N° d'ordre : 16

Année 1988

THÈSE

présentée en vue de
l'obtention du titre de

DOCTEUR

de

L'ECOLE NATIONALE SUPERIEURE
DE L'AERONAUTIQUE ET DE L'ESPACE

SPECIALITE : INFORMATIQUE

par

Michel BOSCO

CONTRIBUTION A LA SPECIFICATION ET A LA CONCEPTION DE SYSTEMES D'INFORMATION INTELLIGENTS POUR LE GENIE LOGICIEL

Soutenue le 23 juin 1988 devant la Commission d'Examen :

MM. S. MIRANDA	**Président**
L. BOI **J.L. DURIEUX** **M. GRIFFITHS** **B. LECUSSAN** **I. THOMAS**	**Examinateurs**

Je tiens à remercier sincèrement ici

Monsieur ***Boi****, Ingénieur Chef de Service au CETE Méditerranée, qui a dirigé et encouragé mes travaux,*

Monsieur ***Miranda****, Professeur à l'Université de Nice, qui m'a fait l'honneur de présider le Jury de Thèse,*

Monsieur ***Lécussan****, Professeur à l'ENSAE, rapporteur de cette thèse,*

Monsieur ***Durieux****, Professeur à l'Université de Toulouse,*
Monsieur ***Griffiths****, Professeur à l'Université d'Aix-Marseille,*
Monsieur ***Thomas****, Professeur à l'Université de Hall (GB) et chef de groupe de projets au GIE Emeraude,*
qui ont bien voulu s'intéresser à ce travail,

Messieurs ***Masson*** *et* ***Garin Sapin****, pour le soutien de la direction du CETE et de son Département Informatique,*

mes amis et collègues du Projet ESPRIT P938, financé partiellement par la CEE, ainsi que ceux du Projet CONCERTO, financé par le CNET,

Michel ***Gibelli****, qui, par ses remarques, ses travaux, et par la richesse des discussions dans lesquelles nous nous sommes "embarqués" ensemble, a beaucoup apporté aux réalisations présentées dans cette thèse.*

Moins sérieusement, enfin, merci à Monsieur MacIntosh et à Madame Sun3/50.

à Nathalie

à mes parents

Résumé

Les logiciels sont fréquemment, aujourd'hui, des ensembles d'outils, des "ateliers" dont l'aspect intégré est un point fort: Cette intégration, est, le plus souvent, obtenue par l'utilisation d'un Système d'Information qui supporte les informations partagées par les outils.

Nos travaux ont consisté à examiner quelles étaient aujourd'hui les lacunes des logiciels proposés pour construire des Systèmes d'Information et de définir certains concepts qu'il fallait leur ajouter. Nous avons aussi réalisé un prototype qui les implémente, en nous appuyant sur un système de gestion de bases de données relationnelles, un langage logique (Prolog), et un système de gestion de fichiers.

Ce travail a pris le nom d'Impish, et s'inscrit dans le cadre du projet ESPRIT P938, "Integrated Management Process Workbench", dont l'objectif est de réaliser un atelier de gestion de projets logiciels.

---- Table des matières -----

---- Plan

- Plan détaillé des parties I, II, III

---- Table des figures

0. Introduction

0.1 Définitions

0.1.1 Les Systèmes d'Information

La notion de système d'information (SI) est désormais un concept largement défini et commenté. Dans le cadre de cette étude, nous nous réfèrerons aux définitions d'un tel système par J.L. Le Moigne [LEMO73], reprises par d'autres [TARD79] :

Ses fonctions, en résumé, sont :

- de collecter l'information,
- de mémoriser la connaissance en l'organisant,
- d'en assurer la communication,
- d'en traiter automatiquement une partie pour alléger la tâche du système de décision (ou de pilotage).

On trouvera plus loin une définition de ce que l'on entend par connaissance. Nous dirons ici que, pour nous, une information est une connaissance **nouvelle**, qui vient s'ajouter à la représentation que l'on a d'un monde : un système d'information est un moyen de partager des connaissances par communication des informations.

Si l'on considère une organisation dont l'objectif est de réaliser un produit, le système d'information de cette organisation peut donc être défini comme le centre de gestion des informations que peuvent échanger système opérant (SO) (le système qui opère la transformation de la matière "brute" en produit) et système de pilotage (SP) (le système qui conçoit et met en oeuvre les procédures organisant la production).

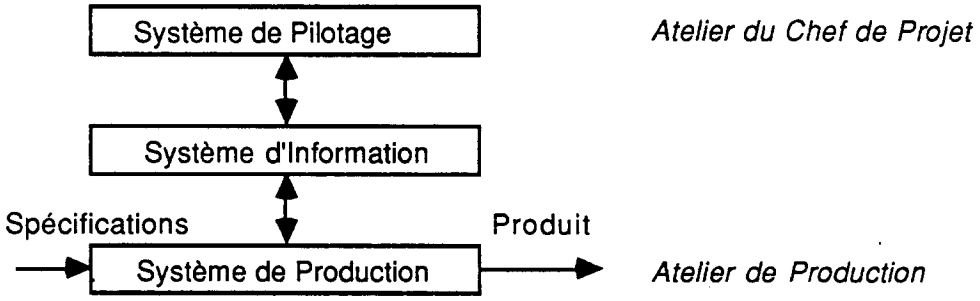


fig. 1 : SI - SO - SP

Une telle approche (dite "approche systémique des organisations") fixe certaines règles :

Notamment, toute information doit transiter par le SI sans jamais circuler directement du SP vers le SO, ou du SO vers le SP, et ce afin de dédier au SI un rôle de vérification de l'intégrité des informations échangées.

D'abord destinés à gérer des données "statiques", les SI peuvent logiquement être étendus à la gestion de l'ensemble des informations partagées par SO et SP, ou du moins d'un plus grand nombre, par exemple de certaines règles générales, de documents, …

Le point commun de ces informations est leur partage possible par un ensemble de plusieurs utilisateurs (du SP et SO).

0.1.2 La structure d'accueil (SA) d'un SI

Les caractéristiques d'une structure d'accueil, d'un logiciel destiné à supporter un système d'information doivent

et à ceux qui en découlent :

Tout d'abord, la SA doit permettre de **définir** et de **stocker** différents concepts et types de connaissance, types que l'on s'attachera à préciser dans un prochain chapitre.

A ces propriétés statiques sont liées les possibilités d'**ajouter** une nouvelle connaissance, de **supprimer** ou d'en **modifier** une existante.

Une autre fonctionnalité, essentielle elle aussi, est de permettre la **recherche** des informations par des moyens aussi riches que possible, pour assurer à son utilisateur une "**navigation**" complète et aisée.

Ensuite, la position centrale du SI impose à la SA de pouvoir gérer les **accès concurrents** aux informations, donner aux différents utilisateurs la possibilité de considérer les informations selon leur propre point de **vue**, et être suffisamment **flexible** pour permettre, lors de la mise en place d'un SI, une grande capacité à supporter son évolution.

La plupart des travaux actuels ont conduit à implémenter les SI sur des systèmes de gestion de bases de données. L'évolution des techniques de représentation des connaissances, la volonté de regrouper au sein du SI l'ensemble des connaissances partagées par SO et SP pousse à l'étude de systèmes plus complexes. Les travaux présentés ci après se veulent d'être une contribution dans cette voie.

0.2 Présentation des travaux et études préalables à la rédaction de cette thèse

0.2.1 Une approche "synthétisante" et globale

De nombreux travaux sont ou ont été menés, qui traitent des structures de représentation et de support des connaissances, ou bien encore de leur spécification.

Il nous a paru fort intéressant de chercher dans quelle mesure il était possible d'intégrer le plus grand nombre de solutions aux divers problèmes posés, et ce au sein d'un seul et même système.

Ainsi, pour étudier quelle pourrait être une structure d'accueil adéquate à l'implémentation d'un système d'information capable de gérer des données simples, mais aussi des connaissances complexes, il nous a paru avantageux de nous situer dans le cadre d'un projet visant à implémenter un tel SI, et, mieux, un ensemble d'outils (du SP ou du SO) ayant à traiter ces informations : c'était l'assurance de pouvoir considérer le problème avec une **vision globale** et **synthétisante**.

0.2.2 Les projets hôtes

Dans l'esprit de notre approche, les ateliers de gestion de projets logiciels constituent un fond de connaissance particulièrement attrayant :

- ils partagent des informations nombreuses et variées, des documents, formulaires, règles, données factuelles, hypothèses de raisonnement,…

- ils sont d'autre part constitués d'outils de calcul mais aussi d'aide à la décision qui viennent traiter ces informations.

Leurs systèmes d'informations constituent donc à nos yeux l'une des classes de SI les plus riches, par la diversité des types d'informations gérées et par celle des types de traitements qui leur sont appliqués.

Nous avons donc participé à plusieurs projets visant à réaliser de tels outils :

Dans un premier temps, c'est le projet Concerto [ANDR84] (CNET) qui nous a permis, non seulement la familiarisation avec les concepts de SI, mais aussi de premiers essais de spécifications d'une SA "idéale" (?) [BOSC85].

Par la suite, l'un des projets du programme européen ESPRIT, "Integrated Management Process Workbench" (IMP Workbench), a fourni l'essentiel de la matière qui nous à conduit jusqu'à l'implémentation d'un logiciel SA et d'un SI destiné à un ensemble d'outils de gestion de projets logiciels, travaux présentés dans la suite de ce document.

Leurs besoins importants de manipulations et de recherches d'informations en font donc de bons candidats pour mener à bien notre étude.

Une bibliographie importante (Concerto en France (CNET), PMDB aux Etats Unis (TRW) [PENA84], IMP Workench [JENK87], PIMS, SPMMS [HURS86], PCTE [THOM87] en Europe (ESPRIT), pour ne citer qu'eux), et notre participation à deux d'entre eux, seront une source appréciable pour l'analyse des besoins et la conception d'une SA, selon l'optique décrite

plus haut.

0.2.3 Présentation des travaux poursuivis

Les "fils conducteurs" de nos travaux sont :

- la volonté de synthèse, dans le but de réaliser l'intégration, au sein d'un système homogène et aussi riche que possible, de concepts divers utiles à l'élaboration de systèmes d'informations étendus et intelligents.

