire que toutes les opérations élémentaires (d'accès ou mise-à-jour) qu'en-

---

[I] en fait, la résolution d'un problème peut exiger des espaces auxiliaires

traîne la transofrmation, prendront effet localement dans l'espace appelé 'toto'.

Un espace ne peut être utilisé sans avoir été engendré auparavant : [I]

espace (toto)

Ceci n'est pas vrai seulement pour deux espaces qui existent a priori : celui qui contient toutes les définitions implicites (built-in) II et qui est indiqué par la variable système, et la base de données qui est indiquée par la variable base.

3.3.4.1. Initialisation

Une fois qu'un espace a été engendré, on peut l'initialiser. Par exemple, après :

espace(toto)

on peut faire :

ccatg (voiture) dans toto
ccatg (personne)
rel (propriétaire (1,1),personne,propriété(0,-),voiture)
dans toto

après ces opérations, les états de la base et de toto sont ceux représentés symboliquement par ce que le lecteur voit sur la page III.28 et sur la page III.27 en tenant compte, pour celle-ci, de ce qu'il voit à travers.

Si ensuite, on fait :

x1 :∍ g(personne)dans toto
x2 :∍ g(voiture) dans toto
:∍ (propriété [x1] ,x2) dans toto

les nouveaux états sont ceux qu'il voit représentés de façon anlogue, sur les pages III.35 ,III.36 et III.37

[I] 'espace' est une reln définie dans le système (bint-in).

[II] c'est-à-dire : (a) - catégories implicites (par exemple page II.3)
actg(entier)
(b) - fonctions d'accès implicites (par exemple dom, co-dom, inv (page II.7)
(c) - méthodes implicites (par exemple les opérateurs élémentaires et les opérateurs implicites n̄ élémentaires comme cctg, acatg, etc)

SYSTEME : BASE DE DONNEES: TOTO :

catégorie

• personne

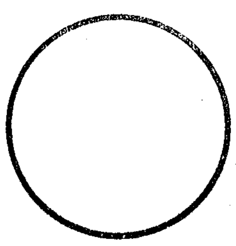
• voiture

fonction d'accès

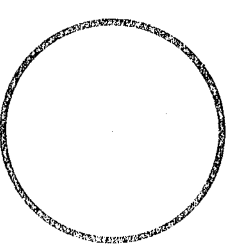
propriétaire

propriété

personne

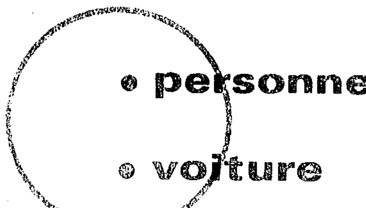


voiture



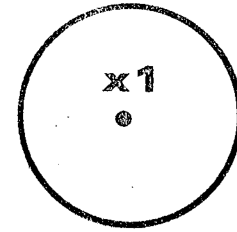
SYSTEME : BASE DE DONNEES: TOTO :

catégorie

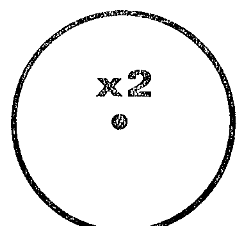


fonction d'accès

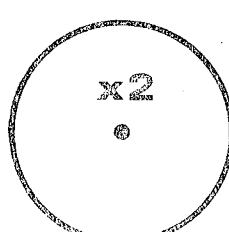
personne



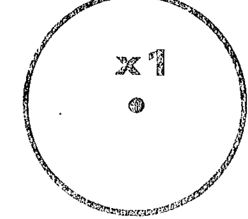
propriété [x1]



voiture



propriétaire [x2]



SYSTEME : BASE DE DONNEES: TOTO :

catégorie

• personne

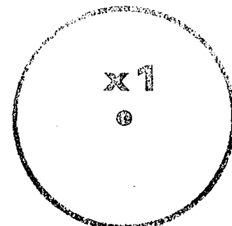
• voiture

fonction d'accès

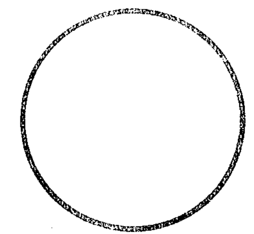
propriétaire

propriété

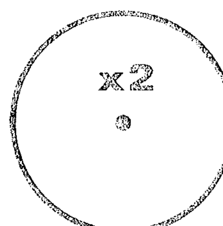
personne



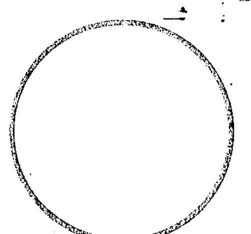
propriété [x1]



voiture



propriétaire [x2]



Voici pourquoi ces opérations produisent ces états.

Deux des catégories définies implicitement dans système sont 'c-catégorie' ________ et 'fonction-d-accès'.

Les expressions :

ccatg personne
(ou, ce qui est équivalent, "ccatg personne dans base")
ccatg voiture dans toto

seront interprétées en engendrant une c-catégorie et en lui donnant le nom 'personne' ce qui est possible parce que base ' ' tout ce que "voit" système donc la catégorie 'catégorie' entre autres, et, ensuite, en générant une c-catégorie 'voiture' dans toto, ce qui est possible pour la même raison puisque 'toto' "voit" tout ce que "voit" base qui elle-même "voit", comme on dit, ce que "voit" système, donc 'catégorie', entre autres.

Par un raisonnement identique le lecteur peut essayer de comprendre le sens des autres expressions.

Si j'emploie encore les termes intuitifs dont je viens de me servir pour décrire les règles de "partage" des données entre espaces locaux, base et système, je dirais que le système ne "voit" pas ce que "voit" la base et celle-ci ne "voit" pas ce que voient les espaces locaux.

Par exemple, si l'on faisait :
ccatg (voiture) dans toto
g voiture

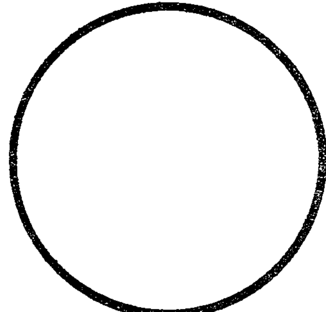
cela provoquerait ECHEC parce que base ne "voit" pas voiture.

3.3.4.2. Changement du contenu

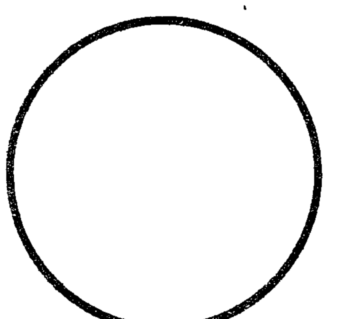
Changer localement le contenu d'un espace (E) c'est changer dans cet espace (E) l'extension (ou extensions) d'un (ou plusieurs) prédicat(s) que E "voit".

SYSTEME :

catégorie



fonction d'accès



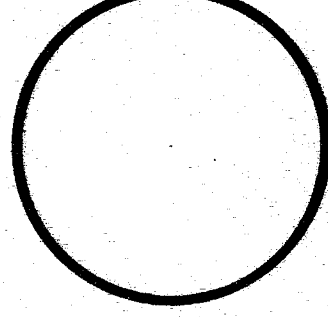
TOTO :

propriétaire

• voiture

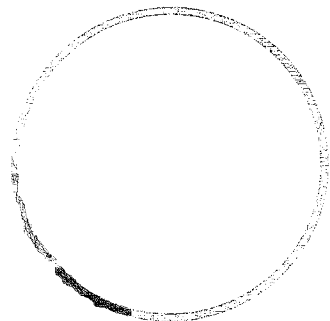
propriété

voiture



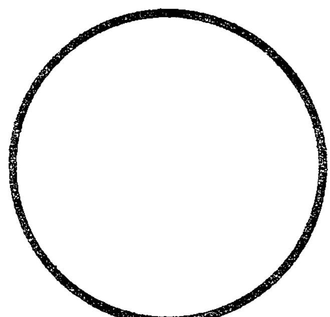
BASE DE DONNEES:

- **personne**

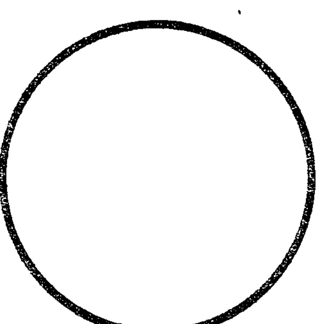


SYSTEME :

catégorie



fonction d'accès



TOTO :

propriétaire

• voiture

propriété

propriété [x1]

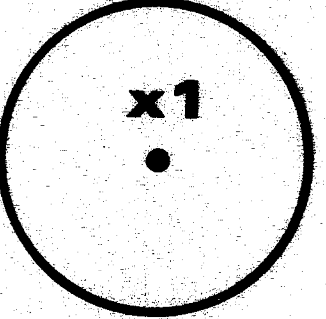
x1

x2

voiture

propriétaire [x2]

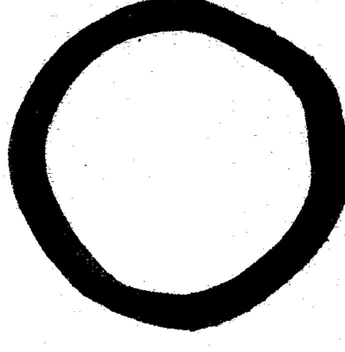
x2



BASE DE DONNEES:

● personne

personne



Comme on l'a déjà vu, il y a 4 façons de changer l'extension d'un prédicat :

- si le prédicat est une catégorie, on peut engendrer (g) ou supprimer (s) des éléments
- si le prédicat est de type FX, on peut lui affecter (:∍) ou désaffecter (:∌) des éléments.

Par exemple, si dans l'état de 'toto' représenté en page III.35 on fait :

:∌ (propriété [x1] x2) dans toto

ceci amènera 'toto' à l'état représenté en page III.30 où les extensions des prédicats 'propriété [x1] ' et 'propriétaire [x2] ' ont été changées

Je souligne extensions pour bien distinguer le fait que ce ne son pas les prédicats qui changent mais leurs extensions.

TOTO :

propriétaire

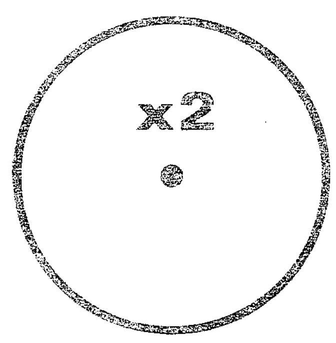
voiture

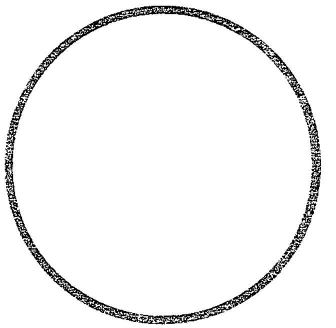
propriété

x1

voiture

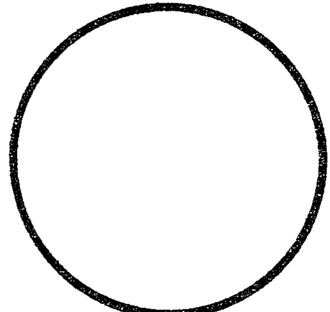
propriétaire [x2]



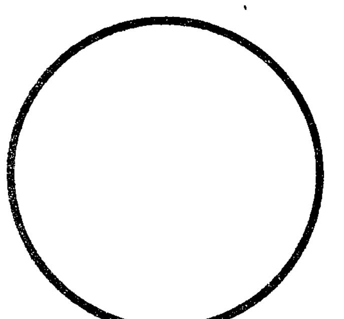


SYSTEME :

catégorie



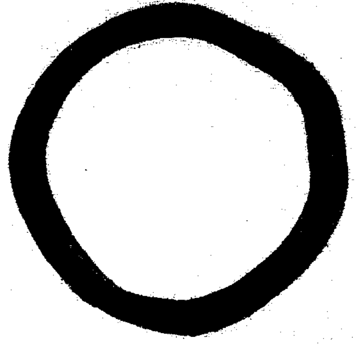
fonction d'accès



BASE DE DONNEES:

- personne

personne



3.3.4.3. Accès au contenu

Si, l'on veut connaître le contenu d'un espace, on énonce des preuves ——— ou énumérations ——— dans cet espace. Le système répondra en fonction du contenu de la base (en prédicats et leurs extensions) et en tenant compte des modifications locales apportées par l'espace. Intuitivement, depuis un espace, on "voit" la base de données modifiée par les données locales.

Par exemple, si l'on exécute :

s catégorie(personne) dans toto

on ne "verra" plus dans 'toto' la catégorie 'personne', mais on continuera à la "voir" dans la 'base'

Si, par exemple, on énonce :

∋ propriété ([x2], x1)

la réponse du système est ECHEC et si l'on exécute :

∋ propriété ([x2], x1) dans toto

la réponse est SUCCES.

3.3.4.5. Validation d'un espace

Toutes les données locales à un espace peuvent être validées d'un seul coup par l'emploi de l'opérateur 'valider'. Par exemple :

valider (toto)

aura comme effet d'intégrer tout ce qui est dans toto à la base (symboliquement cela se représenterait par le collage des deux feuilles III. 35 et 36

Naturellement, on peut aussi valider une partie seulement des données locales. Par exemple :

```
pour x1 :∋ catégorie dans toto
|    x2 :∋ nom  x1
|    ccatg (ele x2)
fin
```

a comme effet de "valider" toutes les catégories déclarées en toto (et seulement les catégories).

Suppression d'un espace

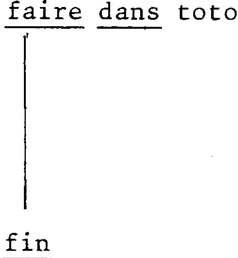
Si l'on veut supprimer un espace, on utilise s. Par exemple :

s toto

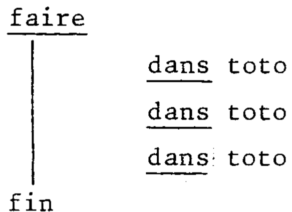
veut dire que 'toto' a disparu sans laisser de trace (symboliquement cela s'exprimerait en déchirant la feuille III.35

3.3.4.6. Résolution d'une suite de problèmes dans un espace

Si on veut exprimer qu'une suite de problèmes doit être résolu dans un espace, par exemple 'toto', on écrit simplement :



ce qui est équivalent à :



c'est pourquoi, j'ai le droit d'écrire :

:∋ sur ([cube 1] cube 2) dans toto

La méthode (:∋,sur) sera complètement appliquée dans toto, c'est-à-dire que chacun des sous-problèmes énoncés dans le bloc seront résolus dans 'toto' et, récursivement, les sous-problèmes des sous-problèmes, etc... Tout l'arbre des sous-problèmes sera résolu dans toto.

3.3.4.7. Contextes

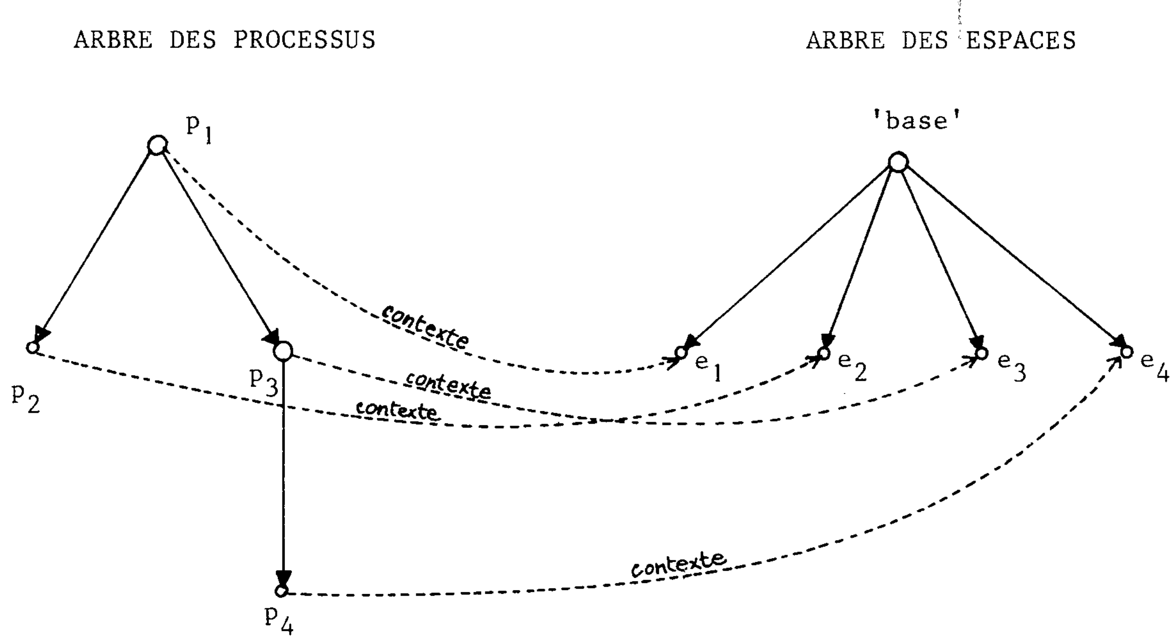
Tout processus de résolution a en plus d'un espace de résolution un contexte qui est un autre espace.

Les 'contextes' sont des espaces avec une sémantique assez particulière : toute affectation a des variables (var) prend systématiquement effet dans le contexte du processus et non pas dans son espace de résolution. Par ailleurs, le contexte d'un processus de résolution est automatiquement supprimé avec le processus et ceci n'est pas vrai pour l'espace de résolution qui peut être conservé.