- la notion de cycle de vie des Systèmes d'Information : Notre objectif final est de proposer des moyens de représentation de la connaissance, des concepts pour la modélisation des informations, mais aussi d'implémenter un langage et un système logiciel pour leur définition et leur manipulation, qui soient basés sur les concepts précédemment formalisés. Nous restreindrons nos travaux au domaine du Génie Logiciel.

Notre démarche comportera les étapes suivantes :

Tout d'abord, différentes études :

- Une typologie et une analyse de ce que l'on peut appeler connaissances, ou informations, …, rattachées à l'observation du monde réel,

- Un état de l'art des techniques de représentation des connaissances,

- Le choix de celles qui nous paraîtrons les plus adéquates à la construction d'une SA adaptée à la réalisation d'un SI pour ateliers de Génie Logiciel. Ce choix a pour

objectif de restreindre l'éventail des possibilités, et va permettre :

- Une étude des techniques d'implémentation qui en découlent,

- Un bref état de l'art des méthodes de spécification des connaissances qui peuvent leur être associées,

Ensuite, des réalisations :

- La définition des concepts attachés aux informations que nous souhaitons voir gérer par notre système, des possibilités de manipulations qu'il se doit de fournir à ses utilisateurs, et de ses capacités à en réaliser lui même de manière autonome et automatique,

- Une réflexion sur des techniques d'acquisition et de spécification des informations relatives du monde réel que l'on veut modéliser avec les concepts choisis,

- Une architecture logicielle permettant d'implémenter ces concepts,

- L'implémentation d'un langage permettant la définition et la manipulation de ces concepts,

L'évaluation de ces concepts, de cette structure d'accueil et de son langage d'accès sera assurée par l'implémentation d'un système d'information (celui d'IMP Workbench) et l'intégration d'outils de gestion de projets (qui travailleront sur ce SI).

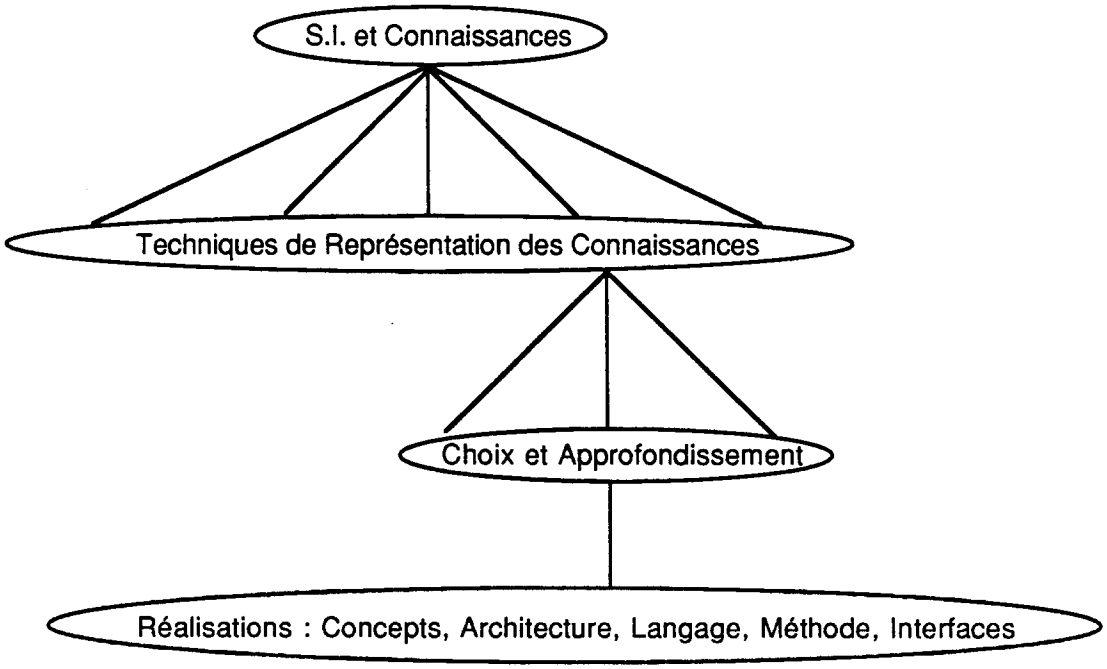


fig. 2 : Notre démarche

1.

Typologie des Connaissances

1.1 Nature des connaissances et informations

Notre objectif, ici, est d'analyser l'"univers d'un discours", de mettre en évidence les divers types d'éléments qui le composent, et, pour cela, de donner une "représentation symbolique de certains de ses différents aspects" (C'est ce que Frost [FROS86] définit comme étant l'"expression d'une connaissance") qui soit gérable par un système automatisé.

L'univers d'un discours porte sur un certain nombre d'entités physiques, composées, interréliées, qualifiées.

La connaissance sera l'expression d'un certain nombre de ces éléments dans un langage donné, et supporté par un vecteur de communication.

Nous essayons, dans ce qui suit, de préciser ces notions.

1.1.1 Entités et faits

1.1.1.1 Le concept d'entité atomique :

On pourra dire que "la chaussure de numéro de série unique 1234565" constitue une entité atomique si l'on considère toujours, dans l'univers du discours considéré (c'est à dire toutes les fois qu'un utilisateur manipule ou observe sa représentation symbolique), l'objet physique "chaussure numéro 1234565", comme un tout indissociable (et que jamais, par exemple, on ne sépare la semelle ou le lacet pour les manipuler indépendamment).

Cela sous entend que l'on est capable de parfaitement

absolument, sans se référer à sa position relative par rapport à d'autres éléments de l'univers considéré.

Toutefois, une entité atomique n'est pas pour autant "immuable" : teindre "la" chaussure précitée en bleu, modifiera l'objet (l'une de ses qualités) mais n'entamme en rien son intégrité (cela si l'on ne s'attache pas à considérer l'objet (?) teinture).

Cette définition revient à considérer notre objet comme une instance d'entité et de ses valeurs d'attributs, si l'on se réfère au "modèle EAR" [CHEN76].

1.1.1.2 L'aspect structuré des entités

Les objets décrits ci avant subissent naturellement des manipulations, qui, pour certaines, se rapportent à des entités isolées, mais qui, pour d'autres, se rapportent à des compositions structurées d'entités :

Appelons "Titi" notre chaussure de numéro de série 1234565 … (on peut être familier avec ses vêtements, et cela change des canaris …) :

"Brûler Titi"

est une action qui, dans un univers de discours où l'on ne s'intéresse pas à la transformation de la matière, consiste à faire disparaitre l'entité isolée "Titi".

Si l'on considère maintenant un "groupe momentanément solidaire" de trois entités, Monsieur Grandblond, Titi-G, et Titi-D, les deux chaussures qu'il a aux pieds, et que l'on s'intéresse aux conséquences que pourrait avoir la chute de ce groupe dans une piscine, il apparait la double nécessité :

- de pouvoir désigner un **aggrégat** d'objets physiques par un seul symbole,

- de pouvoir manipuler cet aggrégat par la simple manipulation de ce symbole.

1.1.1.3 L'aspect générique dans un univers de discours

Il est utile de constater que plusieurs manipulations seront communes à toutes les chaussures :

"chausser","déchausser", …

D'où l'attrait de pouvoir considérer et désigner l'**abstraction**, le concept de chaussure, sans pour autant en nommer une précisément.

De plus, une manipulation peut concerner un ensemble d'entités :

"je range les chaussures, les sabots et les charentaises de Pôpa et Môman dans le coffre sous l'escalier"

Il est alors pratique de pouvoir désigner "les chaussures, les sabots et les charentaises de Pôpa et Môman" comme une **union** d'entités.

1.1.1.4 L'observation des entités, le concept de "fait"

Par "**fait**", dans notre propos, nous désignerons :

- toute connaissance déductible de l'observation d'une entité atomique comme défini ci-dessus :

"la couleur de la chaussure de numéro de série

12234226 est rouge à l'instant t0"

"la couleur de la chaussure de numéro de série 12234226 est bleue à l'instant t1"

- toute connaissance portant sur une entité qui soit déductible des interactions que celle-ci peut avoir avec une autre dans le même environnement :

"la chaussure de numéro de série 12234226 est à l'instant t1 portée par la personne dont le numéro INSEE est 1610506088323"

Pour se résumer, un fait est donc relatif :

- à une entité (ou éventuellement plusieurs) absolument identifiée,

- à l'instant de son observation.

De plus, c'est son "existence" à l'instant t qui en fait sa véracité. On pourra alors se poser les questions suivantes :

Q1 : Un fait a-t-il une certaine pérennité, pourrais-je m'appuyer sur une observation faite à l'instant t0 qui a précédé t1 pour vérifier sa validité à t1 ?

Q2 : Dois-je observer à l'instant t1 pour vérifier un fait à cet instant ?

Notre position - arbitraire, certes, mais plausible -, consistera à répondre oui à Q1 (et donc non à Q2).

Nous reviendrons sur ces problèmes en abordant la notion de "qualité" de l'information, où nous traiterons de la

question du temps comme facteur considérable dans l'appréciation de l'information.

1.1.2 D'autres types de connaissances

1.1.2.1 Globalisation des faits

Définir un fait tel que cela a été proposé permet de considérer des éléments distincts et de n'appréhender que des informations très élémentaires et "ponctuelles".

Or, l'observation d'un monde physique, quel qu'il soit, permet à l'être humain d'établir des règles générales, ou du moins concernant un ensemble de ces faits élémentaires, et que nous nommerons "**connaissances**" :

"Toutes les chaussures ont une couleur"

"Il existe une personne nu-pieds"

"Si elle ne porte pas de chaussure, une personne est nu-pieds "

et bien d'autres exemples qui sont soit des généralisations, soit des déductions, soit des assertions logiques, et qui, toutes, sont obtenues par analyse et abstraction à partir de plusieurs de ces entités mises en évidence précédemment.

1.1.2.2 Connaissance statique

Nous venons là de décrire des connaissances que l'on pourrait qualifier d'"absolues, certaines et statiques" :

- absolues, parce qu'elles ne traitent pas ce que

l'homme qualifie d'exception :

"Un cul de jatte n'est pas nu-pieds, mais n'a pas de chaussures"

- statiques, parce qu'elles ne portent que sur une observation instantanée du monde réel,

- certaines, parce qu'elles ne sont pas assorties d'une modulation quant à leur vraisemblance : elles sont vraies, dans tous les cas et tout le temps.

1.1.2.3 Connaissance dynamique

Elle prend en compte l'évolution temporelle de l'univers de discours :

"En France, un kaki est de couleur verte en automne et orangée en hiver"

concerne une même entité dont la qualité aura évolué dans le temps, mais ce d'une manière absolue, sans non plus traiter l'exception :

"Les kakis portés par un plaqueminier qui brûle le 21 décembre ont été verts en automne, mais ne seront pas orangés l'hiver suivant… "

vient invalider le caractère absolu de l'assertion précédente, même si sa probabilité de vraisemblance est faible. Nous venons là de mettre en évidence quelques nuances et précisions sur cette notion de connaissance telle que nous la présentions plus haut.

Un système d'information, tel que nous le concevons, à pour objectif de supporter une représentation abstraite,

"modélisée", mais aussi riche que possible, de ce monde réel dont nous parlons ici. Tout l'intérêt d'une structure d'accueil d'un SI sera dans sa capacité à autoriser l'expression de ces connaissances.

1.1.2.4 Connaissance opérationnelle

Une autre classe de "connaissance" peut encore être isolée : celle du savoir faire, ou connaissance opérationnelle. Elle a pour caractéristique de porter sur le traitement des informations précédentes, et s'apparente, à notre sens, à ce que les méthodes de modélisation les plus utilisées appellent "modèle des traitements" :

"Pour fabriquer une chaussure rouge, il me faut assembler une forme en cuir, une semelle, des oeuïllets et des lacets, puis teindre le tout en rouge" (?)

Nous nous interrogerons sur l'intérêt et les possibilités de traiter ce dernier type de connaissances, au sein d'un SI, et nous nous verrons si, dans notre approche "systèmes d'information", il devra être modélisé par les "applications", ou supporté par la structure d'accueil du SI.

1.1.2.5 Le concept de "contrainte"

Comme nous avons pu le voir, les connaissances portent soit sur des faits, soit sur d'autres connaissances générales. Or, quand elles portent sur des faits, ces connaissances reviennent souvent à des précisions apportées soit sur des conditions d'existence des faits, soit sur les modes de manipulations des objets qui sont sous jascents aux faits :

Une contrainte est une connaissance qui précise une

connaissance opérationnelle, qui restreint son mode d'exécution en limitant les possibilités de manipulations des objets.

Il est bon de remarquer ici que le Génie Logiciel et plus particulièrement la Gestion de Projets Logiciels sont des disciplines fortement "formalisantes", voire "contraignantes" pour les intervenants d'un projet informatique : organiser, rationaliser, c'est notamment contraindre. Logiquement donc, un grand nombre de connaissances relatives à ces domaines s'exprimeront assez naturellement sous forme de contraintes ; plusieurs modèles de données et connaissances de ces domaines l'ont montré [HURS86][CONC86].

1.1.3 Les métaconnaissances

C'est la connaissance que l'on a sur la connaissance opérationnele que l'on vient de définir:

"Nous savons fabriquer des chaussures rouges"

Au sujet de ces **"métaconnaissances"**, nous nous poserons aussi la question de savoir quel élément structurel doit les supporter.

1.2 Caractéristiques des informations Essai de formalisation

1.2.0 Introduction

Nous allons présenter ici quelques aspects, "points de vues" sur les informations et les connaissances.

Dans notre démarche, qui consiste à choisir un support à l'expression des connaissances pour notre futur système, il nous a semblé important de dégager certaines de leurs caractéristiques et de les formaliser selon des principes qui ressemblent à ceux définis par Mac Call [MACC79] et Boehm [BOEH83] sur la qualité des logiciels.

Une étude complète de ce point est en soit l'objet d'une thèse, et nous nous contenterons de poser simplement quelques jalons, qui nous aideront dans le choix de techniques de représentation.

1.2.1 Caractère existenciel et véridique des connaissances

En 1.1, nous assimilions existence et véracité d'une connaissance : pour préciser cela, il faut noter qu'une assertion est connaissance si elle est reconnue pour VRAIE absolument (dans l'univers considéré).

Divers principes de représentations sont envisageables :

- une connaissance "existe" si elle est représentée

explicitement :

un fait par une "ligne" dans une base de données,
une procédure par un programme,
…

pour donner quelques exemples.

- une connaissance peut aussi exister implicitement :

si les connaissances :

"les chaussures ont des semelles"
"Titi est une chaussure"

sont représentées dans le système d'information, alors la connaissance :

"Titi a une semelle"

y est implicitement contenue.

Ainsi, des connaissances de types différents peuvent se "contenir" l'une l'autre.

Ce qui n'est pas un problème dans le cas d'existence peut le devenir quand une connaissance existante, donc vraie, contient la négation, l'expression de l'inexistence d'une autre : elle doit implicitement empêcher la "création" d'une connaissance qui serait incohérente :

si la connaissance :

"les chaussures n'ont pas d'ailes"

existe, et est donc reconnue pour vraie,

"la chaussure Titi a une aile"

doit être reconnue comme fausse, et ne pas pouvoir exister.

Nous appellerons **fidélité** la qualité qu'a l'expression d'une connaissance de représenter, à un instant donné, la réalité correspondante, et ce au même instant. On rattachera ce concept à celui de "véracité".

La **consistance**, elle, définit la non contradiction d'une connaissance par d'autres éléments de la représentation d'un univers de discours donné.

1.2.2 Caractère temporel des connaissances

Ce premier point traitait d'une connaissance que nous avions qualifiée, en 1.1, de statique. Il nous appartient maintenant d'abstraire le caractère temporel des connaissances: Cet aspect est primordial en Génie Logiciel, et notamment en Gestion de Projets Logiciels, où le raisonnement par analogie et les références à l'histoire du projet sont fréquents.

- Un caractère "présent" :

La connaissance "instantanément" observée appartient au "contexte courant". Celui-ci contient toute connaissance qui a été établie à un instant antérieur et qui n'a pas été invalidée (donc supprimée) depuis :

Une connaissance appartenant à l'un des types définis en 1.1 reste vraie tant qu'elle est représentée dans le

contexte courant.

- Un caractère "passé":

Comment se poser, maintenant, le problème de la représentation de cette connaissance qui est vraie (contexte courant ?) mais qui porte sur le passé :

"Le conflit WW2 a commencé"

peut être considéré comme faisant partie du contexte courant si celui-ci veut représenter l'histoire des conflits dans le monde et de tout temps, mais doit être représenté différemment si le contexte courant ne veut s'intéresser qu'aux conflits d'actualité.

Nous essayerons, par la suite, de traiter de ce problème et de concevoir un langage qui prendra cela en compte.

- Un caractère "futur" :

Est il possible de dire, à l'instant t0, qu'une connaissance "sera" entre les instants td et tf, où tf > td > t0 ?

Oui, si cela signifie qu'elle sera "de toute façon" vraie à l'instant tj tel que tf > tj > td > maintenant du référenciel temporel de vérité de l'ensemble des utilisateurs de cette connaissance. Le "de toute façon" sous entend que la non existence de cette connaissance, dans le futur, serait dûe à des phénomènes remettant en cause à la fois le système d'information et l'ensemble de ses utilisateurs.

Les connaissances statisfaisant à ces critères constituent le "contexte futur" d'un système d'information.

- Le caractère évolutif de l'information :

Revenons sur l'exemple cité plus haut, sur l'assertion : "Le conflit WW2 a commencé". On pourra le compléter par une autre assertion : "Le conflit WW2 est terminé", si l'on se situe dans un contexte de temps postérieur à 1946, par exemple.

D'un point de vue temporel, cette nouvelle assertion ne constitue pas un fait nouveau (le fait indépendant du temps, c'est "Le conflit WW2"), mais simplement une modification (en l'occurence, une modification d'état) de celui ci : on dira qu'elle en définit ce que l'on appellera une nouvelle "version".

La **rémanence** sera, pour nous, la propriété qu'a l'expression d'une connaissance d'être interprétable, soit parce qu'elle est toujours vraie, soit, si elle ne l'est plus, parce que l'on sait encore quand elle a été vraie. On rattachera ce concept à la prise en compte du temps et de l'évolutivité d'un univers de discours.

1.2.3 Caractère hypothétique des connaissances

On vient de souligner, notamment à propos des connaissances futures, la nécessité de traiter de connaissances certaines. Pourtant, une "information" incertaine peut présenter, par exemple dans les systèmes d'information des systèmes d'aide à la décision (et, nous le rappelons, c'est l'un des aspects des ateliers logiciels qui sont le cadre de nos travaux), un caractère intéressant, voire important.

Précisons cette notion d'incertain :

Une connaissance est dite hypothétique dans les deux cas suivants :

- quand elle est sciemment supposée par un utilisateur du système d'information : on dira alors qu'elle fait partie du "contexte" d'une hypothèse.

- quand elle est déduite, par une utilisation normale du système, à partir à la fois de connaissances qui font partie du même contexte d'hypothèse et d'autres qui sont certaines.

Pour les premières se pose le problème de leur éventuelle incohérence avec l'état des connaissances à l'instant de leur supposition : peut-on supposer vrai une connaissance I1 si on peut la déduire fausse à partir d'autres connaissances In ?

On peut répondre affirmativement, si l'on suppose la base de connaissances évolutive, et donc les In susceptibles de disparaitre (en rendant par là même I1 plausible).

Pour les secondes se posera le problème de leur validation en cas de vérification des hypothèses dont elles sont déduites.

On s'intéressera donc à comment il est possible de faire vivre ces connaissances incertaines, en les validant ou les invalidant.

Nous nous attacherons aussi à les lier au caractère temporel des connaissances "certaines".