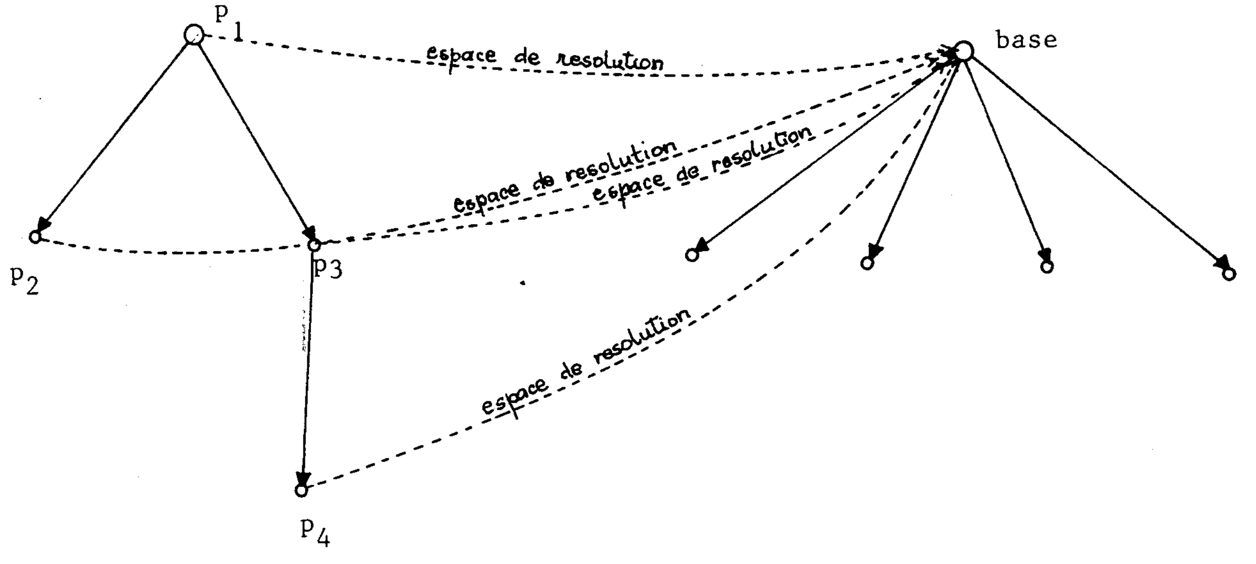
Le contexte d'un processus est engendré de façon interne, au contraire de son espace de résolution qui doit être engendré explicitement, s'il n'est pas la base de données

Arbre des processus et les contextes

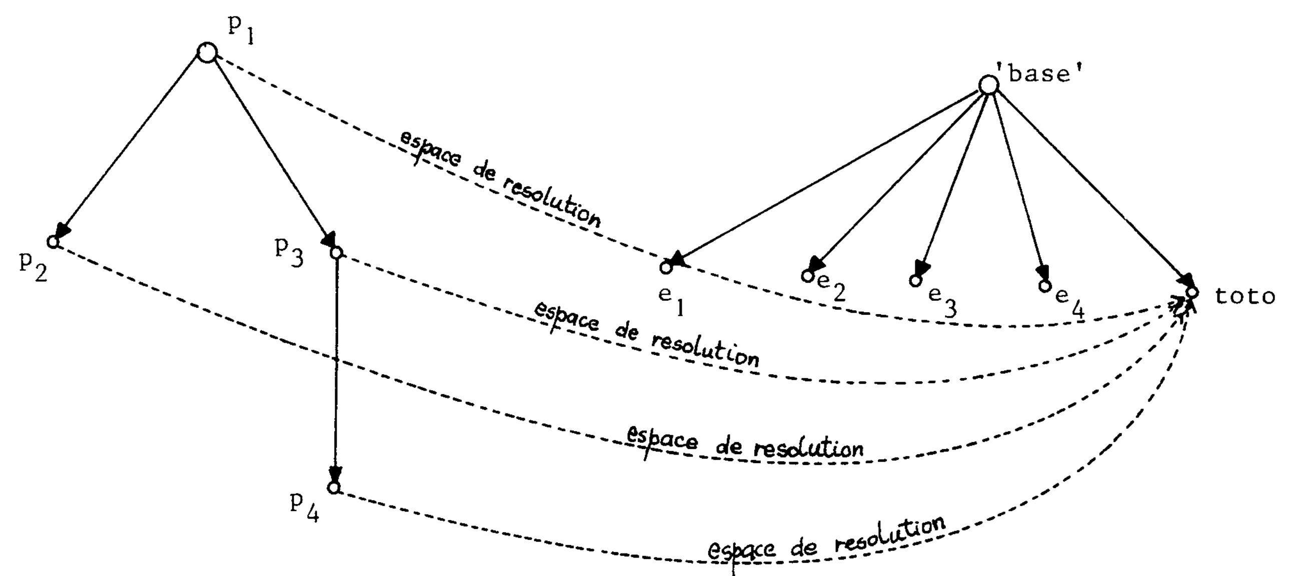
A un instant donné, si l'utilisateur a des processus vivants p_1 p_2 p_3 et p_4, ils peuvent être reliés comme suit à leurs contextes :



Si tous les processus se résolvent <u>dans</u> la 'base' on a, en plus :



ou, s'ils se résolvent localement en 'toto' :



33.4.8. Sémantique de l'utilisation des espaces

Les contraintes d'utilisation des espaces sont rigoureusement définies dans système.

Tout processus de résolution est demandé par un utilisateur (élément de la c-catégorie implicite 'utilisateur') l'initialisation exige :

1 - son identification
2 - l'indication de l'espace de résolution (par dans)

L'initialisation sera refusée (ECHEC) si :

$$\neg \exists x1 :\ni \text{utilisateur} \{\exists \text{ ident } ([x1], \text{input})\}$$

ou si le sujet n'est pas suffisamment qualifié pour résoudre le problème dans l'espace.

Qualifications

Tous les utilisateurs sont caractérisés, à l'intérieur du système, par une a-qualification minimum et une t-qualification qui sont des nombres naturels.

Un espace a toujours une a-qualification minimum et une t-qualification minimum qui sont aussi des nombres naturels.

La a-qualification minimum d'un espace X (a-qual-min[X]) est la qualification minimum que doit avoir un utilisateur pour énoncer des preuves ou énumérations élémentaires dans X.

La t-qualification minimum d'un espace X (t-qual-min[X]) est la qualification minimum nécessaire à un utilisateur pour pouvoir énoncer des transformations élémentaires dans X.

Les a et t-qualifications des espaces observent certaines règles précises :

Par exemple :

pourtout x :∋ espace {a-qual-min[x] ⩾ a-qual-min[base]}

pourtout x :∋ espace {t-qual-min[x] ⩽ t-qual-min[base]}

pourtout x :∋ espace {t-qual-min[x] ⩾ a-qual[x]}

pourtout x :∋ espace {a-qual-min[x] ⩾ a-qual-min[système]}

pourtout x :∋ espace {t-qual-min[x] ⩽ t-qual-min[système]}

⋮

Limitations ad-hoc

Ce système de qualifications est un moyen général de règler l'utilisation des espaces, cependant, ce mécanisme étant trop général, on doit pouvoir apporter des exceptions ad-hoc.

Les limitations ad-hoc s'expriment au niveau des méthodes.

3.4. Utilisation du système

Les actions internes qui correspondent à une utilisation du système, sont les suivantes :

1 - l'utilisateur fait 'l'initialisation et indique l'espace sur lequel il veut résoudre les problèmes.

2 - si l'initinialisation est acceptée , il énonce un **problème**.

3 - le système sélectionne une méthode de résolution ______________ et engendre un processus de résolution qui l'applique. L'espace de résolution est celui qui a été indiqué en (1).

4 - pour chaque problème élémentaire, le système vérifie si la qualification de l'utilisateur est supérieure à la qualification minimum de l'espace.

Si à un certain instant, il y a plusieurs utilisateurs qui se partagent un même espace, cela veut dire qu'il y a plusieurs arbres de processus, (engendrés par le système et engendrés par l'utilisateur) dont l'espace de résolution est le même.

Evidemment, les transferts de contrôle entre processus d'utilisateurs différents se feront selon d'autres critères que ceux qui règlent les transferts de contrôle entre processus d'un même utilisateur.

Je pense qu'une voie de recherche très utile se présente dans l'étude de l'utilisation "simultanée" d'un espace par différents utilisateurs, pour résoudre des problèmes.

Conclusion

Les mécanismes qu'on trouve dans les nouveaux langages d'Intelligence Artificielle [34] [35] [39] [40] [41] [43].
pour énoncer et résoudre des problèmes, reposent sur un certain nombre de concepts de base que je vais comparer avec ceux du Langage Sémantique. Je m'appuierai principalement sur leur implémentation en PLANNER

En PLANNER, il y a trois formes générales d'énoncés de problèmes :

(goal (A_1 A_2 ... A_i)) appelés"goal statement"

(assert (A_1 A_2 ... A_i)) appelés"assert statement"

(erase (A_1 A_2 ... A_i)) appelés"erase statement"

où A_i représente un objet ou alors une variable.

Un "goal statement" énonce une preuve. Par exemple :

(goal (composant b a))

se traduirait dans :

∍ composant([b],a)

expression trouvée déjà [I]

[I] si le nombre d'arguments du "goal statement" est supérieur à 2, alors la traduction est différente
Par exemple :
(goal (achat antoine fiat grenoble 2-janvier))
se traduirait :
existe x ∍ achat (antoine,fiat,grenoble,2-janvier)

Les "assert statements" et "erase statements" énoncent des transformations : les "assert" se traduiraient dans des affectations ou générations, les "erase" **dans** des désaffectations ou suppressions.

Par exemple :

(assert (composant-direct **c** d)

se traduirait :

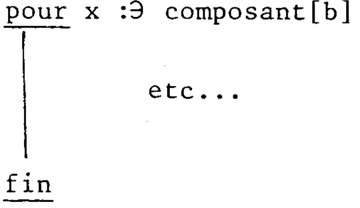
:∋ composant-direct ([c], d)

En PLANNER, il n'y a pas d'énumérations.

Cette construction est fondamentale en bases de données, parce que les programmes parcourent souvent des ensembles, afin de faire des traitements sur chacun des éléments.

Les énoncés d'énumérations sont semblables aux "foreach statements" de SAIL [39] , que celui-ci a d'ailleurs hérité de LEAP [17]

Par exemple, l'énumération :



```
pour x :∋ composant[b]
|
|      etc...
|
fin
```

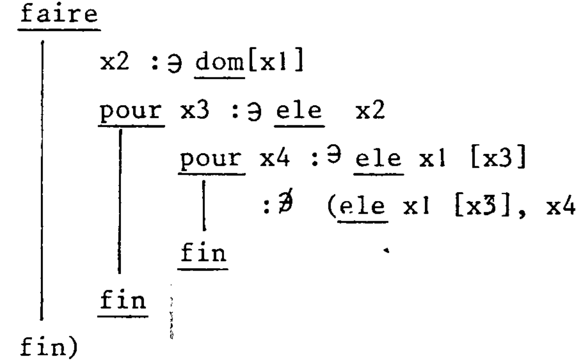
se traduirait, en SAIL :

```
foreach x suchthat composant ⊕ b = x
do
|
|    etc...
|
end
```

Les énumérations composées ——— n'ont pas d'équivalent en SAIL.

Par ailleurs, il y a une sorte d'énumérations que je voulais mettre en évidence : ce sont toutes celles qui concernent des ensembles d'objets du

2nd ordre (fonctions d'accès, prp's, etc...) et qui peuvent être très utiles, comme, par exemple, le programme suivant :



qui permet d'effacer de la base de données tous les prédicats d'une fonction d'accès x1.

En ce qui concerne, maintenant, les méthode de résolution de problèmes, j'ai expliqué en II.39 que celles-ci ont, en PLANNER, la forme de "theorems".

Les "theorems" sont de 3 types, chacun correspondant aux 3 formes d'énoncés :

"consequent theorems"

Ces "theorems" s'appliquent à la résolution de problèmes énoncés par "goal statements".

La méthode (∋, composant) se traduirait dans le "consequent theorem" suivant :

```
[consequent
        (composant ?x1 ?x2)                 } "pattern" cf p. II 39
        (goal (composant-direct ?x3 ?x2))
        (goal (composant ?x1 ?x3))]
```

et la résolution de :

```
        (goal (composant  b  a))
```

dans le cas où les données stockées sont les triplets suivants :

(composant-direct b e)   [I]
(composant-direct g a)
(composant-direct f a)
(composant-direct b f)

se passera comme suit :

- chercher le triplet indiqué dans le "goal statement"
  puisqu'il ne figure pas parmi ceux qui sont stockés,
- chercher un "theorem" dont le "pattern" soit conforme au triplet. (page II.40)
  Supposons que le "theorem" trouvé est celui que j'ai défini.
- b est affecté à x1 et a est affecté à x2
- le procédé que je viens de décrire est appliqué au 1er "goal statement" et, comme le triplet (composant-direct g a) est conforme au triplet du "goal", g est affecté à x3
- l'application du procédé du 2ème goal conduit à "failure"(correspondant d'ECHEC) et alors le contrôle revient automatiquement en arrière jusqu'au 1er "goal statement"; l'affectation de g à x3 est remise en cause, et le procédé repart avec une nouvelle valeur pour x3,qui est f.
- le triplet "(composant-direct b f)"est trouvé,et la résolution termine en SUCCES.

"antecedent theorems"

Ces "theorems" sont appliqués, de façon analogue,pour résoudre des problèmes énoncés par "assert statements". Ils correspondent donc aux méthodes de génération (g) et affectation (:∋).

[I] Ceci veut dire la même chose que :
$e \in$ composant-direct [b]

"erase theorems"

Les "erase theorems" s'appliquent à la résolution de problèmes énoncés par "erase statements".

Evidemment, un "theorem" peut contenir des "goal", "assert", ou "erase statements", quelque soit son type.

Les deux caractéristiques qui différencient ce mécanisme de résolution de problèmes de celui que j'ai proposé sont :

* le choix des "theorems" est moins déterministe que le choix des méthodes
* la structure de contrôle est fondée sur un mécanisme de retour en arrière automatique.

En Langage Sémantique il y a aussi retour en arrière automatique, mais dont l'emploi est limité aux énumérations et aux preuves avec quantificateurs. On le comprendra si on compare le "theorem" avec la méthode (∋, composant) définie en p. II 40

L'exemple que j'ai donné sur les composants n'illustre pas toute la puissance du mécanisme en PLANNER : Si la résolution d'un "statement" est en ECHEC, après que des changements des données soient intervenus, tous ces changements seront automatiquement effacés lors du retour en arrière.

En Langage Sémantique, cet effacement des données n'est pas fait automatiquement, comme en CONNIVER [40]

Je vais expliquer pourquoi.

Reprenons l'exemple des cubes

Supposons que j'avais défini la méthode (:∌, lieu) de la façon suivante :

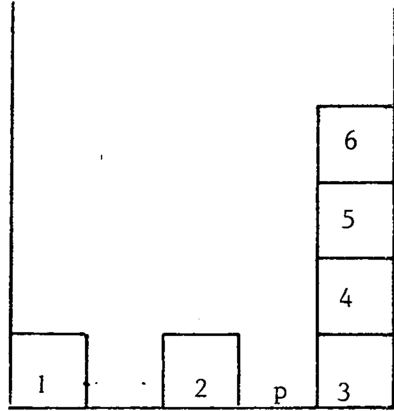
```
mth ((:∌ ,lieu),,, faire
                      | x3 :∋  lieu[x1]
                      | x4 :∋  place-sur[x3]
                      | x5 :∋  occupant[x4]
                      | :∌ lieu ([x5],x4)
                      | :∋ (lieu[x1],x2)
                   fin)
```

méthode qui est,donc,appliquée "récursivement" : pour enlever un cube de son lieu, il faut d'abord enlever de son lieu celui qui est sur lui.

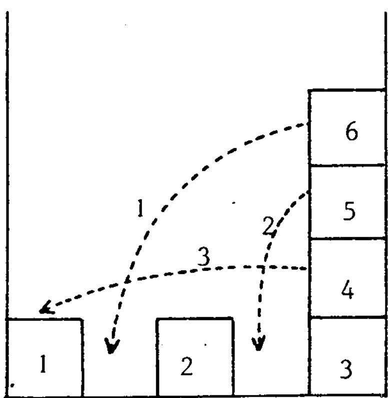
Cette méthode appliquée à la résolution de :

:∃ lieu ([cube 3],p)

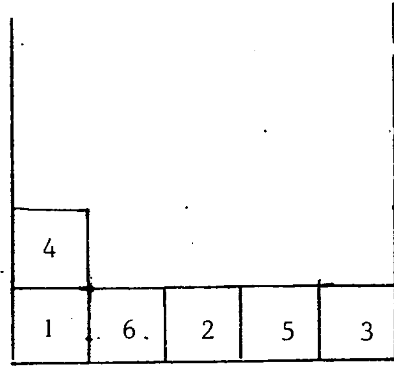
quand les données représentent l'état suivant des cubes :



va produire les "déplacements" suivants :



qui conduisent à une configuration



où l'objectif de la résolution (mettre le cube 3 en p) ne peut plus être atteint.

Si on programmait en PLANNER une résolution analogue, l'échec provoquerait automatiquement la "remise" des cubes dans leur état de départ et la recherche d'une nouvelle méthode. D'ailleurs on voit bien que l'un doit entraîner l'autre.

Mon point de vue est que le retour au point de départ n'est pas toujours nécessaire et cet exemple le montre bien : il vallait mieux enlever le cube 5 de la place p.

Ceci ne veut pas dire qu'on ne puisse pas programmer la solution PLANNER, en Langage Sémantique :

```
pour x1 :∋ méthode (:∋, lieu)
|    x3 :∋ espace
|    :∋ lieu, ele x1 ([cube 3], p) dans x3
|    si succès
|    alors valider (x3)
|    |      succès
|    fin
|    s (x3)
fin
```

Espaces

Dans la méthode que je viens d'écrire, j'emploie un espace afin de résoudre provisoirement le problème de mettre un cube dans une place.

Les espaces peuvent être employés d'autres manières, et on peut trouver en [37] une bonne étude.

D'ailleurs, certains de ces emplois correspondent à ce qui, dans le domaine des bases de données, est appelé espaces provisoires ou espaces de travail

En CONNIVER et QA4 les espaces ("data base contexts") ont une structure d'arbre à profondeur variable qui est donc plus générale que celle que j'ai proposée. J.R. ABRIAL propose une structure semblable dans [2]

Pseudo-parallèlisme

La structure de contrôle qu'on trouve en CONNIVER [40] INTERLISP [39] et POPLER [39] a les caractéristiques suivantes :

- un processus peut réveiller un autre processus quelconque, qui n'est pas forcément son père ni son fils.
- le point de reprise d'un processus n'est pas forcément celui où il s'était endormi,
- il peut y avoir plus d'un processus réveillé à chaque fois.

J'ai préféré une structure de contrôle moins flexible parce que je n'ai pas trouvé de contre-exemples et elle me paraît favoriser la correction des programmes.

Cependant, l'introduction du multitraitement me paraît une voie à étudier.

ANNEXE 1

TRADUCTION DE PREUVES

ET D'ENUMERATIONS COMPOSEES, DANS

DES PROGRAMMES

J.R. ABRIAL a construit un algorithme capable de traduire des expressions prédicatives dans des programmes avec des boucles '<u>pour</u>' [2].
Je vais tout d'abord appliquer cette idée pour traduire toute preuve composée dans un programme où il n'y a que des preuves simples et des blocs, qui sont des concepts primitifs.

Ensuite, je proposerai un algorithme pour traduire les énumérations composées en blocs d'énumération simples qui me permettra à la fin de donner une définition homogène d'énoncé composé (où les preuves et les énumérations composées sont traitées de la même façon que les transformations composées)

Je présente ensuite 4 règles qui constituent l'algorithme de traduction que je propose pour les preuves composées.

Règle 1 :

```
existe x1 :∋ <prédicat>{<preuve>}
```

est traduite par :

```
x :∋ open (méthode (pour,<prédicat>)

faire
|    x1 :∋ get(x)
|    si x1 = ∅ alors échec, down fin
|    {<preuve>}
|              si échec
|              alors up
|              fin
fin
```

Règle 2 :

```
pourtout x1 :∋ <prédicat> {<preuve>}
```

est traduite par :

```
x :∋ open (méthode (pour,<prédicat>)
faire
|     x1 :∋ get (x)
|     si x1 = ∅ alors succès fin
|     {<preuve>}
|     up
fin
```

Règle 3 :

```
<preuve1> ∧ <preuve2>
```

est traduite par :

```
faire
|     <preuve1>
|     <preuve2>
fin
```

et <preuve$_1$> ∨ <preuve$_2$>
se traduit par :

```
faire
|     <preuve1>
|     si succès
|     alors succès          [I]
|     fin
|     <preuve2>
fin
```

[I] Remarquez que le premier succès est celui de la <preuve$_1$> alors que le deuxième est celui du bloc.

Règle 4 :

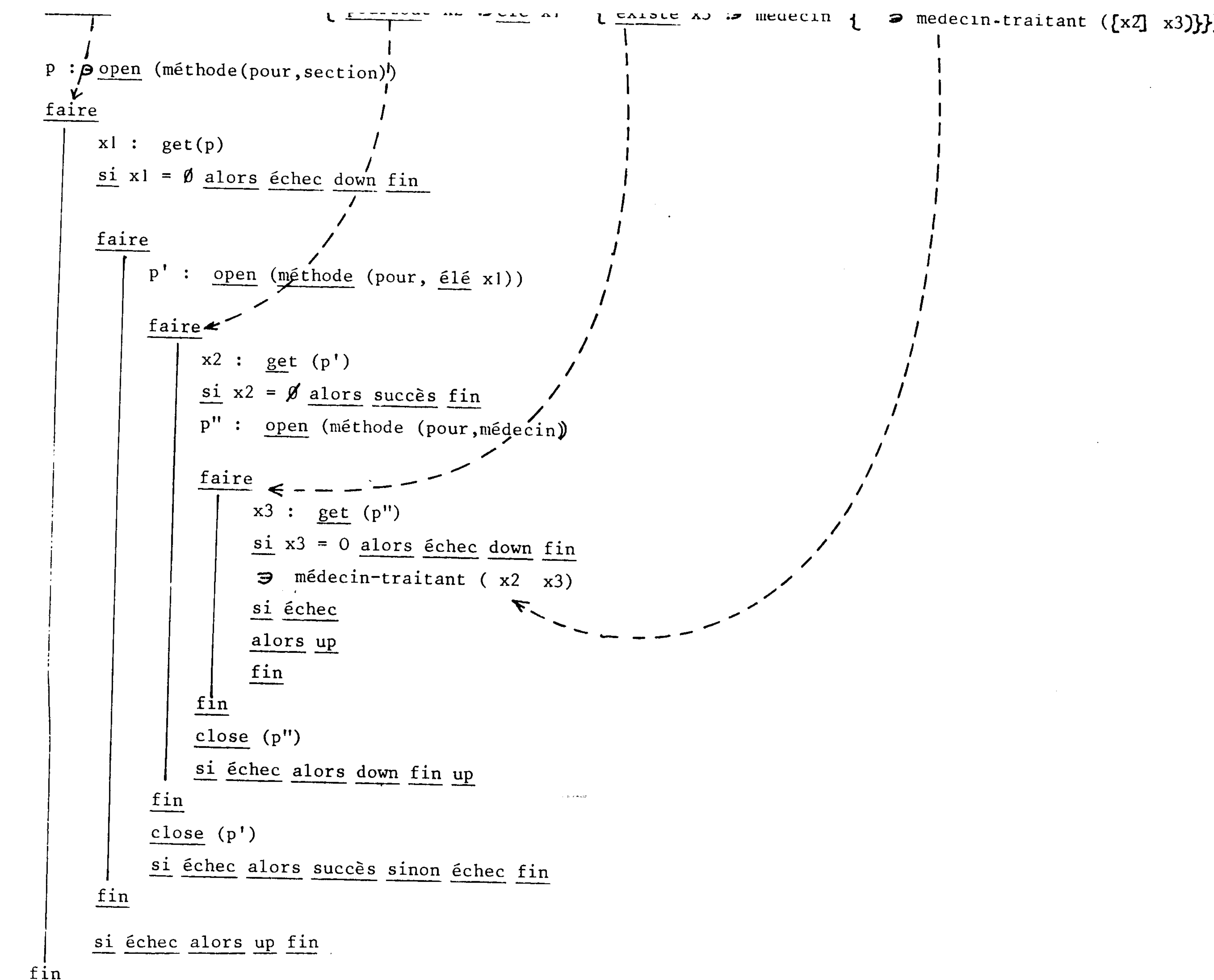
¬ {<preuve>}

est traduite par :

faire
    {<preuve>}
    si échec
    alors succès
    fin
    échec
fin

Exemple : traduction de la preuve de la page

Je vais apppliquer ces 4 règles pour traduire l'exemple de preuve sur les sections, malades, etc... dans un programme où il n'y a que des preuves simples.



médecin-traitant ({x2] x3)}}}
p : open (méthode(pour,section))
faire
x1 : get(p)
si x1 = Ø alors échec down fin
faire
p' : open (méthode (pour, élé x1))
faire
x2 : get (p')
si x2 = Ø alors succès fin
p" : open (méthode (pour,médecin))
faire
x3 : get (p")
si x3 = 0 alors échec down fin
∋ médecin-traitant (x2 x3)
si échec
alors up
fin
fin
close (p")
si échec alors down fin up
fin
close (p')
si échec alors succès sinon échec fin
fin
si échec alors up fin
fin
close (p)

1.2. Conversion de pourtout en existe et de existe en pourtout

Si on veut démontrer que :

1) "pourtout x1 :∋ <prédicat> {<preuve>}" est SUCCES si et seulement si
¬ existe x1 :∋ <prédicat> {<preuve>}" est SUCCES

et

2) "existe x1 :∋ <prédicat> {<preuve>}" est SUCCES si et seulement si
"¬pourtout x1 :∋ <prédicat> {¬<preuve>} est SUCCES

alors il suffit de démontrer que quelque soient le <prédicat> et la <preuve>, les programmes correspondants aux preuves composées écrites à gauche et à droite de chaque "si et seulement si" sont équivalents.

Je vais le faire pour 1) de façon informelle.

L'expression "pourtout x1 :∋ <prédicat> {<preuve>}" se traduit par :

```
x :∋ open (méthode (pour,<prédicat>))
faire
|   x1 :∋ get(x)
|   si x1 = Ø alors succès fin
|   {<preuve>}
|   si échec
|   alors down
|   fin
|   up
fin
```

et l'expression "¬ existe x1 :∋<prédicat> { ¬ {<preuve>}}" se traduit par :

```
faire
    x :∋ open (méthode (pour,<prédicat>))
    faire
        x1 :∋ get(x)
        si x1 = Ø alors échec down fin
        faire
            {<preuve>}
            si échec
            alors succès
            sinon échec
            fin
        fin
        si échec
        alors up
        fin
    fin
    si échec
    alors succès
    sinon échec
    fin
fin
```

Ce programe est trivialement équivalent au suivant :

```
faire
    x :∋ open (méthode (pour,<prédicat>))
    faire
        x1 :∋ get(x)
        si x1 = Ø alors échec down fin
        {<preuve>}
        si échec alors succès sinon up fin
    fin
    si échec alors succès sinon échec fin
fin
```

qui est lui-même équivalent au programme :

```
faire
    x :∋ open (méthode (pour, <prédicat> ))
    faire
        x1 :∋ get(x)
        si x1 = 0 alors succès fin
        {<preuve>}
        si échec
        alors échec
        sinon up
        fin
    fin
fin
```

lui-même équivalent au programme :

```
faire
    x :∋ open (méthode (pour,<prédicat>))
    faire
        x1 :∋ get(x)
        si x1 = 0 alors succès fin
        {<preuve>}
        up
    fin
fin
```

qui est celui qui traduit la sémantique de la partie gauche de l'équivalence 1).

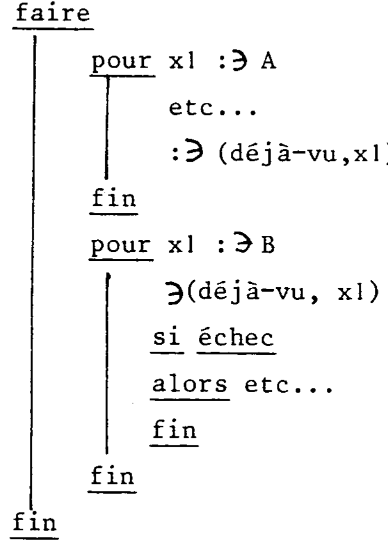
2. ÉNUMÉRATIONS COMPOSÉES

Les énumérations composées, c'est-à-dire d'ensembles qui sont l'union (∪) l'intersection (∩) et le complément (∁) d'extensions de prédicats, se traduisent dans les blocs contenant des énumérations simples.

Je vais traduire comme exemple l'énumération de l'union de deux ensembles A et B :

```
pour  x1 :∋  A ∪ B
|
|      etc...
fin
```

se traduit en :



3. CONCLUSION

Je peux, enfin, définir un énoncé de problème comme suit :

DEFINITION : Un énoncé de problème est soit :

* un énoncé de transformation, preuve ou énumération simple

soit :

* une suite (bloc) d'énoncés de problèmes.

PARTIE 4

CONCLUSION

BILAN

La suite logique du formalisme présenté est, à mon avis, l'expérimentation avec un prototype de système.

C'est le meilleur moyen de savoir quels mécanismes devra posseder une réalisation plus complète.

En effet, c'est en programmant avec SOCRATE-P [1] que j'ai pris un premier contact, toutefois insuffisant, avec les difficultés et les avantages de ce genre de programmation. Je rapporterai mes conclusions et résultats dans la suite de cette partie et j'indiquerai des directions de recherche (sur le plan pratique ou formel) qui en découlent.

Avant celà je présente rapidement SOCRATE-P.

PROTOTYPE SOCRATE-P

Le but de l'implémentation SOCRATE-P était de permettre le test rapide des nouveaux concepts sans souci de performances qui sont évidemment indispensables dans une réalisation plus complète.

L'interpréteur SOCRATE-P est écrit en SOCRATE, pour permettre une implémentation dans le délai permis par la durée du Projet SOCRATE.

J.C. FAVRE l'a réussie dans un temps record et on pourra y trouver un grand nombre des concepts que j'ai exposés dans les parties 2 et 3.

Ce qui m'a le plus gêné, c'est l'absence des 'reln' (pages II.9 et II.26), paramètres explicites (page II.32) (surtout dans l'exemple de Gestion), l'impossibilité de définir plusieurs méthodes de résolution d'un problème (cf. exemple dans la page III.51) ou d'itérer sur des méthodes ou autres objets d'ordre $\geqslant 2$ (en termes de PLANNER, cela veut dire que les tests de conformité-"pattern-matching"- sont à un seul niveau.)

Le calcul numérique n'est pas facile tout d'abord en raison de la façon dont sont stockés les nombres, et ensuite parce que les opérations arithmétiques exigent la définition de procédures SOCRATE.

Je n'ai senti la difficulté que dans l'exemple de Gestion qui est essentiellement numérique.

Je pense d'ailleurs qu'il est très important que le calcul numérique puisse être étendu aux nombres réels.

Enfin, je crois qu'il faut implémenter le mécanisme des espaces locaux. C'est certainement difficile, parce que toute solution qui consisterait à faire une deuxième base de données est évidemment à rejeter.

Je ne parle pas ici des aménagements syntaxiques absolument nécessaires dans une réalisation définitive parce que je m'intéresse seulement à la sémantique.

Dans l'annexe, je présente les 4 exemples dont j'ai parlé

Pour chacun, je donne le Modèle Sémantique Complet sous deux formes: en Français (ps. IV.15 IV40 IV 65 bis) et défini, rigoureusement. Chaque méthode est présentée d'abord en langage sémantique et ensuite en SOCRATE-P.

Après les méthodes, je donne les "listings" avec les résultats obtenus en chaque manipulation.

LES MÉTHODES

La première chose que j'ai apprise avec SOCRATE-P est que la manipulation des méthodes est difficile et ceci essentiellement sous deux points de vue :

1 - correction de l'ensemble du Modèle Sémantique
2 - dépendance vis-à-vis des données.

La correction est très difficile à contrôler et prouver surtout parce que :

* les règles sémantiques sont indissociables des choix de stratégie pour leur application, (c'est le principe même des METHODES)
* la structure de contrôle est très flexible.

Ce qu'on perd avec ces deux propriétés est gagné en efficacité.

Je vais illustrer le compromis correction-efficacité à l'aide de deux exemples SOCRATE-P.

Exemple de DROIT (page IV-13)

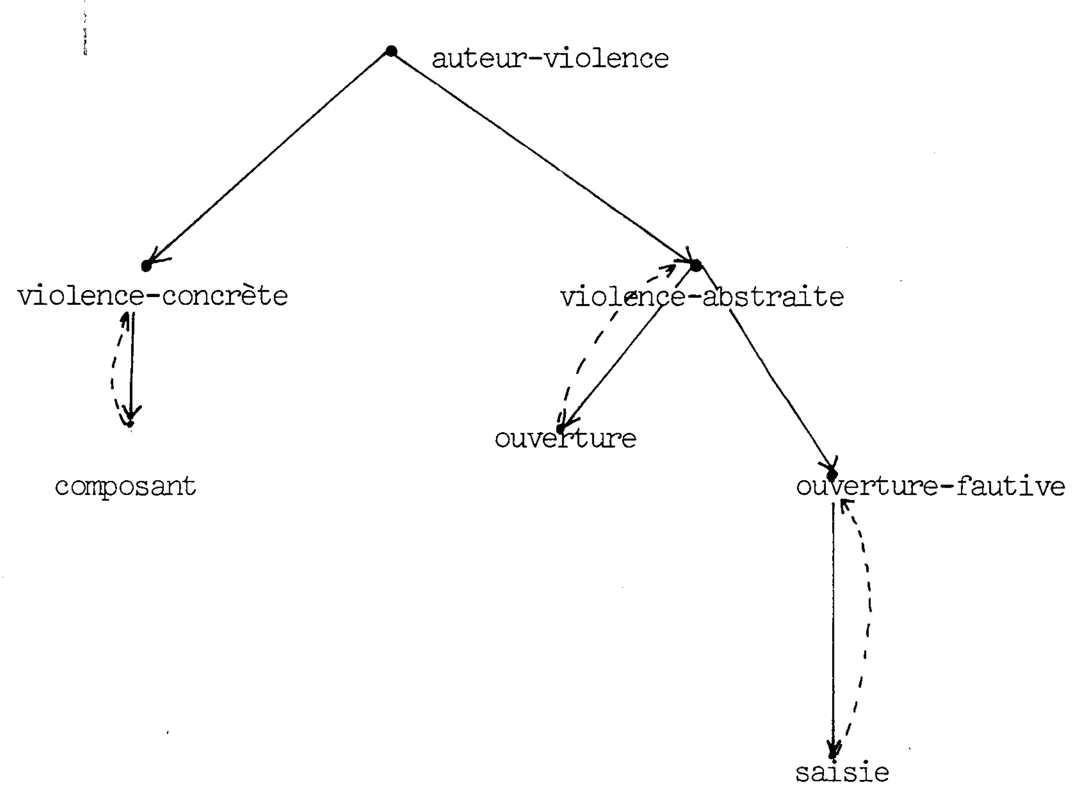
Je rappelle dans cet exemple, qu'il s'agit de méthodes pour prouver que quelqu'un a violé un certain domicile.