1.2.4. Granularité, densité et degré d'abstraction des connaissances

1.2.4.1 Comparaison

Peut on comparer faits, connaissances, documents et métaconnaissances en termes de fréquence (ou de quantité) dans le monde réel ? Peut on définir une densité d'informations "atomiques"(?) par connaissance ?

Une constatation triviale consiste à dire qu'une connaissance générale est plus dense que l'expression d'un fait particulier :

La "connaissance" :

"Toutes les chaussures ont une semelle"
"Titi est une chaussure",
"Tata est une chaussure",
"Toto est une chaussure",
"Tutu est une chaussure",
…

est plus "dense" que la liste (non exhaustive) suivante :

"Titi a une semelle",
"Tata a une semelle",
"Toto a une semelle",
"Tutu a une semelle",
…
"Titi est une chaussure",
"Tata est une chaussure",
"Toto est une chaussure",
"Tutu est une chaussure",

La densité d'information contenue par l'expression d'une certaine connaissance est fonction du potentiel d'inférence du système dans lequel on exprime cette connaissance :

Ici, l'inférence triviale consiste à déduire, pour toutes les chaussures que l'on sera ammené à considérer, qu'elles ont une semelle. L'économie dans l'expression des connaissances est évidente.

Le mode d'expression de la connaissance sera étroitement lié aux mécanismes d'inférence des systèmes supports : Leur capacité à représenter des informations dépend non seulement des moyens d'expression explicites des connaissances mais aussi des moyens associés de déduction d'informations implicites.

Nous parlerons de **granularité** pour définir la capacité qu'a un moyen de représentation de la connaissance, d'exprimer les différents degrés d'abstraction et de généralisation nécessaires à la représentation d'un univers réel.

Nous parlerons aussi de la **sélectivité** d'un moyen d'expression de la connaissance, qui permet plus ou moins de représenter des connaissances ponctuelles qui font exception à d'autres connaissances plus générales, mais reconnues comme non absolument vraies.

Une granularité étendue posera plus de problèmes de consistance qu'une granularité restreinte (parce qu'il est plus facile de maintenir une consistance entre niveaux de représentation homogènes qu'hétérogènes). De même, il ne se posera de problème de sélectivité que dans des cas d'univers à granulométrie étendue.

Nous dirons enfin que la **modularité** qualifie l'indépendance relative de groupes de connaissances, au sein d'un même univers de discours, et l'aptitude d'un outil de représentation des connaissances à la respecter.

1.2.4.2 Degrés de modélisation des connaissances

On a vu qu'il était intéressant de gérer d'une manière très "stricte", certaines informations, en s'employant à en assurer l'intégrité sémantique maximale, parce que leur criticité est importante : c'est le cas des faits et connaissances.

Mais il peut être aussi utile de gérer certains supports de la connaissance, que l'on peut appeler **"documents"**, dans lequel l'information critique est diluée, éventuellement redondante, mais au profit d'une représentation plus compréhensible par l'être humain :

Pour nous, un document contiendra donc une connaissance moins abstraite que les types cités plus haut. Il constitue une expression "terminale" de la connaissance (plutôt qu'une connaissance en soi), et on ne le considèrera qu'en "lecture" : il ne requiert pas un controle sémantique aussi fort que les types de connaissances plus abstraits présentés plus haut.

Toutefois, il sera nécessaire et utile de connaître le degré de validité de la connaissance contenue dans un document (et donc sa validité propre), en liant documents et connaissances. La structure que nous souhaitons mettre en place s'efforcera de le permettre.

1.3 Transition

Volontairement, cette étude a traité des "connaissances" d'un emanière fort générale, avec peu de références au monde de la gestion de projets logiciels et du Génie Logiciel. Nous verrons, dans les chapitre 2.7 et 4 quels sont, parmi ces concepts, ceux qui sont utiles dans ces domaines, et nous nous rendrons compte que tous, ou presque, le seront : c'ets d'ailleurs pour cette raison que , selon nous, une structure d'accueil pour le SI Génie Logiciel est sans doute utilisable dans bien d'autres domaines.

Nous allons maintenant présenter un état de l'art des différentes techniques de représentation des connaissances qui sont "candidates" pour supporter un système d'information étendu.

Nous en retiendrons celle qui nous paraîtra assurer l'expression d'un très grand nombre des concepts utiles au Génie Logiciel, ou qui, du moins, pourra être facilement étendue pour les supporter.

2. Techniques de représentation des informations

2.0 Introduction

Nous allons présenter ici diverses techniques en nous attachant à mettre en évidence, pour chacune d'elles :

- leurs concepts de bases,

- comment les types d'informations présentées en 1. peuvent être modélisés, en essayant de préciser les difficultés rencontrées généralement, ou bien même les impossibilités.

Nous essayerons, chaque fois, de nous placer à un niveau "conceptuel". Toutefois, si l'on sent bien que, pour chaque technique, il existe un cycle de vie de leur mise en oeuvre, la définition d'un cycle de vie générique nous semble bien hasardeuse : en effet, selon la "puissance", le pouvoir d'expression des formalismes de ces différents outils, la "spécification", la "conception" ou même l'"implémentation" d'un monde peut désigner des activités fort diverses. Nous traiterons en fait de ce qu'il est possible d'exprimer en utilisant assez simplement les concepts de base supportés par les formalismes des techniques étudiées.

Nous parlerons des supports que l'on peut qualifier de "peu formels" (les réseaux sémantiques, les représentations centrées objets, ...), puis des supports inspirés de la logique formelle, des systèmes de règles de productions, et de ceux qualifiés généralement de "systèmes de bases de données".

Nous terminerons en examinant quelques produits de l'hybridation de plusieurs de ces outils de représentation.

Notre conclusion, au terme de cette étude, consistera à choisir la ou les techniques qui supporteront notre système

d'information étendu.

Sauf exceptions, qui seront précisées, nous suivrons dans ce chapitre, et pour chaque représentation étudiée, la plan suivant :

- Concepts.

- Modélisation des informations par l'utilisation de ces concepts.

- Qualité de la représentation des informations représentées :

    - Véracité et validité,
    - Prise en compte du temps,
    - Prise en compte de données hypothétiques,
    - Synthèse sur la qualité.

- Conclusion et remarques.

Nous ne présenterons pas, en principe, les manipulations possibles de ces concepts, dans la mesure où il s'agit généralement d'insertions, suppressions, modifications et recherches de leurs occurences, et qu'elles sont le plus souvent assurées par les systèmes qui les implémentent. Toutefois, nous soulignerons les éventuelles exceptions à cette règle.

- liens sémantiquement définis par leur nom :
  "a pour valeur courante",
  "mesure",
  ...

Ce formalisme initial a été étendu progressivement par de nombreuses équipes, dont [HEND75] [SHAP77] [DELI79] [FAHL75] [BRAC77],dans le but de suppléer à certaines lacunes et à des ambiguïtés à propos de la sémantique des liens (notamment ceux induits par la non différenciation de la "classe" (abstraction) et de l'"individu").

2.1.2 Modélisation des connaissances par les RS

- Les faits et les connaissances

Pour les plus simples, ils se modélisent par un usage des concepts de base. Pour les plus complexes, ils ont nécessité des extensions que nous présentons ici :

Par exemple, la quantification des entités et la définition des "espaces" qui partitionnent un réseau et établit des liens entre partitions [HEND75]. Ces ajouts lèvent l'ambiguïté "appartenance"/"inclusion" et enrichissent le pouvoir expressif du formalisme :

"Il existe un homme qui porte une chaussure" peut être alors représenté :

2.1 Les réseaux sémantiques (RS)

2.1.1 Les concepts de la modélisation des connaissances par les RS

Un réseau sémantique est un graphe orienté dont les noeuds représentent des entités logiques, et les arcs des relations binaires entre ces entités. On peut attribuer leur paternité à Raphael et Quillian [RAPH68][QUIL68].

Les relations y sont typées :

- liens isa (définition de sous-ensembles),
- liens d'appartenance (définition des éléments d'un ensemble),
- liens "agents" et "objets", qui permettent de définir des événements :

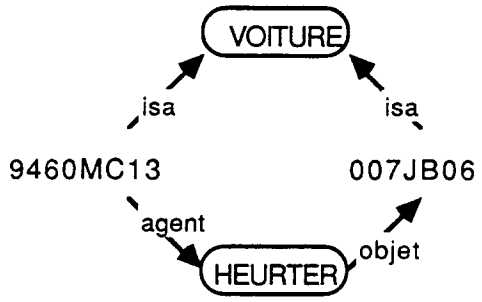


fig. 3 : RS, exemple n°1

qui se traduit par :
"La voiture 9460 MC 13 heurte la 007 JB 06"

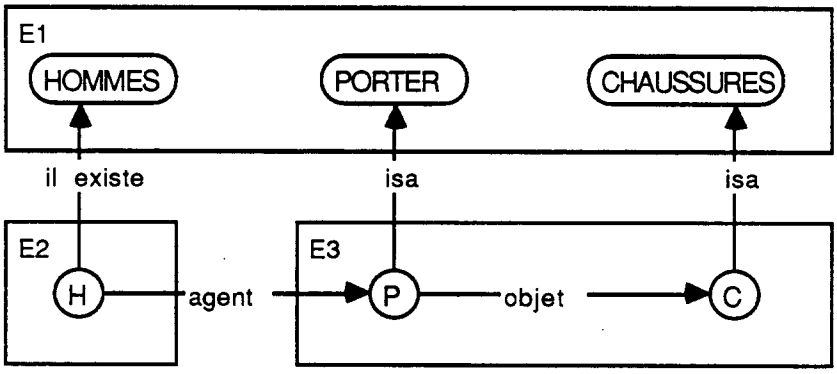


fig. 4 : RS, exemple n°2

où E1 représente l'espace des entités basiques : les hommes, les chaussures, le fait de porter un vêtement,

où E2 représente "un homme quelconque" et le fait qu'il existe (lien "existe"),

où E3 traduit l'évènement "porte une chaussure",

et où ((E1,E2), (E1,E3)) est la "vue" qui exprime la connaissance énoncée ci dessus.