Les méthodes pourraient être définies de la façon suivante :

```
mth((∋,auteur-violence),,,faire
                          | ∋ auteur-violence,violence- concrète ([x1] ,x2)
                          |   si échec
                          |   alors
                          |   | ∋  auteur-violence, violence-abstraite([x1] ,x2)
                          |   fin
                          fin)

mth((∋,auteur-violence,violence-concrète),,
    faire
    |  existe x3 :∋ composant [x1] {∋ briseur ([x3] x2)}
    fin)

mth((∋,auteur-violence,violence-abstraite),,,
    faire
    |  existe x3 :∋ ouverture (x1) {∋ ouvreur ([x3] x2) ∧
    |                               ∋ ouverture-fautif(x3)}
    fin)
```

```
mth(∋,ouverture-fautive),,,faire
                        | ¬ ∋ propriétaire ([chose-ouverte[x1]],ouvreur[x1])
                        |  ∋ clé-fausse(clé[x1]) V
                        |   V existe x2 :∋ saisie (clé[x1],ouvreur[x1])
                        |        {¬ ∋ propriétaire ([chose-ouverte[x1]],
                        |                           donneur[x2])}
                        fin)
```

définition qui est assez proche du langage de la Logique. Le structure de contrôle est :



Dans ce schéma, les transferts en pseudo-parallèle sont les transferts que contiennent implicitement les boucles "pour".

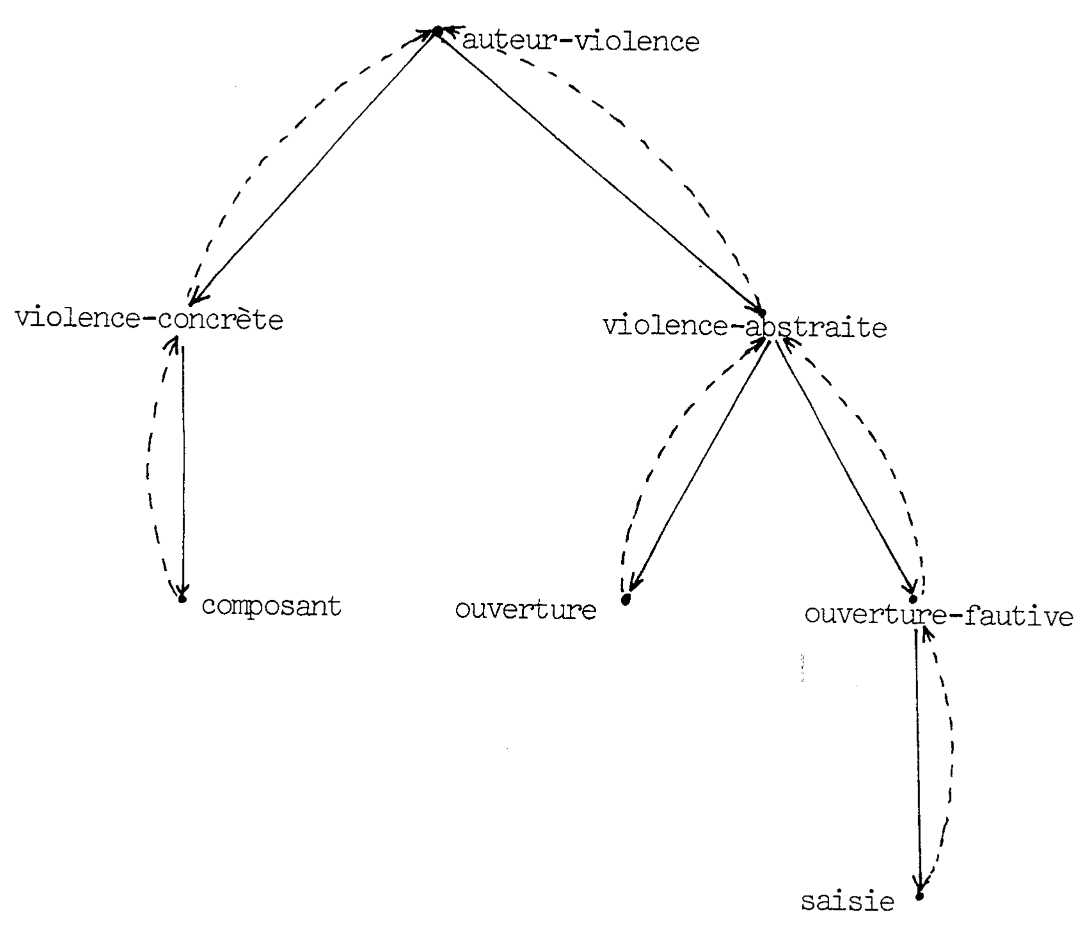
Simplement, cet ensemble de méthodes n'est pas optimisé du point de vue du coût parce que la preuve :

∋ auteur-violence([a] b)

pourra être très longue.

Par exemple, si b est entré en a avec une clé-fausse. La raison est que la "méthode violence-abstraite" ne sera appliquée qu'en cas d'échec total de la méthode violence. (cf. page IV.34)

Si on veut améliorer le coût, on peut définir le Modèle comme en page IV.15, ce qui donne la structure de contrôle :



Les deux méthodes sont gérées en pseudo-parallèle.

Evidemment, l'ajoût du pseudo-parallèlisme se traduit par l'ajoût d'une structure de contrôle explicite (get, resume, si, alors, fin, etc...) et des étiquettes échec,1, échec,2 etc... qui obscurcissent la sémantique et ont des conséquences sur la correction.

Une deuxième mauvaise incidence de la flexibilité de la structure de contrôle sur la correction est dûe au mécanisme d'appel des méthodes : si en ALGOL on a un appel de procédure :

call toto (a,b)

la simple vue du programme permet de savoir que la procédure a été appelée.

En Langage Sémantique, si on a :

```
pour x :∋ bénéfice[a]
|      etc ...
fin
```

rien ne m'indique si l'interprétation a eu recours à une méthode (pour,bénéfice) ou non. Plus que çà, les méthodes changent tout le temps et il faudrait savoir quelle méthode aurait été appliquée au cas où il y en aurait une.

Dans les exemples SOCRATE-P j'ai adopté le système suivant pour contrôler les applications de méthodes : j'ai ajouté au début et à la fin deux messages typiques de la méthode (par exemple dans les cubes, les messages "je-vais-enlever" etc... (page IV-49)

Ceci facilite beaucoup le contrôle de correction.

Un autre mécanisme qui peut être utilisé est la mise-à-jour de raison (page III.16)

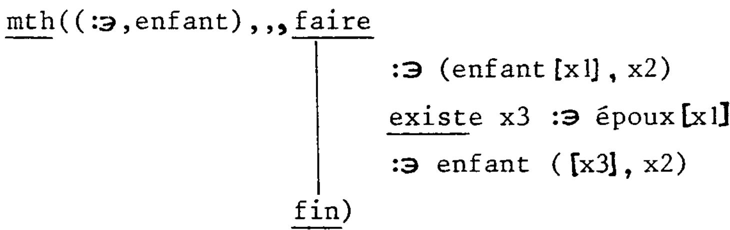
Dépendance vis-à-vis de l'état des données : exemple des descendants

J'ai exécuté un exemple très simple qui montre la dépendance des méthodes vis-à-vis de l'état des données.

Si je veux exprimer la règle sémantique suivante :

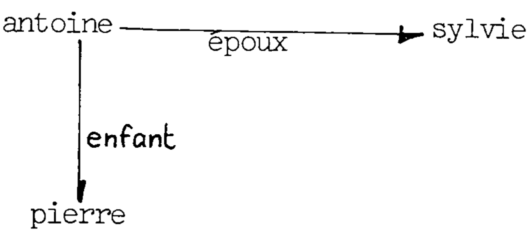
"les enfants d'une personne sont les mêmes que ceux de son époux"

Je peux définir la méthode suivante :



Cependant, cette méthode ne garantit pas la règle sémantique dans certains états des données.

Ainsi, si,par exemple, celui-ci est :

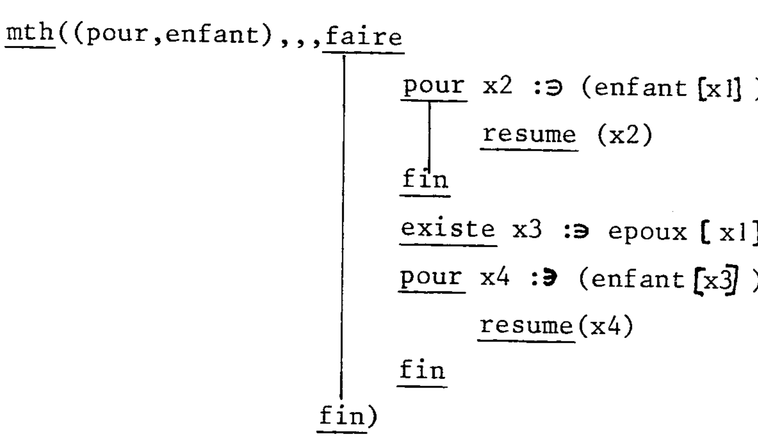


quand on définit la méthode (:∋, enfant),et on écrit :

:∋ enfant ([antoine] jacques)

alors à la fin 'jacques' est le seul enfant de sylvie.

En fait, la sémantique décrite serait mieux représentée par la méthode suivante :



On voit donc que, si, apparamment, on peut stocker une règle sémantique :

$P \rightarrow Q$

mth(:∋ ,P)
|
| :∋Q
|

ou

mth(∋ ,Q)
|
| ∋P

ces deux formes ne sont pas équivalentes, contrairement à ce que l'on peut penser [35].

Risque de boucles

Sur ce même exemple, on peut constater qu'il suffirait de mettre :

pour x4 :∋ enfant ([x1],x2)
|
fin

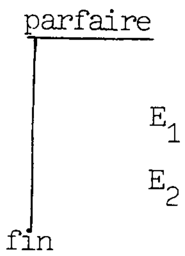
dans la première méthode pour qu'elle "boucle".

Le risque de boucles est un autre point délicat de la définition des METHODES.

Parallèlisme

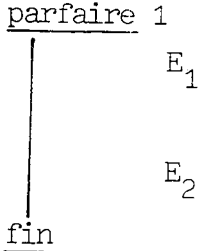
La résolution de problèmes en pseudo-parallèle ou parallèle (si on dispose du hardware nécessaire) me paraît une direction de recherche privilégiée sur le plan du formalisme.

On devrait pouvoir définir des blocs par-faire. Un bloc

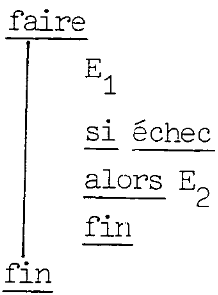


voudrait dire "résoudre par un ordre quelconque les deux problèmes E_1 et E_2".

Dans ces blocs on remplacerait la construction "si échec", par l'indication du nombre de problèmes à résoudre pour avoir SUCCES du bloc.



aurait alors une sémantique analogue à :



Au niveau des processus, il faudrait étudier l'extension du mécanisme de réveil au mode multitraitement, tel qu'il existe en [40] p.ex.

UN NIVEAU INTERMÉDIAIRE ?

Les répercussions de l'optimisation des méthodes sur la clarté de structure des Modèles, que j'ai analysées et qui ne feraient que s'aggraver si on ajoutait encore de nouvelles primitives de contrôle propres au parallélisme, pourraient peut-être se réduire ou même éliminer si on introduisait un niveau intermédiaire entre le niveau sémantique et le niveau mémoire.

L'idée de ce 3ième niveau est de J.R. ABRIAL qui le voit comme celui auquel on optimiserait le code "objet" produit par la décomposition du problème en sous-problèmes.

Pour bien fixer les idées, on peut concevoir que, de DROIT, le code "objet" correspondrait aux méthodes que j'ai définies en

Evidemment, ce 3ième niveau demande des études assez poussées des mécanismes de résolution ce qui constitue une direction de recherche à long terme.

NIVEAU MÉMOIRE : ORGANISATION DES DONNÉES

L'optimisation au niveau sémantique (ou éventuellement à un niveau intermédiaire) ne suffit pas à rendre les résolutions efficaces.

Il faut aussi diminuer le coût de l'accès aux extensions des prédicats et pour cela il faut, entre autres, organiser celles-ci de façon convenable.

Chaque espace est constitué par un ensemble d'extensions de prédicats.

La première décision à prendre,quand ont veut organiser les données, consiste à choisir pour chaque couple de fonctions d'accès laquelle on va stocker, ou si on stocke les deux.

Quand on a choisi de stocker une fonction d'accès F on peut organiser les éléments de chaque prédicat F[X] soit en <u>tableau</u>, soit en <u>chaîne</u> (simple ou double) ce qui dépend du fait que le cardinal des prédicats F x varie peu ou beaucoup avec x.

On supperpose ensuite à cette organisation, l'organisation de l'ensemble des extensions de prédicats.

Soit on les groupe par fonctions d'accès (on retrouve alors un analogue de l'organisation verticale de SOCRATE), soit les groupes par objets.

Organisation par objets. Stockage des 'reln'

L'organisation par objets consiste à grouper tous les prédicats F_1 a , F_2 a , F_3 a où F_1 F_2 et F_3 etc... sont différentes fonctions d'accès. C'est l'analogue de l'"organisation horizontale" de SOCRATE.

Cette organisation est particulièrement adaptée au stockage des '<u>reln</u>'

Une <u>reln</u> est déclarée par n fonctions d'accès F_1 F_2 ... F_n.

Si on veut connaître les arguments d'une réalisation, il suffit d'attribuer un indice à chacune des n fonctions d'accès et à calculer leur position de F_1 r , F_2 r , F_3 r ... à partir de cet indice.

Aussi cela rend facile l'interprétation de :

<u>existe</u> x :∍ R(a_1,a_2,a_3 ... a_n)

ou encore :

<u>pour</u> x :∍ R(a_1,a_2)

| etc...

<u>fin</u>

Adressage dispersé

Dans le cas d'extensions de prédicats ayant cardinal presque toujours égal à zéro, on peut employer une technique d'adressage dispersé.

Comment choisir l'organisation ?

Le fait de décider de stocker une fonction d'accès F ou de laisser que le système la calcule à partir de son inverse, peut avoir un impact très grand ur le coût de l'accès aux prédicats F[x].

Quand on décide de stocker F, l'organisation choisie pour les extensions des prédicats F [x], et pour leur ensemble est elle aussi très importante.

Le système pourra se charger d'espionner une application, c'est-à-dire de mesurer la fréquence des accès aux différents prédicats, ce qui peut être très utile pour choisir une organisation.

L'automatisation du choix ou changement de structure ne sont pas impossibles à concevoir mais je crois qu'en ce moment ils relèvent de la science fiction. J'ai l'intention de continuer le travail de recherche que j'ai fait mais dans le sens d'une automatisation partielle.

APPLICATIONS À COURT-TERME

Les personnes qui ne voudraient pas changer SOCRATE ou se lancer dans l'implémentation d'un nouveau système général de bases de données, peuvent utiliser les fonctions d'accès et les méthodes pour construire les systèmes ad-hoc; comme a fait F. PECCOUD en[20](cf. par exemple 2.3.5 bis, 2.3.6.2., 4.3.3.)

On peut se servir des algorithmes décrits en Partie 1 pour "rajouter" à SOCRATE un algorithme qui réécrit les programmes quand la structure change (page I.43) et un autre qui aide à choisir la structure (page I.46). Ces algorithmes s'écrivent facilement dans le langage SOCRATE lui-même, mais il faut étudier un peut plus la syntaxe du langage (en particulier, l'emploi de tout, un et existe).

Si on ne veut pas toucher ou étendre SOCRATE, on peut se servir du formalisme des Parties 2 et 3 pour présenter des applications de grande taille (par exemple, comme a fait Dominique PORTAL dans [19] paragraphes 1.1 et 5.2.) ou encore essayer de mettre en place une méthodologie de programmation "structurée" en SOCRATE.

Pour cela, il est utile de se référer aux ouvrages [24] et [25] et de faire l'étude d'une pédagogie analogue à celle exposée en [31] et [32] pour le domaine du COBOL et du FORTRAN.

CONCLUSION

Je ne voudrais pas "conclure" la CONCLUSION sans avouer mon profond sentiment que quelques uns des nouveaux concepts que j'ai exposés aideront à mettre en place une nouvelle manière de programmer en bases de données.

Si je le crois, ce n'est pas en raison de ma compétence à résoudre les obstables que cela va poser, mais parce que ces concepts s'inscrivent dans des recherches actuelles en programmation, dont on pourra bénéficier.

A N N E X E

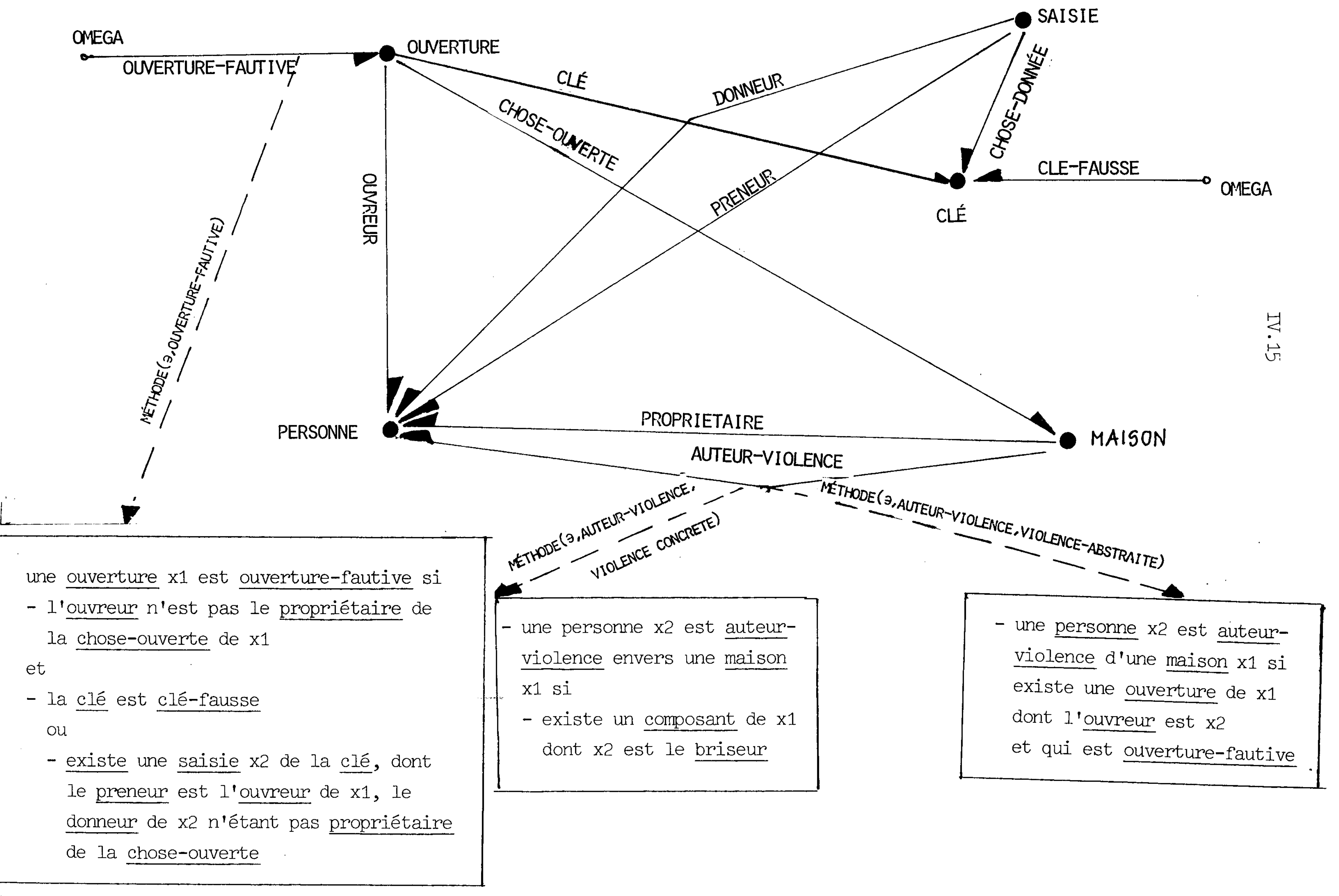
EXEMPLES QUI ONT ÉTÉ IMPLÉMENTÉS EN

SOCRATE-P

EXEMPLE DE DROIT

Cet exemple a été discuté au début de cette Partie.
L'aspect le plus intéressant est l'emploi du pseudo-parallèlisme.

J'ai aussi fait une manipulation dans laquelle j'ai ajouté et supprimé des données pendant le processus de preuves.


OMEGA
OUVERTURE-FAUTIVE
OUVERTURE
CLÉ
CHOSE-OUVERTE
DONNEUR
SAISIE
CHOSE-DONNÉE
CLE-FAUSSE
OMEGA
CLÉ
PRENEUR
OUVREUR
MÉTHODE(∋,OUVERTURE-FAUTIVE)
PERSONNE
PROPRIETAIRE
AUTEUR-VIOLENCE
MAISON
MÉTHODE(∋,AUTEUR-VIOLENCE, VIOLENCE CONCRETE)
MÉTHODE(∋,AUTEUR-VIOLENCE,VIOLENCE-ABSTRAITE)
une ouverture x1 est ouverture-fautive si
- l'ouvreur n'est pas le propriétaire de la chose-ouverte de x1
et
- la clé est clé-fausse
ou
- existe une saisie x2 de la clé, dont le preneur est l'ouvreur de x1, le donneur de x2 n'étant pas propriétaire de la chose-ouverte
- une personne x2 est auteur-violence envers une maison x1 si
- existe un composant de x1 dont x2 est le briseur
- une personne x2 est auteur-violence d'une maison x1 si existe une ouverture de x1 dont l'ouvreur est x2 et qui est ouverture-fautive

M E T H O D E S

```
mth((∋ ,auteur-violence),,,
   faire
   |     t1  :∋ open(méthode(∋ ,auteur-violence,violence-abstrte))
   |     t2  :∋ open(méthode(∋ ,auteur-violence,violence-concret))
   |     faire
   |     |     get (t1)
   |     |     si échec
   |     |     alors get (t2)
   |     |     |     si échec
   |     |     |     alors up
   |     |     |     fin
   |     |     fin
   |     fin
   fin)
```

```
mth((∋ ,auteur-violence,violence-concret),,,
   faire
   |     pour  x3 : ∋ composant [x1]
   |     |        ∋ briseur ([x3] x2)
   |     |     si succès
   |     |     alors succès
   |     |     sinon échec,1
   |     |     fin
   |     |     resume
   |     fin
   |     échec,2
   fin)
```

```
proc violence-concret pour x3 := composant de x1
REP1
si x2 = briseur de x3 alors succes sinon m x8 := 1 ,
REP1
echec , a à 1 nom ( x8 ) fin , resume fin , m x8 :=
REP1
2 , echec , scall nom ( x8 ) fproc ?
 SUCCES       : 01
REP1
```

```
mth((∋,violence-abstrte),,,
   faire
      print('est-ce-que',x2,'commit','violence-abstrte',x1)
      pour x3 :∋ inv[chose-ouverte][x1]
         ¬ ∋ ouvreur([x3] x2)
         si succès
         alors échec,1
               resume
               up
         fin
            ouvertur-fautif(x3)
            resume
      fin
      échec,2
   fin)
```

```
     proc violence-abstrte
   FEP1
   m x8 := est-ce-que , m x9 := commit , m x7 := violence-abstrte ,
   REP1
   call message( 'x8 x2 x9 x7 x1 x6 ), '
   FEP1
   pour x3 := l-chose-ouverte de x1
   REP2
   si x2 ¬= ouvreur de x3
   REP2
   alors m x6 := 1 , echec , scall nom ( x6 ) , resume ,
   REP1
   up ↓ fin , call ouvertur-fautif ( x3 ) , resume fin ,
   FEP1
   m x6 := 2 , echec , scall nom ( x6 ) fproc ?

    SUCCES          : 01
```

```
mth((∋ ,ouverture-fautif),,,
    faire
    |   x2 :∋  clé [x1]
    |   x3 :∋  ouvreur [x1]
    |   x4 :∋  chose-ouverte [x1]
    |   print ('est-ce-que', x3, 'ouverture-fautif')
    |    ∋  propriétaire ([x4], x3)
    |   si succès
    |   alors échec,1
    |   |     resume
    |   fin
    |    ∋ clé-fausse (x2)
    |   si succès
    |   alors succès,1
    |   |     resume
    |   fin
    |   pour x5 :∋  inv [chose-donnée][x2] ∩ inv [prenneur][x3]
    |   |   x6 :∋ donneur [x5]
    |   |   ¬ ∋ propriétaire ([x4] , x6)
    |   |   si succès
    |   |   alors succès,2
    |   |   |     resume
    |   |   fin
    |   fin
    |   echec,2
    fin)
```

```
proc ouvertur-fautif
REP1
m x8 := est-ce-que , m x9 : =ouvertur-fautif¢
m x8 := est-ce-que , m x9 := ouvertur-fautif ,
REP1
m x2 := cle de x1, m x3 := ouvreur de x1 , m x4 :=
REP1
chose-ouverte de x1 , call message ( x8 x3 x9
REP1
x6 x7 )
REP2
, si x3 = proprietaire de x4
REP2

alors m x6 := 1 , echec , scall nom ( x6 ) ,
REP1
resume fin ,
REP1
si x2 = cle-fausse
REP2
alors m x6 := 1 , succes scall nom x6@( x6 ) ,

alors m x6 := 1 , succes,@ , scall nom ( x6 ) ,
REP1
resume fin ,
REP1
pour x5 := ( i-chose-donnee de x2 et
REP1
i-preneur de x3 )
REP2
m x6 := donneur de x5 , si x6 ¬= proprietaire de x4
REP2
alors m x7 := 2 , succes , scall nom ( x7 ) , resume ,@
REP2
fin fin , m x7 := 2 , echec , scall nom ( x7 ) fproc ?
 SUCCES              : 01
REP1
```

MANIPULATIONS

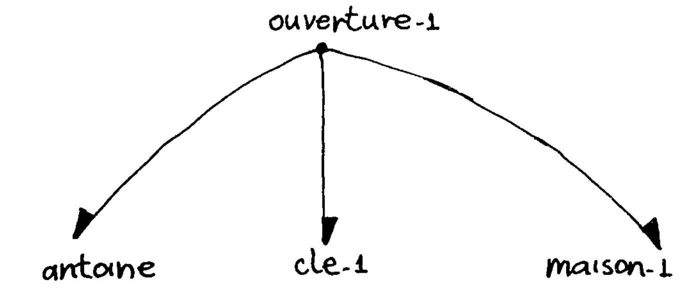
```
pour x1 := saisie scall nom ( x1 ) fin ,
REP1
pour x1 := ouverture scall nom ( x1) fin ,
REP1
pour x1 := maison sca@@@pour x2 : = composant de x1 pour
REP1

x3 : = b lseur de x2 scall nom ( x3 ) fin fin fin ?

 SUCCES             : 01

REP1
```

aucune information sur les ouvertures, saisies ou brise de composants

```
m x1 := maison-1 , m x2 := proprietaire de x1 , scall nom ( x2 ) ?
JEAN
 SUCCES             : 01
REP1
```

jean —— proprietaire —— maison-1

```
pour x1 := ouverture m x2 := ouvreur de x1 , m x3 : = chose-ouverte de
REP1
x1, m x4 := cle de x1 , call message (  x2 x4 x3  x5 x6 x7) fin ? ,'
ANTOINE
CLE-1
MAISON-1 SUCCES            : )2REP
                             1
```



```
g saisie-1 , m x1 := saisie-1 , a x1 est saisie ?

 SUCCES              : 01
REP1

m x2 := sylvie , m x3 := antoine , m x4 := cle-1 ,
REP1

call gmth-saisie ( x1 x2 x3 x4 ) ?

 SUCCES              : 01
REP1

m x1 := maison-1 , m x2 := antoine , open t1 := violence-abstrte
REP2

( x1 x2 ) ,

REP1

get t1 ?

EST-CE-QUE
ANTOINE
COMMIT
VIOLENCE-ABSTRTE
MAISON-1
EST-CE-QUE
ANTOINE
OUVERTUR-FAUTIF
SUCCES
2
 SUCCES              : 01
REP1
```

ouverture-1
saisie1
antoine
clé-1
maison-1
sylvie
jean
propriétaire

← — — — — — *succès* parce que la clé-1 a été donnée à antoine par quelqu'un qui n'est pas propriétaire (raison 2 du succès)

```
g ouverture-2 , m x2 := ouverture-2 , m x3 := jacques ,
REP1

m x4 := maison-2 , call gmth-ouverture ( x2 x3 x1 x4 ) ?

 SUCCES             : 01
REP1

a x1 est cle-fausse  ?

 SUCCES             : 01
REP1

open t1 := violence-abstrte ( x4 x3 ) ,

REP1

get t1 ?

EST-CE-QUE
JACQUES
COMMIT
VIOLENCE-ABSTRTE
MAISON-2
EST-CE-QUE
JACQUES
OUVERTUR-FAUTIF
SUCCES
1
 SUCCES             : 01
REP1
```

ouverture-2 → jacques, cle-2, maison-2
cle-2 — cle-fausse

← — — — succes de la methode (ouverture-fautive) parce que jacques ouvrit la maison-2 avec cle-fausse (raison 1)

```
REP1
open t1 := violence-abstrte ( x1 x2 ) ,

REP1
get t1 ?

EST-CE-QUE
PIERRE
COMMIT
VIOLENCE-ABSTRTE
MAISON-1
EST-CE-QUE
PIERRE
OUVERTUR-FAUTIF
ECHEC
2
 SUCCES              : 01
REP1


g saisie-2 , m x1 := saisie-2 , a x1 est saisie ,

REP1
m x2 := marie , m x3 := pierre , m x4 := cle-2 ,

REP1

call gmth-saisie ( x1 x2 x3 x4 ) ?

 SUCCES              : 01
REP1


get t1 ?

EST-CE-QUE
PIERRE
OUVERTUR-FAUTIF
SUCCES
2
 SUCCES              : 01
REP1
```

TIONS **PENDANT** LA PREUVE

état-actuel : ouverture-1 ouverture-2

saisie-1

ajout de : saisie-2

pierre

cle-2

marie

[illegible] marie

jean

← — — — — le succes est dû à l'ajout d'informations cueillies alors que la preuve avait déjà commencé

avait déjà commencé

```
get tl ?
ECHEC
2
 SUCCES          : 01
REP1
get tl ?
 SUCCES          : 02
REP1
```

```
REP1
pour x1 := saisie scall nom X0( x1 ) fin ?
 SUCCES          : 01
REP1



g ouverture-1 , m x1 := ouverture-1 , a x1 est ouverture ,
REP1
m x2 := pierre , m x3 := cle-1 , m x4 := maison-1 ,
REP1
call gnth-ouverture ( x1 x2 x3 x4 ) ?

 SUCCES          : 01
REP1


g ouverture-2 , m x1 := ouverture-2 , a x1 est ouverture ,
REP1
m x2 := pierre , m x3 := cle-2 , m x4 := maison-1 ,
REP1
call gnth-ouverture ( x1 x2 x3 x4 ) ?

 SUCCES          : 01
REP1


g saisie-1 , m x1 := saisie-1 , a x1 est saisie ,
REP1
m x2 := jeanne , m x3 := pierre , m x4 := cle-1 ,
REP1
call gnth-saisie ( x1 x2 x3 x4 ) ?

 SUCCES          : 01
```

```
 SUCCES            : 01
 REP1

s saisie-2 , m x1 := saisie-2 , m x@@@a x1 est saisie ,
REP1
m x2 := marie , m x3 := pierre , m x4 := cle-2 ,
REP1
call gmth-saisie ( x1 x2 x3 x4 ) ?

 SUCCES            : 01
REP1

m x1 := maison-1 , m x2 := proprietaire de x1 , scall nom ( x2 ) ?

 SUCCES            : 02
REP1

m x2 := jean , a x2 est proprietaire de x1 , m x@@@@@?

 SUCCES            : 01
REP1

m x2 := proprietaire de x1 , scall nom ( x2 )  ?

JEAN
 SUCCES            : 01
REP1

m x1 := maison-1 , m x2 := pierre ,
REP1

open t1 := violence-abstrte ( x1 x2 ) ,

REP1

get t1 ?
```

```
REP1   ... ; x2 := pierre ,

open t1 := violence-abstrte ( x1 x2 ) ,

REP1

get t1 ?

EST-CE-QUE
PIERRE
COMMIT
VIOLENCE-ABSTRTE
MAISON-1
E
EST-CE-QUE
PIERRE
OUVERTUR-FAUTIF
ECHEC
2
 SUCCES              : 01
REP1

get t1 ?

EST-CE-QUE
PIERRE
OUVERTUR-FAUTIF
SUCCES
2
 SUCCES              : 01
REP1

get t1 ?

ECHEC
2
 SUCCES              : 01
REP1
```

← échec parceque la 1ère ouverture avec la clé-1 n'est pas fautive (la seule sortie constatée est en règle) - voir ... 2)

← succès parceque la 2ème ouverture est fautive : la clé a été donnée par quelqu'un qui n'est pas propriétaire

EXEMPLE DE SUPPRESSION D'INFORMATIONS <u>PENDANT</u> LA PREUVE

```
open t1 := violence-abstrte ( x1  x2 ) ,

REP1


get t1 ?


EST-CE-QUE
PIERRE
COMMIT
VIOLENCE-ABSTRTE
MAISON-1
EST-CE-QUE
PIERRE
OUVERTUR-FAUTIF
ECHEC
2
 SUCCES              : 01
REP1


t saisie-2 ,        <- - - - - - - -


REP1


get t1 ?


EST-CE-QUE
PIERRE
OUVERTUR-FAUTIF
ECHEC               - <- - - - - - -
2
 SUCCES              : 01
REP1


get t1 ?

ECHEC                <- - - - - - -
2
 SUCCES              : 01
```

pendant la preuve, on a constaté la fausseté de la saisie-2 qui était fautive

echec, au contraire du cas precedent et dû à la suppression de la saisie

echec parce qu'il n'y a plus d'au-...

```
2
SUCCES          : 01
REP1
Set c1 ?

SUCCES          : 02
REP1
```

◁ — — — — — <u>echec</u> parce qu'il n'y a plus d'ouvertures constatées.

exemple de "troubles sémantiques" (page IV-3)

```
m x1 := maison-1 , pour x2 := composant de x1 scall nom ( x2 ) fin ?

PORTE-1-1
PORTE-1-2
PORTE-1-3
FENETRE-1-1
FENETRE-1-2
FENETRE-1-3
FENETRE-1-4
FENETRE-1-5
FENETRE-1-6
 SUCCES            : 01
REP1
```

maison-1

porte-1-1 ... porte 1-3 fenêtre-1-1 ... fenêtre-1-6

briseur

jacques

```
m x2 := jacques , m x3 := fenetre-1-6 ,
REP1

a x2 est briseur de x3 ,

REP1

open t1 := violence-concret ( x1  x2 ) ,

REP1

get t1 ?

ECHEC
1
 SUCCES            : 01
REP1
get t1¢
```

```
get t1 ?

ECHEC
1
 SUCCES             : 01
REP1


get t1 ?

ECHEC
1
 SUCCES             : 01
REP1


get t1

REP2
?

ECHEC
1
 SUCCES             : 01
REP1


get t1 ?

ECHEC
1
 SUCCES             : 01
REP1
```

```
get t1 ?

ECHEC
1
 SUCCES              : 01
REP1


get t1 ?

ECHEC
1
 SUCCES              : 01
REP1


get t1 ?

ECHEC
1
 SUCCES              : 01
REP1


get t1 ?

SUCCES
 SUCCES              : 01
REP1
```