Fahlman [FAHL75] définit des liens d'exception (qui permettent d'exprimer le célèbre "Kiwi est un oiseau mais il ne vole pas" !), et Brachman [BRAC77] commence à séparer concepts, attributs des concepts, attributs génériques, attributs spécifiques, valeurs d'attributs, … (qui préfigurent l'approche objet décrite par ailleurs).

Deliyanni et Kowalski étendent les réseaux sémantiques en leur donnant une syntaxe qui permet de représenter les "Kowalski clausal forms" [DELI79], soit en abbrégé, des conaissances du type :

"Si a appartient à A, et si A est un B, alors a appartient à B",

ainsi que les négations (par des liens "en pointillés") :

"Pierre n'est pas marié à Marie", qui se représente par :

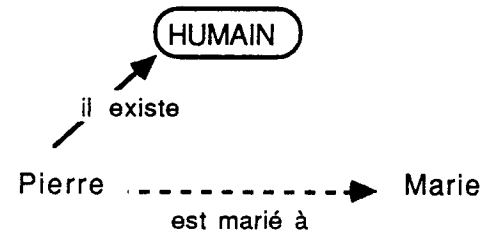


fig. 5 : RS, exemple n°3

On se rapportera à [FROS86] pour de nombreux exemples intéressants.

- Les métaconnaissances

On notera que les réseaux sémantiques sont aussi appropriés pour la représentation des métaconnaissances, dans la mesure où celles-ci peuvent être exprimées en usant du même formalisme, sur le même schéma que les connaissances.

2.1.3 Qualité des Informations en RS

2.1.3.1 Véracité et négation

On l'a vu précédemment, des extensions diverses permettent de définir assez précisémment négations, exceptions et comportements généraux.

Ainsi, les systèmes RS connaissent certaines informations comme vraies (par leur existence, soit par l'expression sous forme de concept abstrait, soit par instanciation), mais aussi comme fausses (par la possibilité offerte par Kowalski [KOWA75] d'exprimer la négation).

2.1.3.2 Notion de Temps en RS

La représentation du temps dans les réseaux sémantiques peut être assurée :

- par la datation des événements, pour ce qui concerne le séquencement logique des états :

Un attribut "date", ou un indice, est lié à chaque concept "événement" et l'estampille.

Kowalski a proposé une méthode d'ordonnancement des évènements qui permet de propager une modification du monde réel.

- par la mise en place d'interpréteurs gérant l'histoire des réseaux, leur évolution dans le temps.

Pour cela, il est nécessaire d'associer une "période de validité" à chaque lien :

"John a été marié à Mary du 01/01/85 au 31/12/86, et est marié à July depuis le 01/01/87" se traduira par :

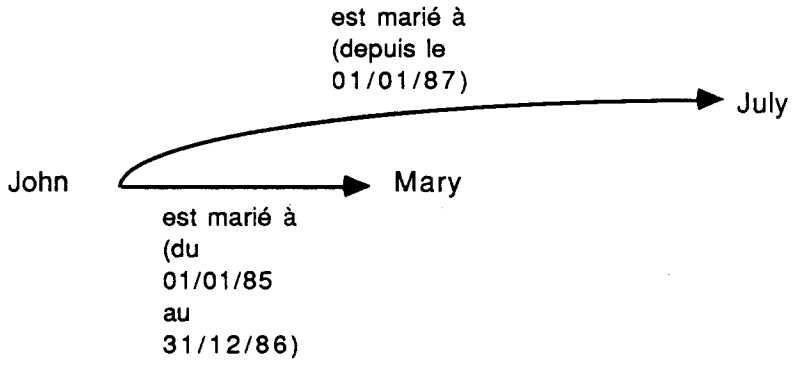


fig. 6 : RS, exemple n°4

et ceci indépendemment de l'estampillage des évènements (qui bien sur peut subsister sans contradiction).

2.1.3.3 Notion d'hypothèses en RS

A notre connaissance, à ce jour, aucun système n'interprète ce type d'information, ni n'assure la gestion de ce que nous définissions en 1. comme "informations hypothétiques".

2.1.3.4 Synthèse sur la qualité des RS comme support de connaissances

Si l'on essaie d'évaluer la qualité des RS en tant que support de l'information, en utilisant les "facteurs" présentés en 1.3, nous dirons que, dans l'ensemble, les RS sont très satisfaisants, mais que leur rémanence n'est suffisante que s'ils sont étendus comme montré en 2.1.3.2.

De plus, d'une sélectivité forte (cf. exemple du

Kiwi), les RS ne savent pas bien mettre en évidence les inconsistances.

Enfin, on a vu que l'expression des connaissances par les RS n'était pas très modulaire (pour s'en convaincre, il suffit de donner un coup d'oeil à un graphe bien choisi ...), et que les RS ne marquaient les différents degré d'abstraction qu'à travers la sémantique des arcs.

2.1.4 Conclusion et remarques

Les RS sont principalement un moyen d'expression, de description de la connaissance, mais ne peuvent apporter des informations nouvelles que supportés par des interprètes capables d'inférer (et donc de modifier un réseau existant) :

Ils peuvent être considérés comme structures de données, comme support de la connaissance pour un système d'information [CARB70], à condition de leur associer un outil d'interrogation.

On soulignera enfin les ressemblances entre les réseaux sémantiques étendus et les représentations centrées objet ou autres scripts.

2.2 Frames et objets

2.2.1 Les concepts

On trouvera, dans la littérature, différents termes pour désigner la "famille" de techniques de représentation des connaissances : frames, prototypes, objets structurés, unités, scripts, …

Toutes ne sont pas strictement équivalentes, mais sont suffisamment ressemblantes pour que nous parlions ensuite de "représentations centrées objets" [RECH87].

Leurs origines sont à rattacher aux frames de Minsky [MINS75], aux réseaux sémantiques (cf. 2.1), et aux langages orientés-objet comme Smalltalk [GOLD83].

Leurs caractéristiques essentielles sont les suivantes :

- la structuration en objets, attributs ou slots (qui peuvent être des objets, c'est l'"encapsulation"), et facettes (qui caractérisent les modes de manipulation des attributs),

- l'implémentation de hiérarchies par spécialisation (classes, sous-classes, …) : une classe (abstraction) est une structure de données générique représentant un stéréotype d'objet réel,

- des mécanismes d'inférence comme :

. l'héritage, qui permet à un objet de posséder

qu'il soit nécéssaire de les réécrire,

. l'attachement procédural, qui autorise une définition fonctionnelle de la valeur d'un attribut,

. les valeurs par défaut, affectables aux attributs,

. la classification, qui autorise la comparaison, pour le "classer" (trouver ses plus proches parents), entre un objet et plusieurs autres.

Un système d'information fondé sur une telle technique consite donc en un ensemble d'objets structurés, tous individuellement identifiés et interreliés par le biais des attachements procéduraux ou des héritages [ZANI86].

Son utilisation passe par la recherche, pour une situation donnée, du ou des objets les plus adaptés et qui sont en fait la solution du problème.

2.2.2 Modélisation de connaissances en représentation centrée objet (RCO)

- Les faits

Assez évidemment, ce que nous appelions "objet atomique", en 1.1.1, s'implémentera sous la forme d'une instance de classe :

On définira d'abord l'objet générique "chaussure" (classe), avec des attributs, dont "couleur" et "identifiant".

est "titi", en affectant "rouge" comme valeur de l'attribut "couleur" de "titi".

- Les connaissances

Les connaissances peuvent se définir assez simplement, soit par la modélisation statique (définition du frame et de ses attributs) :

"Toutes les chaussures ont une couleur"

se traduira par l'existence de l'attribut "couleur" pour la classe "chaussure", soit par modélisation "dynamique" par les attachements procéduraux qui permettent d'exprimer des contraintes sur la manipulation des objets (modifications conditionnelles, …).

Ces mêmes attachements procéduraux supportent l'expression des connaissances opérationnelles.

- Les métaconnaissances

sont implémentées par des classes qui font référence aux classes utilisées pour supporter faits et connaissances : il n'y à pas, en RCO, de différences majeures entre l'implémentation de ces deux concepts.

2.2.3 Qualité des Informations en RCO

2.2.3.1 Véracité et négation

Une information sera "valide" si les valeurs qui la constituent, et qui ont été obtenues par consultation du monde

des objets modélisés, n'y ont pas été modifiées depuis : c'est ce monde "modèle" qui devient référence, en se substituant au monde réel.

On notera que pour un objet identifié, la non affectation d'une valeur à l'un de ses autres attributs ("couleur" par exemple) sera interprêtée comme un constat d'ignorance du monde réel (ignorance de la couleur).

L'utilisation des valeurs par défauts, autre concept des RCO, engendre la possibilité de "supposer" des valeurs vraies parce que ce sont les plus couramment observées : Une valeur obtenue par inférence sur la règle "par défaut" a ainsi une "véracité" relative par rapport aux valeurs obtenues par interaction ou par d'autres moyens de déduction.

La gestion de valeurs "hypothétiques",on le verra plus bas, peut modifier ces axiomes.

2.2.3.2 Notion de Temps en RCO

La modélisation de données "passées" n'est pas intinsèque aux RCO, mais elle peut être réalisée de différentes manières :

- par l'utilisation d'objets "timestamped", estampillés, dans [COPE84] [FERN87], qui sont copiés et timbrés à la date de modification quand ils évoluent,

- par la définition de couples (période de validité - valeur) en lieu et place des valeurs simples terminales (feuilles des objets qui sont de types simples : chaine, entier, réel, ...).

Dans ces deux cas, des mécanismes de recherche

particuliers (filtrant sur les dates) sont nécessairement rajoutés.

Il n'existe pas, à notre connaissance, de mécanisme implémenté qui prenne en compte l'intégration d'informations "futures".

2.2.3.3 Hypothèses en RCO

Non inclue dans les concepts originels des RCO, la gestion d'informations hypothétiques a été implémentée par des moteurs dérivés, comme NOMADE [MELL87].

Elle s'appuie sur le marquage "OUT" des objets, plutôt que leur effacement, leur recopie plutôt que leur modification,... et autorise ainsi des retours arrières sélectifs par des restaurations simples d'états stables.