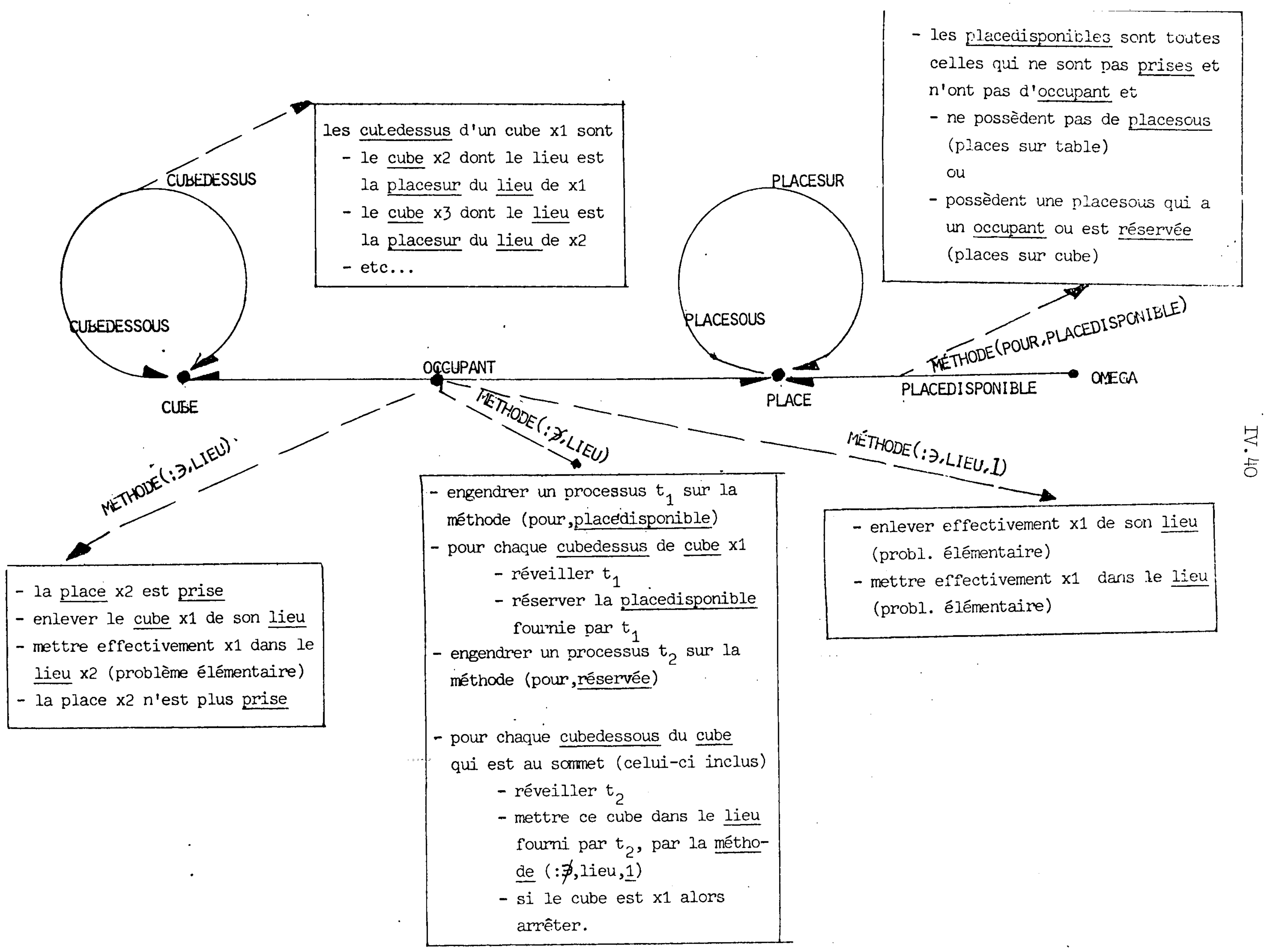
← - - - - le succes n'est obtenu qu'à la 9ième tentative

EXEMPLE DES CUBES

Dans cet exemple, la méthode (:$\not\ni$,lieu) employée pour enlever un cube de son lieu est curieuse : l'évaluation préalable des chances de SUCCES remplace la résolution locale du problème suivie éventuellement de validation ou retour en arrière.

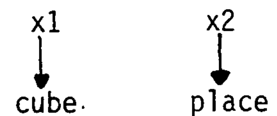
Le lecteur pourra remarquer aussi la façon dont j'ai exprimé que la relation "cubedessus" est la fermeture transitive de "cubesur" (occupant de la placesur du lieu).

Enfin, il peut trouver une illustration de concepts d'ordre d'énumération : les placesdisponibles (page IV.44) sont énumérées par un ordre qui évite la création de piles de cubes et ceci afin de rendre les futurs déplacements de cubes moins coûteux.



les cubedessus d'un cube x1 sont
- le cube x2 dont le lieu est la placesur du lieu de x1
- le cube x3 dont le lieu est la placesur du lieu de x2
- etc...
CUBEDESSUS
CUBEDESSOUS
CUBE
OCCUPANT
PLACESUR
PLACESOUS
PLACE
PLACEDISPONIBLE
OMEGA
MÉTHODE(POUR,PLACEDISPONIBLE)
- les placedisponibles sont toutes celles qui ne sont pas prises et n'ont pas d'occupant et
- ne possèdent pas de placesous (places sur table)
ou
- possèdent une placesous qui a un occupant ou est réservée (places sur cube)
MÉTHODE(:∃,LIEU)
- la place x2 est prise
- enlever le cube x1 de son lieu
- mettre effectivement x1 dans le lieu x2 (problème élémentaire)
- la place x2 n'est plus prise
MÉTHODE(:∄,LIEU)
- engendrer un processus t_1 sur la méthode (pour,placedisponible)
- pour chaque cubedessus de cube x1
- réveiller t_1
- réserver la placedisponible fournie par t_1
- engendrer un processus t_2 sur la méthode (pour,réservée)
- pour chaque cubedessous du cube qui est au sommet (celui-ci inclus)
- réveiller t_2
- mettre ce cube dans le lieu fourni par t_2, par la méthode (:∄,lieu,1)
- si le cube est x1 alors arrêter.
MÉTHODE(:∃,LIEU,1)
- enlever effectivement x1 de son lieu (probl. élémentaire)
- mettre effectivement x1 dans le lieu (probl. élémentaire)

METHODES




```
mth((: ,lieu),,,faire
              | : ∋ (prise,x2)
              | print('je-vais-placer:',x1)
              | x3 : ∋lieu[x1]
              | : ∌lieu ([x1],x3)
              | si succès
              | alors : ∋(lieu[x1],x2)
              |   |   print('ah-je-plaçai:',x1)
              |   |   succès
              | sinon : ∌(prise,x2)
              |   |   échec
              | fin
            fin)
```

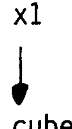
```
mmth lieu a x1 est prise , m x3 := lieu ????je-vais-placer: , call
REP1
message ( x3 x2 ) , m x3 := lieu de x2 , a x3 nestplus lieu de x2
REP2
, si succes alors a st! x1 est lieu de x2 , m x4 := ah-je-placai: ,
REP1
call message ( x4 x2 ) , a x1 nestplus prise , succes sinon
REP1
a x1 nestplus prise , echec fin forc ?

SUCCES            : 01
REP1
```

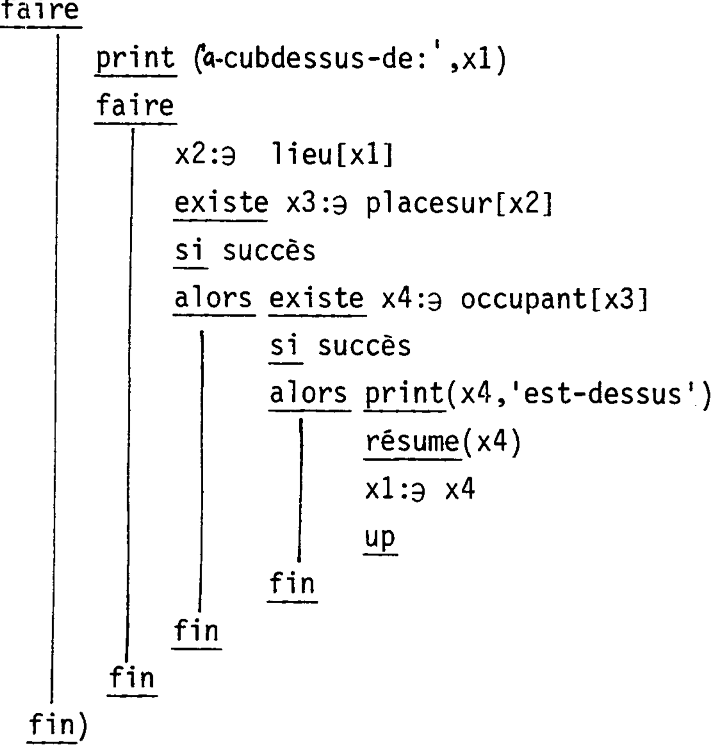
```
                                        x1      x2
                                        |       |
                                        v       v
                                       cube    place

mth((:∋,lieu),,,faire
                    print('je-vais-enlever:',x1)
                    t1 :∋ open (méthode(pour,placedisponible))
                    pour x3 :∋ cubedessus(x2)
                    |   print ('il-faut-enlever:',x3)
                    |   x4 :∋ get(t1)
                    |   :∋ réservée (x4)
                    |   si ¬ existe x8 :∋ cubesur[x3]
                    |   alors x6 :∋ x3 fin
                    fin
                    t2 :∋ open (réservée)
                    t3 :∋ open (cubedessous[x6])
                    faire
                    |   x5 :∋ x6
                    |   x7 :∋ get(t2)
                    |   :∋ lieu,1(x5 x7)
                    |   x6 :∋ get(t3)
                    |   x6 ≠ x1
                    |   up .
                    fin
          fin)
```

```
OK...
REP1
outil lieu r x4 := x1 , r x3 := je-vais-enlever: , call message
REP2
( x3 x2 ) , open t1 := placedisponible , pour x3 := cubedessus de??
REP2
de x2 r x5 := il-faut-enlever: , call message ( x5 x3 ) , get t1,? ,
REP1
a x1 est reservee , ??fin , open t2 := reservee , open t3 := cu??
REP1

cubedessous ( x3 ) , ( r x5 := x3 , get t3 , r x3 := x1 , get t2 ,
REP1
a x1 est 1-lieu de x5 , si x3 ¬= x2 alors-!0up 1 fin-) , a-std
REP1
x4 nestplus lieu de x2 , r x6 := ah-j6enleva!@@@@@@@@!enleva!: ,
REP1

call message ( x6 x2 ) f0,s0 succes fproc ?
```

```
mth((placedisponible),,,faire
                         pour x1:∋ place
                           | ∋ prise(x1) ∨ existe x2:∋  occupant[x1]
                           | si échec
                           | alors existe x3:∋ placesous[x1]
                           |   |   si succès
                           |   |   alors existe x4:∋ occupant[x3]
                           |   |     |      réservée(x3)
                           |   |     |   si succès
                           |   |     |   alors print('ah-place-s.cube:
                           |   |     |     |
                           |   |     |     |    resume(x1)
                           |   |     |   fin
                           |   |   sinon print('ah-place-table:',x1)
                           |   |     |   resume(x1)
                           |   |   fin
                           |   fin
                           |
                         fin
                      fin)
```

```
proc placedisponible pour x1 := place si x1 = prise
REP2
ou existe x2 := occupant de x1 alors up 1 sinon si
REP1
existe x3 := placesous de x1 alors si existe x4 :=
REP1
occupant de x3 ou x3 = reservee alors m x5 := ah-place-s.cube: ,
REP1
call message ( x5 x1 ) , resume ( x1 ) fin sinon m x5 := ah-place-table: ,
REP1
call message ( x5 x1 ) , resume ( x1 ) fin fin fin fproc ?
REP1
```



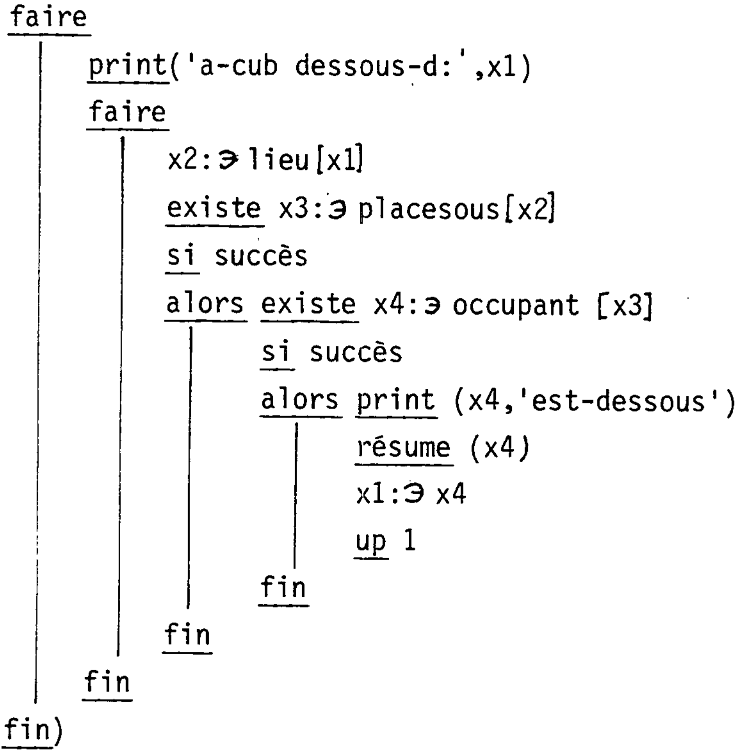


```
mth((pour,cubedessus),,,
                        faire
                            print ('a-cubdessus-de:',x1)
                            faire
                                x2:∋  lieu[x1]
                                existe x3:∋ placesur[x2]
                                si succès
                                alors existe x4:∋ occupant[x3]
                                    si succès
                                    alors print(x4,'est-dessus')
                                          résume(x4)
                                          x1:∋ x4
                                          up
                                    fin
                                fin
                            fin
                         fin)
```

```
mth cubedessus m x3 :=.a-cubdessus-de: , call message ( x3 x1 )
REP2
, ( m x2 := lieu de x1 , si existe x3 := placesur de x2 alors
REP1
si existe x4 := occupant de x3 alors m x5 := est-dessus ,
REP1
call message ( x4 x5 ) , resume ( x4 ) , m x1 := x4 , up 1 fin
REP2
fin ) fproc ?
SUCCES          ) , : : 01
REP1
```

x1
↓
cube

mth ((pour,cubedessous),,,




```
amth cubedessous m x3 := a-cubdessous-d: , call message
REP2
( x3 x1 ) , ( m x2 := lieu de x1 , si existe x3 := placesous
REP2
de x2 alors si existe x4 := occupant de x3 alors m x5 :=
REP1
est-dessous ,  call message ( x4 x5 ) , resume ( x4 )
REP2

, m x1 := x4 , up   fin fin ) fproc ?

SUCCES          : 01
```

x1 → cube   x2 → place

```
mth((:∋,lieu,1),,,faire
                  | print('jvai-placersvoir',x1)
                  | x3:∋ lieu[x1]
                  | :∌ (lieu[x1],x3)
                  | :∋ (lieu[x1],x2)
                  | print ('ah-je-placai:',x1)
                  |
              fin)
```

```
mmth 1-lieu  m x3 := jvai-placersvoir , call message ( x3 x2 ) ,
REP1
m x3 := lieu de x2 , a std x3 nestplus lieu de x2 , a std x1
REP1
est lieu de x2 , m x3 := ah-je-placai: , call message ( x3 x2 ) ,
REP1
succes forcc ?
SUCCES              : 01
REP1
```

DEUXIÈME MANIPULATION

```
call configuration ?

P1
P3
P5
P10
P15
P20
SUCCES                : 01
REP1
pour std x1 := reservee a x1 nestplus reservee fin , pour x1 :=
REP1
prise a x1 nestplus prise fin ?
SUCCES                : 01
REP1
m x1 := c3 , m x2 := p4 , a x1P¢
```

C6
C5
C4
C1
C2
C3

```
m x1 := p4 , m x2 := c3 , a x1 est lieu de x2 ?

JE-VAIS-PLACER:
C3

JE-VAIS-ENLEVER:
C3

A-CI-DESSUS-DE:
C3

C4
EST-DESSUS
IL-FAUT-ENLEVER:
C4

[illegible]-PLACE-TABLE:
P2
```

} trouver une place pour C4

C5
EST-DESSUS
IL-FAUT-ENLEVER:
C5

AU-PLACE-S.CUBE:
P6
C6
EST-DESSUS
IL-FAUT-ENLEVER:
C6

trouver une place pour C5

trouver une place pour C6

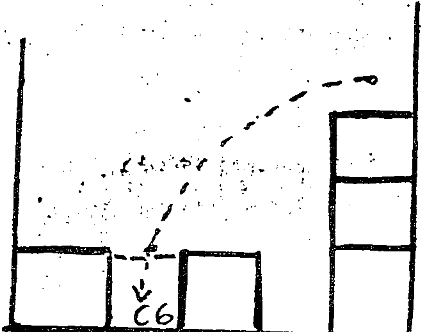
AU-PLACE-S.CUBE:
P7

A-CUBDESSOUS-D:
C6

C5
EST-DESSOUS

JVAI-PLACERSVOIR
C6

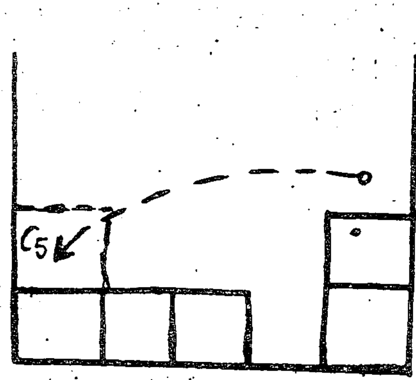
AU-JE-PLACAI:
C6



SUCCES

C4
EST-DESSOUS

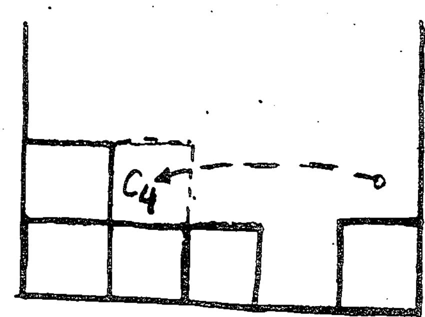
JVAI-PLACERSVOIR
C5
AU-JE-PLACAI:
C5
SUCCES



C3
EST-DESSOUS

JVAI-PLACERSVOIR
C4
AU-JE-PLACAI:
C4

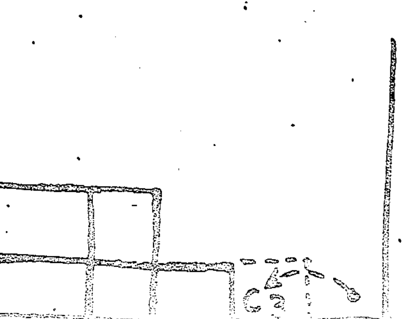
SUCCES



AU-J'ENLEVAI:
C3

SUCCES

AH-JE-PLACAI:
C3



EXEMPLE DES DESCENDANTS

Cet exemple a été présenté brièvement en page IV.6

Je définis quatre formes possibles pour la règle sémantique qui dit que "les enfants d'une personne sont les mêmes que ceux de son époux".

J'ai voulu illustrer quelques difficultés relatives à l'emploi de la déduction et de la lecture pour une même fonction d'accès.

A- Emploi de la methode(:∃, enfant)

antoine → pierre

sylvie → jean ?

```
REP1
m x1 := antoine , m x2 := pierre ,
REP1
m x3 := sylvie , m x4 := jean ,
REP1

a x2 est enfant de x1 ,
REP1

a x4 est enfant de x3 ,
REP1

 mmth enfant
REP1
a std x1 est enfant de x2 ,
REP1
si existe x3 := epoux de x2
REP2
alors a std x1 est enfant de x3
REP2
fin
REP2
```

```
 SUCCES            : 01
REP1


a x3 est epoux de x1 ,

REP1

m x4 := jacques ,   a x4 est enfant de x1 ?

 SUCCES            : 01
REP1


pour x5 := enfant de x1     scall nom ( x5 ) fin ?

PIERRE
JACQUES
 SUCCES            : 01
REP1

pour x5 := enfant de x3     scall nom ( x5 ) fin ?

JEAN
JACQUES
 SUCCES            : 01
REP1
```

antoine — epoux — sylvie

antoine — jacques — sylvie

```
rel descendant ascendant ,

REP1

 amth descendant

REP1
pour x2 := enfant de x1

REP2
resume ( x2 ) ,

REP1

pour x3 := descendant de x2

REP2
resume ( x3 )

REP2
fin fin fproc ?

 SUCCES              : 01
REP1


m x6 := marie , e x6 est

m x2 := pierre , m x6 := marie , a x6 est enfant de x2 ,  Yvonne
REP1


pour x5 := descendant de x3     scall nom ( x5 ) fin ?

JEAN
JACQUES
 SUCCES              : 01
REP1
```

```
pour x5 := descendant de x1     scall nom ( x5 ) fin ?

PIERRE
MARIE
JACQUES
 SUCCES              : 01
REP1
```

B- Emploi de la méthode (pour, enfant)

antoine — pierre

sylvie — jean

```
rel epoux epouse ,

REP1
rel enfant parent ,

REP1

m x1 := antoine , m x2 := pierre ,

REP1
 m x3 := sylvie , m x4 := jean ,

REP1

a x2 est enfant de x1 ,

REP1

a x4 est enfant de x3 ,

REP1

amth enfant

REP1

pour std x2 := enfant de x1

REP2
□resume ( x2 )

REP2

fin ,

REP1

si existe x3 := epouse de x1

REP2
alors pour std x4 := enfant de x3
REP2
```

```
fin
REP2
[illegible]

fin ,

REP1
si existe x[]3 := epoux de x1
REP2

alo[]rs pour std x4 := enfant de x3
REP2

            resume ( x4 )

REP2
        fin
REP2

fin fproc ?

 SUCCES          : 01
REP1


 rel en@nfant-1 parent-1 ,


REP1
amth enfant-1

REP1

pour x2 := ¢pour std x2 := enfant de x1

REP2
            resume x2@@( x2 )
REP2

            fin
REP2
fproc  ?

 SUCCES          : 01
```

```
a x1 est epoux de x5 ,

REP1

m x4 := jacques , a x4 est enfant de x1 ,

REP1

pour x5 := enfant de x1      scall1@ nom ( x5 ) fin ?

PIERRE
JACQUES
JEAN
  SUCCES              : 02
REP1

pour x5 := enfant-1 de x1    scall nom ( x5 ) fin ?

PIERRE
JACQUES
 SUCCES             : 01
REP1

pour x5 := enfant de x3      scall nom ( x5 ) fin ?

JEAN
PIERRE
JACQUES
 SUCCES             : 01
REP1
```

```
pour x5 := enfant-1 de x3    scall nom ( x5 ) fin ?

JEAN
 SUCCES             : 01
REP1
```

C- Emploi des deux méthodes en même temps

```
a x4 nestplus enfant de x1 ?

 SUCCES            : 01
REP1

 mmth enfant  a std x1 est enfant de x2 ,
REP1
si existe x3 := epoux de x2 alors a std x1 est enfant de x3 fin
REP2
, si existe epouse 00000¢
, si existe x3 := epouse de x2 alors a std x1 est enfant de x3 fin
REP2
fproc ?

 SUCCES            : 01
REP1

 a x4 est enfant de x1 ?

 SUCCES            : 01
REP1

 pour x5 := enfant-1 de x3    scall nom ( x5 ) fin ?

JEAN
JACQUES
 SUCCES            : 01
REP1
```

```
amth enfant-2 pour x2 := enfant-1 de x1 si x2 a0= enfant de x1

REP2
alors resume ( x2 ) fin fin fproc ?.



 SUCCES           : 01
REP1



pour x5 := enfant-2 de x1 scall nom ( x5 ) fin ?

PIERRE
JACQUES
 SUCCES           : 01
REP1


pour x5 := enfant-2 de x3 scall nom ( x5 ) fin ?

JEAN
JACQUES
 SUCCES           : 01
REP1


amth enfa
```

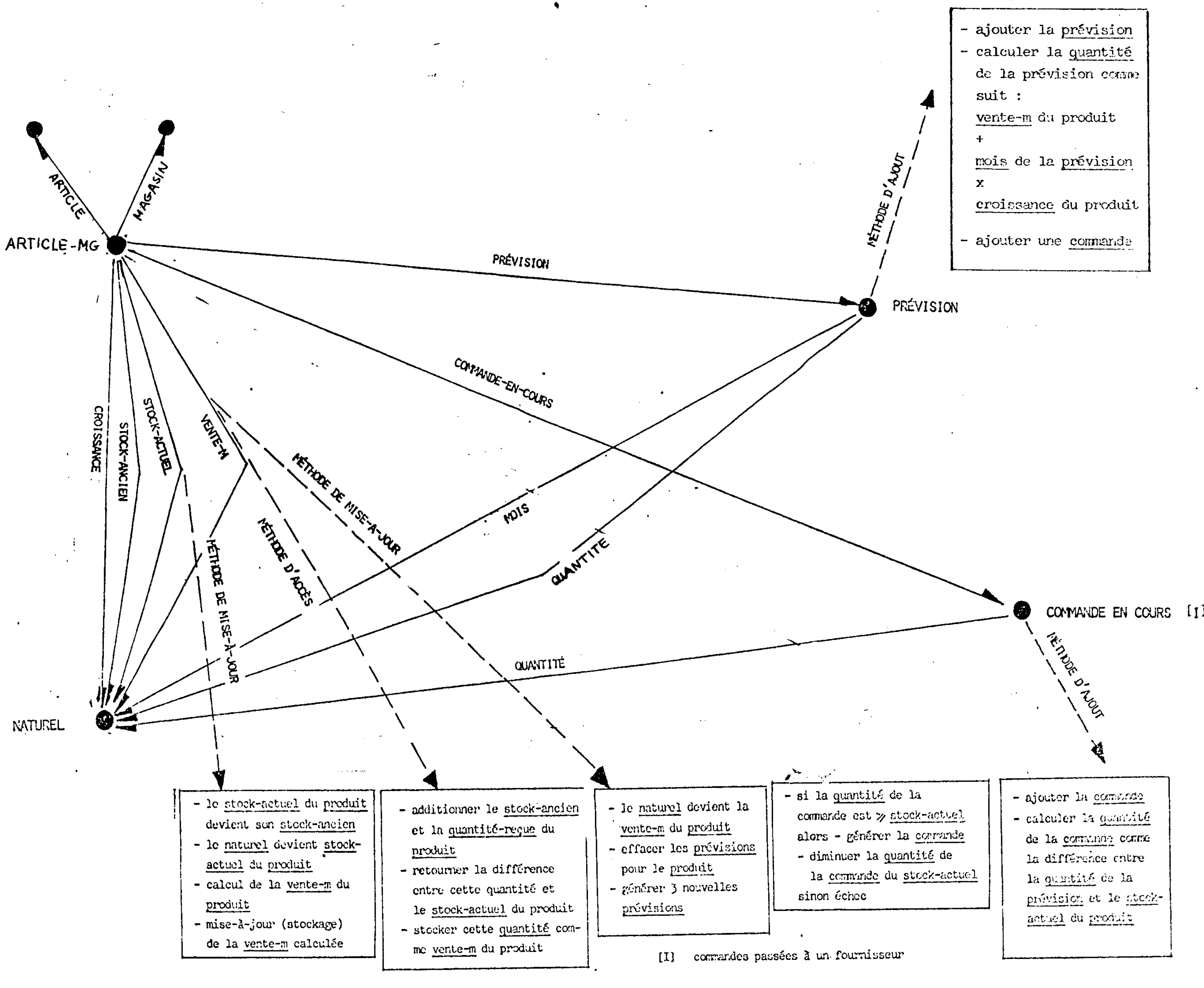
```
amth enfant-3 si existe x2 := epoux de x1 ou existe x2 := epouse
REP2
de x1 alors pour x3 := enfant-2 de x1 si x3 = enfant-2 de x3 @@2
REP2
alors resume (.x3 ) fin fin fin  fproc ?

 SUCCES            : 01
REP1


pour x5 := enfant-3 de x1 scall nom ( x5 ) fin ?
JACQUES
 SUCCES            : 01
REP1


 pour x5 := enfant-3 de x1@3 scall nom ( x5 ) fin /@?

JACQUES
 SUCCES            : 01
REP1
```

EXEMPLE DE GESTION

Dans cet exemple, que j'ai discuté à plusieurs reprises dans le texte, j'ai voulu mettre en évidence l'aspect "simulation" dont j'ai parlé en 0.3.2.

Etant donné qu'il n'y a pas d'espaces locaux en SOCRATE-P, je n'ai pas pu illustré le concept de planification que j'avais exposé. Je laisse à l'imagination du lecteur l'utilisation d'un espace local pour faire des actualisations locales en variant la quantité de la 'vente-m'.

Le lecteur peut trouver aussi un exemple d'utilisation de méthodes d'accès par déduction et lecture (vente-m).


ARTICLE
MAGASIN
ARTICLE-MG
PRÉVISION
PRÉVISION
MÉTHODE D'AJOUT
- ajouter la prévision
- calculer la quantité de la prévision comme suit :
vente-m du produit
+
mois de la prévision
x
croissance du produit
- ajouter une commande
COMMANDE-EN-COURS
COMMANDE EN COURS [I]
MÉTHODE D'AJOUT
- ajouter la commande
- calculer la quantité de la commande comme la différence entre la quantité de la prévision et le stock-actuel du produit
MOIS
QUANTITE
QUANTITÉ
CROISSANCE
STOCK-ANCIEN
STOCK-ACTUEL
VENTE-M
MÉTHODE DE MISE-À-JOUR
MÉTHODE D'ACCÈS
MÉTHODE DE MISE-À-JOUR
NATUREL
- le stock-actuel du produit devient son stock-ancien
- le naturel devient stock-actuel du produit
- calcul de la vente-m du produit
- mise-à-jour (stockage) de la vente-m calculée
- additionner le stock-ancien et la quantité-reçue du produit
- retourner la différence entre cette quantité et le stock-actuel du produit
- stocker cette quantité comme vente-m du produit
- le naturel devient la vente-m du produit
- effacer les prévisions pour le produit
- générer 3 nouvelles prévisions
- si la quantité de la commande est ≥ stock-actuel alors - générer la commande
- diminuer la quantité de la commande du stock-actuel
sinon échec
[I] commandes passées à un fournisseur

METHODES

```
mth((actualisation),,(article,magasin, naturel, naturel),
      faire
        |  existe x5 :∋ (inv[article][x1] ∩  inv[magasin][x2])
        |  si succès
        |  alors x6 :∋ stock-actuel [x5]
        |    |     :∋ stock-ancien ([x5],x6)
        |    |     :∋ qté-reçue ([x5],x3)
        |    |     :∋ stock-actuel ([x5],x4)
        |    |     x7 :∋ vente-m [x5]
        |    |     :∋ vente-m,1 ([x5],x7)
        |    |     sinon échec
        |    fin
       fin)
```

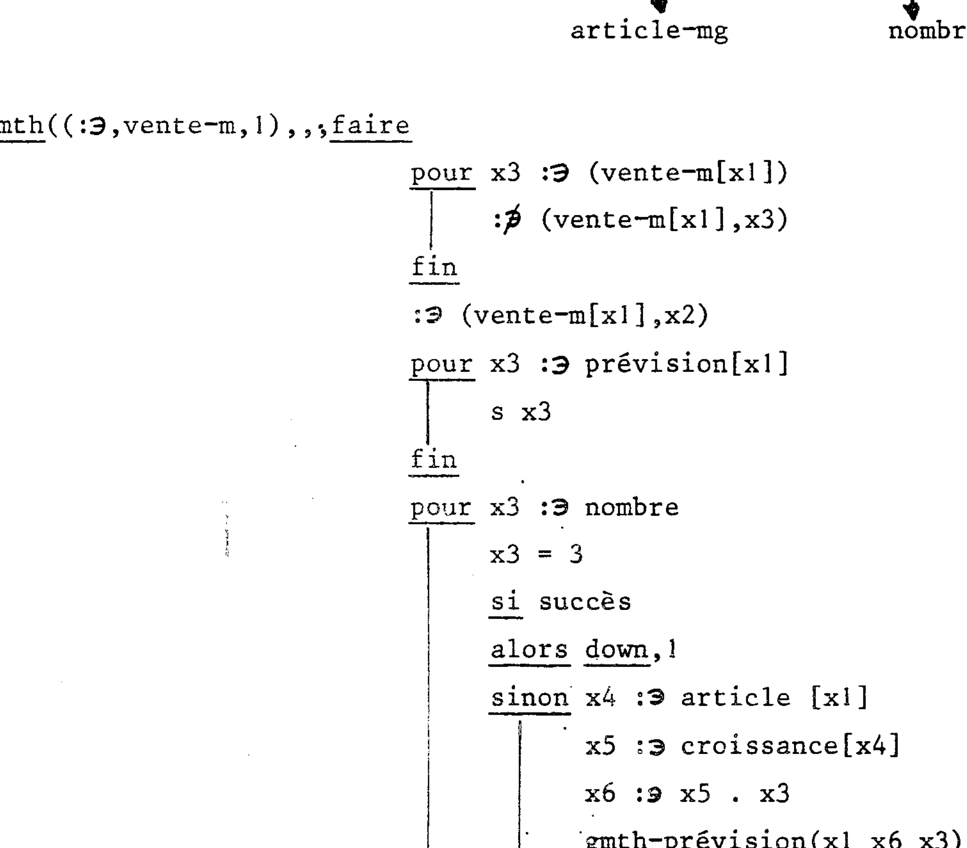
```
proc actualisation si existe x5 :=      ( article-m-a de y1 et
article-m-b de x1?2 )  alors m x6 := stock-actuel de x5 ,

a x6 est stock-ancien de x5,? , a x3 est qte-recue de x5 ,

a x4 est stock-actuel de x5 , m x7 := vente-m-? de x5 ,

a x7 est vente-m-1 de x5 , ??sinon echec fin fproc ?
SUCCE?            :?01
```

```
mth((gmth-prévision),,(article-mg,naturel, naturel),
                        faire
                            print('prévision-vente',article(x1))
                            x4 :∋ g prévision
                            :∋ prévision ([x1],x4)  :  article ([x4],[x1])
                            pour x5 :∋ (vente-m[x1])
                                x6 :∋ x2 + x5
                                :∋  quantité ([x4],x6)
                                down
                            fin
                            :∋ mois ([x4],x3)
                            x5 :∋ stock-actuel[x1]
                            gmth-commande(x1,x6-x5)
                        fin)
```

```
proc gmth-prevision m x7 := prevision-vente , m x8 :=
article de x1 , call message ( x7 x8 ) , g prevision x4 ,
a x1 est article-m-f de x4 , pour std x5 := vente-m de
x1 m x7 := x1 , scall somme ( x2 x5 ) , a x1 est
quantite de x4 , down 1 fin , a x3 est mois de x4 ,
m x5 := so@tock-actuel de x7 , scall difference
( x1 x5 ) , call gmth-commande ( x7 x1 ) fproc ?

SUCCES              : 01
```

```
mth((gmth-commande),,(article-mg, naturel)faire,
                              | print('lancer-commande:',article(x1))
                              | x3 :∋ g(commande-en-crs)
                              | :∋ commande-en-crs ([x1], x3)
                              | :∋ quantité ([x3],x2)
                             fin)
```

```
proc mth-commande m x7 := lancer-comma@nde: ,
m x8 := article de x1 , call message ( x7 x8 ) ,
            g std commande-en-crs x3 , a x1 est
article-mg@-g de x3 , a [illegible] x2 est quantite-b de x3
, ¢fproc ?
SUCCES          : 01
```