2.2.3.4 Synthèse

On résumera notre appréciation des RCO en disant que la fidélité des informations représentées est subordonnée à l'usage que l'on veut faire des valeurs par défaut, et leur rémanence aux extensions éventuelles présentées en 2.2.3.2.

Nous retiendrons, comme pour les RS, leur sélectivité, en remarquant que la concentration des connaissances autour d'objets est un plus, en terme de modularité.

Nous rajouterons enfin que les RCO existantes n'ont pas pour vocation la représentation d'informations de très grande taille et ne sont pas adaptés à la gestion de documents stockés sur une mémoire de masse. Toutefois, la réalisation d'un tel système est possible par interconnexion entre un

système RCO (utilisé pour modéliser certaines informations sur les documents, comme l'auteur, le titre, les domaines, ...) et un système de gestion de fichiers [BOSC88a].

Enfin, nous soulignerons que les RCO ne sont généralement pas conçus pour qu'un utilisateur puisse naviguer au sein du monde des objets, grâce par exemple à un langage de requête sur ces objets : Il ne pourra se déplacer, le plus souvent, qu'en "câblant" ses chemins dans des attachements procéduraux.

2.2.4 Conclusion et remarques

On notera, en guise de conclusion à ce paragraphe, que les RCO possèdent un pouvoir de représentation fort, sont souples d'utilisation et permettent d'implémenter des fonctionnalités de base de systèmes d'information facilement modifiables.

Elles ne favorisent pas l'un ou l'autre des paradigmes de la représentation des connaissances, et autorisent une approche pragmatique et "physique" (le concept d'objet est un concept physique) de la modélisation.

Toutefois, elles n'implémentent pas directement les mécanismes complexes (gestion des hypothèses et des informations historiques) qui nous intéressent.

De plus, on soulignera l'absence de langage d'interrogation de la base d'objets qui soit générique, et les difficultées technologiques rencontrées dans la manipulation de bases importantes et qui nécéssitent un stockage sur mémoire de masse [ZANI86].

2.3 Logiques formelles

2.3.1 Les concepts fondamentaux

Une logique formelle est constituée d'un langage pour exprimer la connaissance (formules), et de règles d'interprétation de formules (règles d'inférences, comme le "modus ponens", ou la "substitution").

Une formule est une combinaison de propositions atomiques :

"La chaussure 007 est bleue",
"La chaussure 007 est pointue", …

et d'opérateurs logiques :

ET, OU, NON, IMPLIQUE

d'où, par exemple, la formule :

"La chaussure 007 est bleue ET la chaussure 007 est pointue".

Les règles d'inférence permettent, à partir d'un ensemble de formules, d'en prouver d'autres :

"Titi est une chaussure",
"Titi est en cuir",
"Un objet qui est une chaussure ET qui est en cuir IMPLIQUE qu'il est putrescible"

permettent de prouver :

"Titi est putrescible"

On reconnaitra plusieurs types de logique :

- la logique propositionnelle (qui manipule des formules invariantes),

- la logique des prédicats du premier ordre (dont les formules peuvent contenir des variables quantifiées),

- la logique des prédicats à variables typées (i. e. d'un "type" conceptuel donné),

- la logique des prédicats à variables contraintes (i. e. vérifiant une formule mathématique donnée) [COLM87],

- la logique des prédicats du 2nd ordre, où les identifiants de prédicats peuvent être variables et quantifiés,

- des logiques particulières :

. non booléenne, à trois état : vrai, faux, inconnu,

. non booléenne, dite floue, [DUBO87], où un degré de certitude est affecté à chaque formule, et calculé pour les formules déduites,

. modale, dont l'origine remonte au début du siècle [LEWI32], et qui inclut des extensions à l'expression des relations au sein des formules, des opérateurs "modaux" qui vérifient différents axiomes (on verra une application à la logique temporelle),

. situationnelle [MACC69], dont l'objet est de

représenter un monde en mouvement, et ses objets en relation situationnelle (sur, sous, dans, …). C'est aussi une application de la logique modale, où les opérateurs vérifient :

op [P Q] op [Q R] -> op [P R]

où op vaut SUR, SOUS ou DANS.

. non monotone [MACD80], [DOYL82], qui permet de définir des assertions comme étant vraies ou fausses, selon des hypothèses de raisonnement précises (selon certains modes de raisonnement, où l'on rejoint la logique modale),

. intensionnelle [MONT73], dont l'objectif était d'utiliser les apports de différentes logiques (modale, temporelle, …) pour traiter des connaissances en fonction de leur "monde contexte" (en donnant une définition de chaque entité selon le contexte dans lequel on la considère) et d'instants ordonnés (temps).

2.3.2 Modélisation de connaissances par la logique

- Les faits

Ils sont évidemment représentés par des formules atomiques. A priori, ils ne sont pas "organisés", et les liens structurels (relations entre faits) n'existent pas en logique propositionnelle (ou d'ordre 0).

Cette structuration devient possible dans la logique des prédicats du premier ordre, par la sémantique donnée aux identificateurs de prédicats [BOUB85]:

homme(Grandblond) ->;

homme(Petitbrun) ->;

sont des instances de l'entité-type "homme".

appartient_a(homme(Grandblond), chaussure(Titi))->;

est, par exemple, la représentation de la relation "appartient_a" entre deux entités "homme" et "chaussure".

- Les connaissances

Comme on l'a vu, la présence, en logique des prédicats du premier ordre, du quantificateur universel, autorise les généralisations.

Le "implique" permet d'exprimer des règles de comportement.

Les logiques sont utilisées en mathématique pour prouver, par inférences sur des formules, de nouveaux théorèmes. Elles permettent donc de modéliser d'une manière très formelle une part appréciable de la connaissance que l'on peut avoir d'un monde réel.

Toutefois, - et c'est la motivation des travaux présentés dans un paragraphe précédent -, des lacunes existent, que des logiques plus complexes essaient de combler (par l'utilisation de variables contraintes, manipulation de l'incertain, gestion temporelle, ...).

- Les métaconnaissances

L'expression des métaconnaissances est assurée par des "méta-formules" qui permettent d'orienter le choix des

règles à utiliser par les mécanismes d'inférence.

2.3.3 Qualité des Informations en logique

2.3.3.1 Véracité et négation

La logique permet de définir explicitement assertions et négations.

L'absence d'une formule nécéssaire à l'inférence est traduite par un échec dans la résolution (on reviendra en 3. sur cette notion), et donc par la négation de ladite formule.

Les mécanismes d'inférence permettent la manipulation de connaissances implicites.

2.3.3.1 Notion de Temps en Logique

Ce thème est l'objet de multiples travaux [SERN80][LUND82].

En logique des prédicats du premier ordre, une approche consiste à introduire le concept de "point de temps", soit en fait une entité à part entière, ce qui revient à introduire le temps dans le modèle du monde considéré.

Ces points de temps sont "reliés" par des prédicats descriptifs de la sémantique de leur ordonnancement :

```
précède(t1, t2) ->;
succède(t1, t2) ->;
```

Cette approche est assez comparable aux travaux de [MACC69] sur la logique situationnelle.

Une autre intégration du concept "temps" est inspiré de la logique modale présentée précédement : elle implémente les concepts de "toujours" et "quelquefois" (par analogie avec "nécessaire" et "possible"), de passé (P) et de futur (F). Ces concepts peuvent être combinés pour déduire des informations intéressantes :

toujours (t) implique quelquefois (t)
toujours (t) implique t
t implique quelquefois (t)
quelquefois (t) implique non (toujours (t)) non (t)

P(f) = f a été vrai
F(f) = f sera vrai

qui peuvent donner :

non P non f : f n'a jamais été faux
non F f : f ne sera plus jamais vrai
…

Ces théories ont pour l'instant trouvé peu d'applications pratiques, mais peuvent être utilisées notamment pour la compréhension des langues naturelles.

On verra que le génie logiciel est un domaine fortement consommateur de règles portant sur le passé (analogie) et sur le futur hypothétique (planification, scenari, …).

2.3.3.3 Hypothèses en logique

La notion d'hypothèse est implicite à la logique : vérifier une formule revient à faire l'hypothèse de validité. Des mécanismes de résolution font "naturellement" de la

gestion d'hypothèse (backtrack), mais ces "hypothèses" restent invisibles et inaccessibles à l'utilisateur.

Toutefois, il est possible et intéressant de réaliser des mécanismes traitant le "what-if?" sur des moteurs d'inférence logique : faire une hypothèse revient, au sein d'une transaction, à modifier la théorie logique et à lancer la résolution d'un but dans ce nouveau modèle, pour revenir ensuite dans l'état voulu. La gestion des hypothèses passe donc par une "métamanipulation" logique réalisable sur la base d'un outil qui implémente une logique.

2.3.3.4 Synthèse

Nous verrons, dans le paragraphe suivant consacré aux systèmes à règles de production, que la qualité d'une représentation des connaissances basée sur la logique dépend considérablement de son implémentation.

Toutefois, les caractéristiques notables d'une représentation logique sont la consistance vérifiable des connaissances qu'elle supporte, et sa sélectivité.

Sa modularité est faible : les connaissances, exprimées en vrac sous forme de formules. Elle ne le serons que grâce à des fonctionnalités particulières implémentées par tel ou tel outil (par exemple certains outils logiques orientés objets, comme LAP [ILIN86], et des systèmes à règles de production).

2.3.4 Conclusion et remarques

Les logiques sont des moyens très importants de représentation des connaissances.

Des logiciels de résolution de problèmes ont été réalisés, qui s'appuient très fidèlement sur des concepts mathématiques : cette formalisation est un avantage appréciable sur des outils plus souples mais au comportement moins formel (comme les RCO par exemple).

On notera d'ailleurs que le grand intérêt des logiques "étendues" réside dans la formalisation qu'elles réalisent d'une classe particulière des connaissances relatives au monde réel. De nombreux travaux montrent leur puissance, et permettent de penser que leurs développements apporteront encore beaucoup à la modélisation des connaissances.

2.4 Systèmes à règles de productions (SRP)

2.4.1 Les concepts

Un SRP est la combinaison :

- d'une base de données,

- d'une base de règles,

- d'un module d'interprétation des règles.