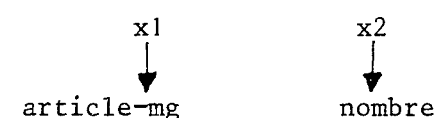


```
                                        x1                x2
                                        ↓                 ↓
                                   article-mg          nombre

mth((:∋,vente-m,1),,,faire
                    pour x3 :∋ (vente-m[x1])
                    |     :∌ (vente-m[x1],x3)
                    fin
                    :∋ (vente-m[x1],x2)
                    pour x3 :∋ prévision[x1]
                    |     s x3
                    fin
                    pour x3 :∋ nombre
                    |     x3 = 3
                    |     si succès
                    |     alors down,1
                    |     sinon x4 :∋ article [x1]
                    |     |     x5 :∋ croissance[x4]
                    |     |     x6 :∋ x5 . x3
                    |     |     gmth-prévision(x1 x6 x3)
                    |     fin
                    fin
                fin)
```

```
mth vente-m-1 pour std x3 := vente-m de x2 a std x3 nestplus
vente-m de x2 , ??fin , a std x1 est vente-m de x2 , pour x3
:= provision de x2 t x3 fin , m?pour x3 := nombre si x3 = 3
alors down 1 sinon m x4 := article de x2 , m x5 := croissance
de x4 , m x6 := x1 , ccall produit ( x5 x3 ) , call mth-provision
( x2 x1 x3 ) fin fin fonct?
```

```
                                      x1
                                      |
                                      v
                                 article-mg

mth((pour,vente-m),,,faire
                    | print('calcul-vente:',article[x1] )
                    | x2 :∋ stock-ancien[x1]
                    | x3 :∋ qté-reçue[x1]
                    | x4 :∋ x2 + x3
                    | x5 :∋ stock-actuel[x1]
                    | x6 :∋ x4 - x5
                    | print ('vente',x6)
                    | return(x6)
                   fin)
```

```
amth vente-m m x7 ;0:= calcul-vente: , m x8 := article de x1 ,
call message ( x7 x8 ) , m x2 := stock-ancien de x1 , m x3 :=
qte-recue de x1 , m x4 := x1 , call 00000scall somme (x0 x2 x3 )
!
, m x5 := stock-actuel de x4 , scall difference ( x1 x5 ) ,
m x7 := vente , call message ( x7 x1 ) , return ( x1 ) fproc ?
SUCCES              : 01
```




```
mth((:∋,stock-actuel),,,faire
                        x3 :∋ stock-actuel [x1]
                        :∌  (stock-actuel[x1],x3)
                        :∋  (stock-actuel[x1],x2)
                     fin)
```

```
                                  x1              x2
                                  ↓               ↓
                             article-mg         nombre
```

```
mth((:  ,qte-reçue),,,faire
                        x3 :∋ qté-reçue[x1]
                        :∌  (qté-reçue[x1],x3)
                        :∋  (qté-reçue[x1],x2)
                   fin)
```

```
mth stock-actuel n x3 := stock-actuel de x2 , a std x3
nestplus stock-actuel de x2 , a std x1 est stock-actuel de
x2 foroc ?
```

```
mth qte-recue n x3 := qte-recue de x2 , a std x3 nestplus
qte-recue de x2 , a std x1 est qte-recue de x2 foroc ?
SUCCES                  : 01
```

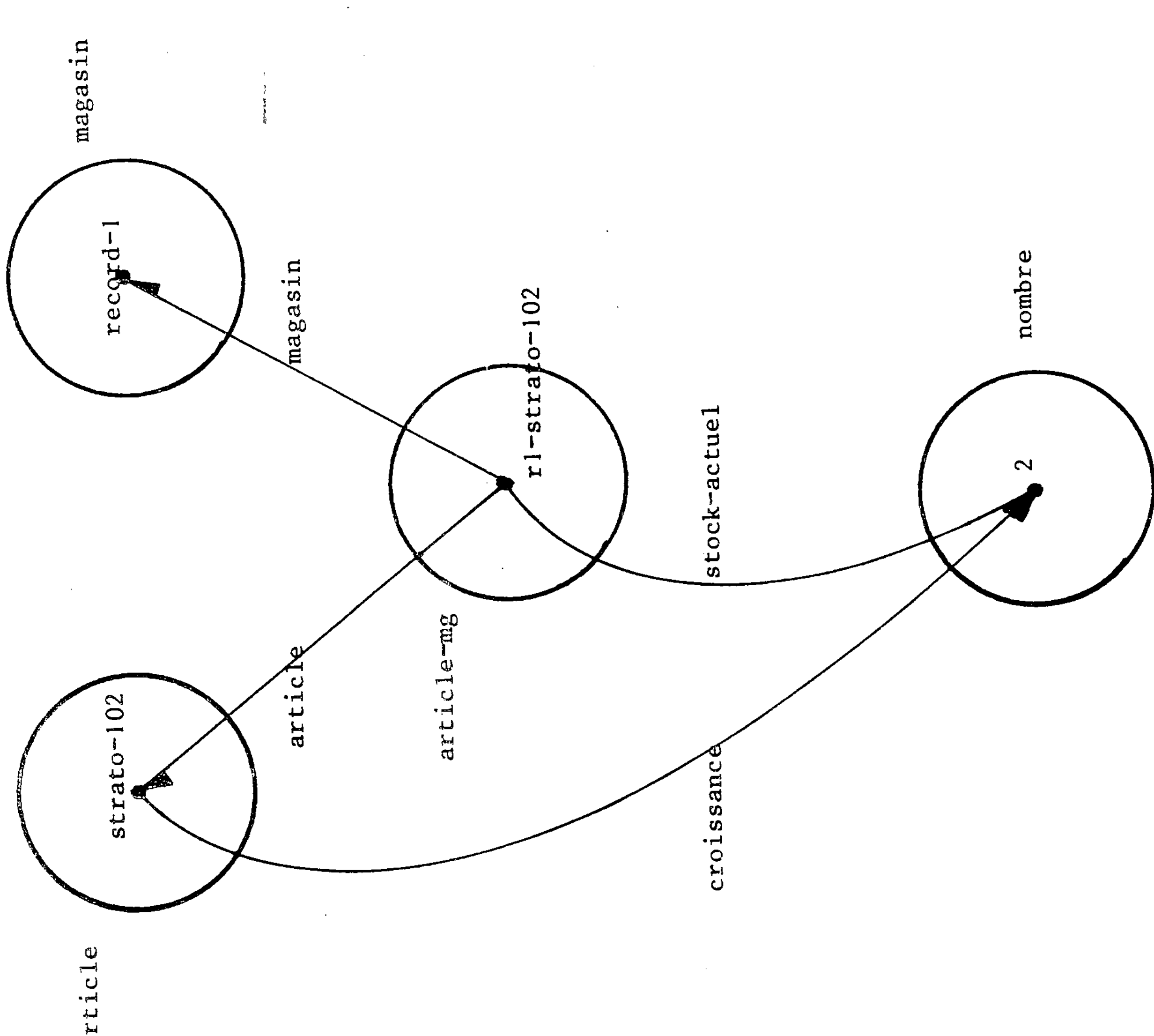
	x1	x2
	article-mg	nombre

```
mth((:∍,stock-ancien),,,faire
                        | x3 :  stock-ancien(x1)
                        | :∌  stock-ancien ([x1],x3)
                        | :∍  (stock-ancien[x1],x2)
                        fin)
```

mth stock-ancien m x3 := stock-ancien de x2 , a x3 nestplus stock-ancien de x2 , a std x1 est stock-ancien de x2 foroc ?

SUCCES : 01

INITIALISATION



```
m x1 := strato-102 , m x2 := record-1 , m x3 := 4 ,
m x4 := 3 , call actualisation ( x1 x2 x3 x4 ) ?
```

← ENTRÉE : niveau stock = 3
quantité reçue depuis la dernière commande

```
!CALCUL-VENTE:
STRATO-102
VENTE
3
```

← le calcul de la vente du mois m écoulé est lancé automatiquement

```
!PREVISION-VENTE
STRATO-102
```

← le calcul de la prevision pour le mois m+1 est lancé (et suivi de stockage de la valeur)

```
LANCER-COMMANDE:
STRATO-102
```

← le calcul de la commande pour le mois m+1 est lancé (et suivi de stockage de la valeur)

```
!PREVISION-VENTE
STRATO-102
LANCER-COMMANDE:
STRATO-102
```

← le calcul de la prevision pour le mois m+2 est lancé (et suivi de stockage de la valeur)

← le calcul de la commande pour le mois m+2 est lancé (et suivi de stockage de la valeur)

```
SUCCES          : 01

m x1 := r1-strato-102 , pour x2 := prevision de x1 m x3
:= mois de x2 , m x4 := quantite de x2 , scall nom ( x3 )
, scall nom ( x4 ) fin ?
```

RESULTATS :

```
11
5
```

← prevision calculée pour le mois m+1

← prevision calculée pour le mois m+2

```
2
7
SUCCES          : 01

pour x2 := commande-en-crs de x1 m x3 := quantite-b de x2
, scall nom (x0 x3 ) fin ?

2
4
SUCCES          : 01
```

← commande calculée pour le mois m+1

← commande calculée pour le mois m+2

B I B L I O G R A P H I E

(1) FAVRE J.C. : "APPLICATION SOCRATE-P"
Note technique SOCRATE n° 23, Laboratoire d'Informatique de Grenoble l'Université Scientifique et Médicale de Grenoble

J'ai présenté SOCRATE-P dans la partie 4.

L'interpréteur SOCRATE-P est écrit en langage SOCRATE.

Pour son utilisation, on peut s'adresser au Centre Interuniversitaire de Calcul de Grenoble (Madame A. STIERS - Bâtiment B) ou à J.C. FAVRE (I.U.T. d'informatique de Grenoble)

(2) ABRIAL J.R. : "DATA SEMANTICS"
Cours donné à l'Ecole d'Hiver de la CEE à l'Alpe d'Huez, 1973

Dans ce cours, le Lecteur trouvera entre autres, une définition plus générale du mécanisme des "espaces" ainsi qu'un traitement des problèmes relatifs au partage des données.

BASES DE DONNEES

(3) PROJET SOCRATE : "SPECIFICATIONS GENERALES"
Laboratoire d'Informatique de l'Université Scientifique et Médicale de Grenoble, 1970

(4) ABRIAL J.R., J.BAS, J.P. CAHEN, J.C. FAVRE, G. MAZARE, R. MORIN : "PROJET-SOCRATE : Données, Mémoires et Utilisateurs d'une Banque de Données"
Journées AFCET de Banques de Données, Aix en Provence, 1971

(5) FAVRE J.C. : "LANGAGE DE CITATIONS - REQUETES"
Note technique Socrate n° 8, Laboratoire d'Informatique de l'Université Scientifique et Médicale de Grenoble, 1972

(6) CODD E.F. : "A Relational Model of Data for Large Shared Data Banks"
Communications ACM, 13, 6, juin 1970, pp 377-387

(7) CODD E.F. : "Further Normalization of the Data Base Relational Model"
IBM Research, août 1971, RJ 909

(8) CODD E.F. : "A Data Base Sublanguage Founded on the Relational Calculus"
Communications ACM, HQ, 1972

(9) CODD E.F. : "Relational Completeness of Data Base Sublanguages"
IBM Research, Mars 1972

(10) STRNAD A.J. : "The Relational Approach to the Management of Data Bases"
Compte-rendu du Congrès IFIP 1971, Ljubljana, août 1971

(11) SYNTEX : Manuel, Centre d'Etudes et de Recherches de Toulouse

(12) SENKO, ALTMAN et al. :"Data Structures and Accessing in Data Base Systems"
IBM Systems Journal, vol. 2, n° 1, 1973

(13) CODASYL DEVELOPMENT COMMITEE : "An Information Algebra"
Phase I, Report, Communications ACM, 5, avril 1962, pp 190-204

(14) IMS (Information Management System)
Manuel IBM

(15) LEVEIN R.E., MARON M.E. : "A Computer System for Inference Execution and Data Retrieval"
Communications ACM, 10, 11, 1967, pp 715-721

(16) CRICK M.F., LORIE R.A. et al : "A Data Base System for Interactive Applications
IBM Data Processing Division, Cambridge Scientific Center

(17) FELDMAN J., ROVNER P. : "An Algol-based Associative Language"(LEAP)
Communications ACM, vol 2, n° 8, août 1969

(18) ASH [illegible], SIBLEY E.H. : "TRAMP : An Interpretive Associative Processor with
Deductive Capabilities"
Université de Michigan, [illegible]

(19) PORTAL D. : "Conception d'un Système Automatique de Gestion de la Scolarité de l'Enseignement Supérieur"
Thèse de Docteur-Ingénieur, Université Scientifique et Médicale de Grenoble, 1975

(20) PECCOUD F. : "MACSI : Un Système d'Aide à la Conception de Systèmes d'Informations"
Thèse d'Etat, Laboratoire d'Informatique, Université Scientifique et Médicale de Grenoble, 1975

(21) BARBOSA R. : "Modèles Relationnels de Structure"
Note technique Socrate, n°11, 1972

(22) BARBOSA R. : "Etapes de Définition d'une Application"
Note technique Socrate, n°12, 1973

PROGRAMMATION ET THEORIE DE LA PROGRAMMATION

(23) VAN WIJNGAARDEN A. et al. : "Report on the Algorithmic Language Algol 68"
Mathematisch Centrum Amsterdam, MR 101

(24) DAHL O.J., DIJKSTRA E.W., HOARE C.A.R. : "Structured Programming"
Academic Press, 1972

(25) WOODGER : "Semantic Levels of Programming"
Compte-rendu du Congrès IFIP 1971, Ljubljana, août 1971

(26) CORAY G., HATCHER W.S. : "A Logical Framework for Large File Information Handling"
Ecole Polytechnique de Lausanne, Departement de Mathématiques.

(27) WULF W.A. : "Programming Without the Goto"
Compte-rendu du Congrès IFIP 1971, Ljubjana, août 1971

(28) BOCHMANN G.V. : "Multiple Exits from a Loop Without the Goto"
Faculté des Arts et Sciences, Université de Montréal

(29) GOLOMB S., BAMERT L.D. : " Backtrack Programming"
JACM, vol 12, n° 4, oct. 1965, pp 516-524

(30) BALZER R. : "Dataless Programming"
FJCC, 1967

(31) COURTIN J., VOIRON J. : "Introduction à l'Algorithmique et aux Structures de Données"
Cours polycopié 1974-75, IUT d'Informatique, Grenoble

(32) CHIARAMELLA Y. : "COBOL : Présentation d'un sous-ensemble du langage. Eléments de programmation structurée".
Cours polycopié 1975, IUT d'Informatique, Grenoble

INTELLIGENCE ARTIFICIELLE

(33) AMAREL S. : "An Approach to Heuristic Problem-Solving and Theorem-proving in the Propositional Calculus"
Université de Toronto, Toronto Press

(34) WINOGRAD T. : "Procedures as a Representation for Data in a Computer Program for Understanding Natural Language"
Thèse Ph. D., MIT, février 1971

(35) HEWITT C. : "Description and Theoretical Analysis (using schemmata) of PLANNER : A Language for proving Theorems and Manipulating Models with Robots"
Thèse Ph. D., MIT, février 1971

(36) CORAY G. : "Intelligence Artificielle" (Première partie)
Cours dispensé à l'Ecole d'Eté de Grenade

(37) SIMON H.A. : "The Theory of Problem-Solving"
Compte-rendu du Congrès IFIP 1971, Ljubljana, août 1971

(38) RAPHAEL B. : "Introduction to the Calculs of Knowledge"
Memo 29, novembre 1971, A.I. Project Computation Center MIT

(39) BOBROW D., RAPHAEL B. : "New Programming Languages for Artificial Intelligence Research"
Computing Surveys, vol 6, n° 3, septembre 1974

(40) Mac DERMOTT V., SUSSMAN G.J. : "The CONNIVER Reference Manual"
Memo 259, A.I. Laboratory MIT, 1972

(41) RULIFSON J.F., WALDINGER R.J., DERKSEN J.A. : "A Language for Writing Problem-Solving Programs" (QA4)
Compte-rendu du Congrès IFIP 1971, Ljubljana, août 1971

(42) BUCHANAM B.G., LEDERBERG J. : "The Heuristic Dendral Program for Explaining Empirical Data"
Compte-rendu du Congrès IFIP 1971, Ljubljana, août 1971

(43) FIKES R.E. : "Monitored Execution of Robot Plans Produced by STRIPS"
Compte-rendu du Congrès IFIP 1971, Ljubljana, août 1971

(44) MINSKY M. : "Matter, Mind and Models"
Memo 77, mars 1965, projet MAC, MIT

(45) NILSSON N.J. : "Problem-Solving Methods in Artificial Intelligence"
Mc Graw Hill Company

(46) Mac CARTHY J., HAYES P.J. : "Some Philosophical Problems from the Standpoint of Artificial Intelligence"
Memo 77, mars 1965, Projet MAC, MIT

GESTION et INFORMATIQUE DE GESTION

(47) MELES J. : "La Gestion par les Systèmes - Essai de Praxéologie"
Ed. Hommes et Techniques

(48) KLEIN M., LEVY J.P., LIMOUSIN P. : "Système Scarabé"
Manuel Utilisateur, octobre 1972

(49) ANDERSIN H. : "Concepts, Techniques and Models for the Process of Management"
Compte-rendu du Congrès IFIP 1971, Ljubljana, août 1971

(50) BARBOSA R. : "Réalisation Rapide d'un prototype de système d'aide à la décision : SOCRATE-S"
Note technique interne n°2 - Laboratoire d'Informatique de l'Université Scientifique et Médicale de Grenoble

---

(51) CARNAP : "Introduction to Symbolic Logic and its Applications"
Dover Publications Inc. New York

(52) ROGERS H. : "Recursive function Theory"

(53) FEYS R. : "Modal Logics"
Monographies réunies par J. DOPP (Louvain)
Editions Gauthier)Villars, Paris 1965