Il peut être considéré comme un outil implémentant une logique (cf. § précédent), mais présente aussi des aspects non logiques, ou "systèmes", de manipulation des connaissances.

La base de données a pour vocation de stocker des connaissances dites "simples", que l'on peut assimiler à ce que nous décrivions comme "faits atomiques" dans notre première partie.

La base de règles décrit l'ensemble des autres connaissances explicitées en 1., et ce sous la forme :

<< SI prémisses ALORS actions >>,

où les prémisses sont un ensemble de faits, et les actions une série de manipulations (insertion d'un nouveau fait, interaction avec l'utilisateur, …).

L'interpréteur de règles, lui, est la partie active du système. Son rôle consiste à parcourir l'ensemble des règles :

- en déclenchant, pour celles dont les prémisses sont vérifiées, la série d'actions correspondantes, puis en recommençant sur la base de données ainsi modifiée (moteurs en chainage avant),

- en visant un but, en le cherchant dans toutes les règles, en partie action, et en essayant de résoudre leurs prémisses (qui deviennent eux même des sous-buts) (moteurs en chainage arrière).

2.4.2 Modélisation de connaissances par les SRP

Comme nous venons de le remarquer, données et faits, règles et connaissances, sont en quasi équivalence : en effet, les SRP représentent, pour de nombreux chercheurs [SIMO69][FEIG71], les outils informatiques dans lesquels le mode d'exécution du raisonnement et de représentation des connaissances sont les plus proches de la nature humaine.

- Les faits

Selon le principe des SRP, qui sont par essence "déclaratifs", les faits sont exprimés "en vrac" [LAUR82], c'est à dire aussi indépendamment que possible les uns des autres : cela est intéressant, du point de vue de la modularité et des possibilités que l'on a de modifier les faits, mais devient néfaste à la consultation de ces faits, lorsque leur nombre augmente considérablement.

- Les connaissances

Exprimées par les règles, elles le sont donc toutes sous la forme :

SI ... ALORS

C'est la puissance du formalisme d'expression des règles qui définit la richesse des connaissances que l'on peut ainsi modéliser :

- les systèmes "bas de gamme" n'acceptent que des expressions de la logique propositionnelle, comme :

SI couleur = noir ALORS propriétaire = pompe_funèbre

(où les expressions ne contiennent que des valeurs constantes.)

- des systèmes plus puissants autorisent le calcul de ces valeurs :

SI hauteur < 1/2 (largeur + profondeur) ALORS type = objet_compact

- d'autres encore permettent l'utilisation du quantificateur existenciel et de variables (ils interprètent des règles de la logique du premier ordre, décrite par ailleurs),

- enfin, les interpréteurs les plus évolués admettent quantificateurs universels et existenciels.

On notera que les SRP sont particulièrement adaptés à l'expression de connaissances opérationnelles.

- Les métaconnaissances

sont aussi exprimées au moyen de règles de

productions qui décrivent la connaissance du sytème par lui-même :

<< SI la couleur de la chaussure est rouge ET que son propriétaire n'aime pas le rouge ALORS ***activer*** les règles qui expriment l'action de ***teinter une chaussure*** >>.

2.4.3 Qualité des Informations en SRP

2.4.3.1 Véracité et négation

L'extrême modularité des systèmes à base de règles donnent à leur utilisateur une très grande latitude dans l'expression de la connaissance. Ils permettent, de plus, de définir assertions et négations.

Un aspect intéressant [LAUR82] de ce mode d'expression réside dans la possibilité de vérifier syntaxiquement une partie de la cohérence des règles :

P1 Δ P2 $\rightarrow$ A

ne peut coexister avec :

P1 Δ P2 $\rightarrow$ $\neg$A

2.4.3.2 Notion de Temps en SRP

La connaissance exprimée à travers une règle de production correspond à une connaissance statique présente. Gérer l'évolution des connaissances rend nécessaire l'estampillage temporel des faits et règles, et la définition d'un moteur capable de manipuler et d'interpréter ces règles.

2.4.3.3 Hypothèses en SRP

Quelques systèmes, à l'heure actuelle, implémentent un concept de "point de vue", i. e. un ensemble d'assertions hypothétiques sur lesquelles un raisonnement peut être mené, et que l'on peut, à volonté,. détruire ou valider [WILL86].

2.4.4 Conclusion et remarques

Un avis positif s'impose ici : les règles de productions sont très adaptées à la structuration du savoir humain, et la possibilité qu'elles offrent de définir des connaissances de manière modulaire (c'est un apport des SRP par comparaison avec la logique fondamentale) est primordiale.

Les difficultés résident dans l'écriture des règles (dont la structure rigide est contraignante), et la gestion d'une grande quantité de faits (manipulation et accès).

Ce dernier point est d'autant plus crucial que la base de fait est un composant fortement intégrant dans le cas de systèmes multi-experts [FAJO88], ou de systèmes aux fonctionnalités multiples, mais qui partagent un grand nombre d'informations [JENK87].

2.5 Systèmes de gestion de bases de données (SGBD)

Le plan adopté dans ce paragraphe différera quelque peu des précédents : il existe plusieurs types de SGBD, qui ont points communs mais aussi différences.

Nous commencerons par présenter, en deux parties, les concepts utilisés par les techniques SGBD. Ensuite, nous parlerons de plusieurs formalismes BD, pour reprendre enfin le plan habituel à ce type de paragraphe.

2.5.1 Les définitions élémentaires

2.5.1.1 Un système de gestion de base de données est composé essentiellement :

- d'une "base" de données, c'est à dire un ensemble important de faits implémentés sur un support informatique,

- de langages d'accès à cette base,

- d'un certain nombre de mécanismes, que nous allons détailler, destinés à protéger la base de données.

2.5.1.2 La base de données est constituée d'éléments, que nous appellerons "enregistrements", pour l'anglais "record". Chacun de ces enregistrements représente une information simple.

Leur organisation physique, dans le détail de laquelle nous n'entrerons pas, est dépendante de choix d'implémentation. Leur organisation "conceptuelle", c'est à

dépendante de choix de modélisation que nous aborderons ci-après.

2.5.1.3 Un enregistrement comporte une série de champs valués, que l'on peut assimiler à la représentation d'un fait atomique comme défini en 1.

Les champs sont typés, et certains systèmes permettent la définition de domaines de valeurs (comme définis dans le paragraphe consacré aux RCO).

Les valeurs des champs deviennent "données" (data) lorsque elles sont utilisées dans un processus de calcul, ou "traitement".

2.5.1.4 Un traitement est une séquence d'opérations dont les paramètres sont des informations présentes dans la base de données, ou bien déduites de celles ci, ou bien encore obtenues par interaction avec un utilisateur.

2.5.2 D'autres concepts

2.5.2.1 Pour la conception d'une base de données

Les SGBD sont des outils d'une certaine manière très proches de la machine. L'implémentation d'une base de données nécessite donc, comme toute réalisation informatique, l'application de diverses phases, qui forment le cycle de vie de cette base.

Les techniques des BD sont aujourd'hui dans l'ensemble bien connues, et de nombreuses méthodes ont été établies. Elles consistent à définir des représentations de l'univers de discours considéré qui aillent progressivement des plus conceptuelles au plus proche de la représentation

machine.

On parlera de schéma (ou niveau) conceptuel, qui constitue une première abstraction, une description de l'univers choisi. Il intègre l'identification des objets (entités) de cet univers, des liens statiques (relations) entre ces objets, et ce que l'on appellera des contraintes, qui précisent les caractéristiques de ces objets et liens.

Le schéma logique, lui, traduit l'expression contenue dans le schéma conceptuel en terme de données qui pourront être utilisées par les traitements. Il intègre la prise en compte des concepts de base du SGBD utilisé : les objets du schéma conceptuel sont projetés sur des relations, si l'on utilise par exemple un SGBD relationnel.

Le schéma physique (ou niveau interne), enfin, décrit le mode d'implémentation des données, et découle de la projection du schéma logique.

La définition successive de ces trois schémas constitue le cycle de vie des données.

2.5.2.2 Pour la gestion des données

Les concepts présentés jusqu'ici traitaient des données en terme de représentation statique. Intéressons nous maintenant au processus de manipulation des données, et des problèmes qui en découlent :

- L'intégrité :

La base de données doit représenter le monde réel d'une façon aussi fidèle que possible (c'est la fidélité telle que nous la définissions en 1.). Il faut donc protéger la base de toute tentative de manipulation qui conduirait à un état

identifié (par mise en évidence d'incohérences) comme ne reflétant pas la réalité, c'est à dire maintenir son intégrité.

Nous verrons, en abordant le paragraphe concernant la qualité des informations supportées par les bases de données quelles peuvent être ces manipulations coupables, et jusqu'à quel point il est possible et nécessaire de les gérer.

- Les droits d'accès :

Une base de données peut représenter un univers de discours dont une partie peut être réservée à certains acteurs. Les systèmes de gestion de base de données permettent ce genre de fractionnement des droits d'accès aux informations, en lecture ou en écriture.

De plus, quand des informations peuvent être partagées, des mécanismes de réservations sont mis en place, pour régler le problème de ces accès concurrents : un ensemble d'informations peut être momentanément réservé à un utilisateur et son accès interdit aux autres, qui sont mis en attente.

- Les mesures de sauvegarde et de restauration :

Pour exprimer des opérations complexes, c'est à dire composées de plusieurs manipulations élémentaires, des SGBD autorisent la définition de transaction : elles ont pour première propriété de restaurer un état stable (à priori cohérent) de la base en cas de mauvaise exécution de la série de manipulations. Leur seconde caractéristique, en cas de succès dans l'exécution de l'opération, est de ne vérifier que la cohérence de l'état final, et pas celle des états intermédiaires.

2.5.3 Formalismes des SGBD

2.5.3.1 Tant au niveau conceptuel qu'au niveau logique, la description des structures de données se fait dans un langage "de définition", et l'accès à leurs valeurs, par l'activation de requêtes écrites dans le langage "d'interrogation" (ou de manipulation) des données.

Ces langages sont fortement liés aux concepts de modélisation adoptés par le SGBD. On peut définir, grossièrement, trois types d'organisation des schémas conceptuels :

- hiérarchique, où les entités sont reliées de manière arborescente (et donc où chacune d'elles ne peut avoir qu'au plus un père),

- réseau, où les entités sont reliées par des relations binaires de cardinalité (1,n),

- relationnel, où toute structure de données est assimilée à une *relation* finie, c'est à dire à un sous ensemble du produit cartésien d'une liste finie de domaines (et où un domaine est un ensemble défini de valeurs).

Nous ne discuterons pas des avantages et inconvénients de ces différentes organisations : de telles études existent depuis longtemps déja, et on se réfèrera à [ULLM80], ou plus récemment [LATO86] et [MIRA86a] pour une évaluation comparée de ces approches. Nous remarquerons simplement que l'approche relationnelle est supportée par des langages d'interrogation qui implémentent des concepts mathématiquement définis, comme l'algèbre ou le calcul relationnel, ce qui en fait une approche bien formalisée.

2.5.4 Modélisation de connaissances par les BD

- Les faits

L'approche base de données, on l'a vu, distingue plusieurs niveaux de modélisation. Toutefois, quel que soit ce niveau, les outils de modélisation proposés considèrent primordialement les faits simples, et les structures statiques.

De ce fait, la gestion des faits est extrêmement développée et riche : les langages d'accès relationnels sont le plus souvent complets (relativement à l'algèbre relationnelle), quelquefois récursifs, et très efficaces. De plus, la quantité de données manipulables est très grande.

Toutefois, il faut remarquer ici que les objets structurés dans les bases de données ont des valeurs d'attributs monovaluées : les BD ne savent pas gérer les multiples.

- Les connaissances

Ce sont surtout les connaissances structurelles ou statiques que l'on sait modéliser en utilisant les langages de définition de données :

Le **modèle hiérarchique** modélise bien entendu aisément les structures arborescentes : nos travaux menés dans le cadre du projet national Concerto [ALLE86] ont montré que le Génie Logiciel était un grand consommateur d'arbres.

Le **modèle relationnel** structure des attributs valuables et le concept d'index permet d'identifier des

objets.

Les concepts du **modèle entité-relation** [CHEN76] permettent la définition de structures fixes (des "entités"), mais aussi de lien entre ces objets (les "relations"). Les cardinalités associées aux relations apportent la possibilité d'exprimer des contraintes comme :

"Monsieur GrandBlond ne peut porter qu'entre 0 et 2 chaussures"

ou, plus "Génie Logiciel" :

"Un projet ne peut utiliser au plus qu'une méthode"

mais :

"Une méthode peut être utilisée par plusieurs projets"

Le concept de "relation" induit aussi celui de "référence" : seules deux (ou plus) instances d'entités qui existent en tant que telles peuvent être en relation.

D'autres travaux de modélisation, auquels nous avons humblement contribué [ALLE85], ont tenté d'approfondir le concept de relation, en augmentant et précisant leur sémantique, par l'intégration de concepts mathématiques comme les relations d'ordre, injections, bijections, et autres. Ils permettent l'expression de contraintes fortes :

"un réseau de taches (dans un planning), ne peut contenir de boucle".

Un autre concept de [ALLE85], la relation "canal", permet d'intégrer des concepts hiérarchiques au modèle

entité-relation :

Soient quatre types d'entités : projet, activité, cycle_de_vie, activité.

Soient quatre relations :

- projet-tache, cycle_de_vie-activité (inclusions),
- projet-cycle_de_vie (qui exprime le fait qu'on choisit un cycle de vie dans un projet),
- tache-activité (qui exprime le fait qu'une tache correspond à un certain type d'activité).

On dira que tache-activité est "canalisé" par projet-cycle_de_vie pour exprimer le fait qu'une tache ne peut correspondre qu'à une activité du cycle_de_vie choisi pour le projet auquel elle appartient [DELA86].

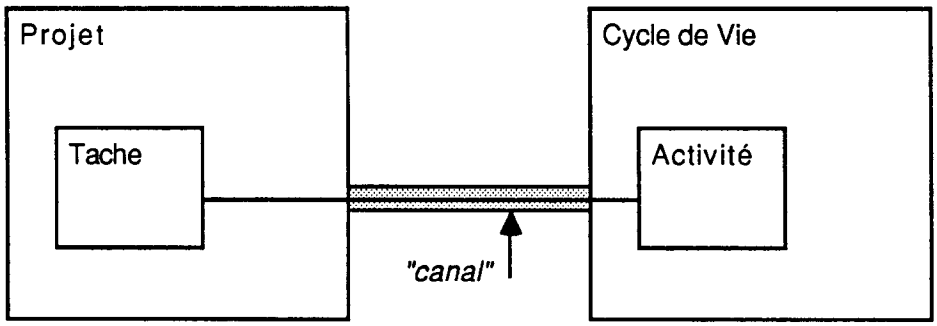


fig. 7 : Patre, sous vue d'un modèle

Enfin, des concepts comme l'**aggrégation** et la **généralisation** ont été intégrés au modèle relationnel par Smith [SMIT78], et au modèle entité-relation [LEDI86].

En conclusion à ce paragraphe, nous noterons que les connaissances modélisables par les BD sont des connaissances structurelles statiques et des contraintes sur les manipulations de faits.

- Les métaconnaissances

Les droits d'accès, mécanismes de maintien de la cohérence, ou de sauvegarde, sont des concepts bien précis, qui correspondent à des métaconnaissances, et ont été présentés plus hauts. Là encore, on notera qu'elles contraignent les manipulations (droits d'accès), ou permettent de conserver des états stables.

- Informations et bases d'informations généralisées

Les SGBD sont adaptés pour la gestion d'un très grand nombre d'informations stockées sur disque. L'utilisation de ce type de support physique a suscité l'idée de lier BD et bases de documents : les années 1982 - 1985 ont été l'époque de projets de bases de données généralisées et multimédia, comme BIG [CHRI83], BAOU [CHRI85], BDM [CHRI85] et TIGRE [BOGO83].

Ces projets ont tenté de modéliser la sémantique contenue dans des documents ou des images en étendant des modèles de données traditionnels, notamment le relationnel. Depuis, les travaux de ce type ont été moins nombreux.

2.5.5 Qualité des Informations dans une BD

2.5.5.1 Véracité et négation

Dans une base de données, c'est la présence d'un fait qui témoigne de sa véracité. Toutefois, certains formalismes autorisent la définition de valeurs par défauts, et en conséquence, permettent d'exprimer des "vraisemblances" comme discutées plus haut (cf. paragraphe correspondant dans le chapitre consacré aux RCO).

De plus, la vérification des contraintes définies lors de la modélisation permet d'affirmer que les faits existants sont globalement cohérents par rapport au modèle.

Un point relatif à l'existence des faits pose toutefois toujours problème : celui du traitement sémantique des valeurs nulles. Il a été largement abordé (par exemple par [VASS79]), et des réponses ont été proposées, pour essayer de donner un sens formel aux valeurs d'attributs non précisées : doit on les considérer comme inconnues, ou susceptibles d'être incohérentes, ou bien traduisent elles le fait que l'attribut considéré est non significatif ? Un choix simple peut être fait, à priori, pour l'une ou l'autre des interprétations, mais qui pose problème lors des manipulations : comment, par exemple, effectuer une jointure sur des valeurs nulles ?

2.5.5.2 Notion de Temps en BD

Prendre en compte, au niveau des bases de données, le concept de temps, revient à considérer divers états des objets modélisés. Il s'agit là d'un problème de gestion des multiples (valeurs multiples pour un même objet), chose non naturelle pour les SGBD. Cet aspect de la modélisation a donc fait l'objet de nombreuses études que nous avons considéré.

Tout d'abord, nous remarquerons qu'il y a deux philosophies de la prise en compte du temps dans un modèle de données :

.la considération du temps comme attribut intrinsèque d'un objet lors de son observation :

"La tache t21 a été créée le 12/12/87"

se définira, dans un modèle relationnel, par la

table: tache(nom, date_de_création), soit, pour l'instance, tache(t21,12/12/87).

.la considération du temps comme une dimension à part entière, orthogonale à toute modélisation, et qui associe à chaque changement d'état, une estampille. C'est cette deuxième approche qui semble la plus enrichissante, et nous décrivons ci après les différentes appréhensions que l'on peut avoir de la dimension temps :

- Le concept d'événement :

Il est défini par Bubenko comme étant "une observation ou une décision survenant dans le système au temps t, du commencement ou de la fin d'une occurrence d'une association". [BUBE77]. Les bases de données, qui sont une image de la réalité, ont à prendre en compte un double problème :

. l'intégration de l'instant de la manipulation de la base, qui est, pour le SGBD, l'image de l'événement,

. l'instant réel de l'événement, qui est généralement antérieur à son signalement au système. Ceci pose le problème, non résolu d'ailleurs, d'une intégration tardive du passé dans le contexte courant d'un système, notamment en ce qui concerne les problèmes d'intégrité relatifs aux manipulations éffectuées entre l'instant réel et l'instant d'intégration de l'observation dans le système.

- Le processus temporel :

L'identification de l'enchainement de différents états d'un système conduit à une classification des dépendances entre événements passés, présents et futurs. Selon Tapiero [TAPI78], on peut mettre en évidence 4 types de systèmes :

. Ceux dans lesquels il n'existe aucune corrélation entre états passées, présents et futurs, que l'on qualifie de "myopiques".

. Les systèmes qui synthétisent, dans leur dernier état, toute leur évolution antérieure.

. Ceux dont l'état courant est fonction linéaire d'un certain nombre d'états antérieurs identifiés.

. Les systèmes dont l'état courant est une fonction pondérée (par un coéfficient d'"importance") d'un certain nombre d'états antérieurs du système.

. Enfin, ceux dans lesquels une extrapolation dans le futur est rendue possible par un corrélation, fonction des états passés, entre l'état courant et les états futurs.

- Les contraintes temporelles :

Une première formalisation de ces concepts peut passer par la mise en évidence de contraintes sur l'état d'une base de données. Carmo et Sernadas [CARM87] présentent plusieurs niveaux de contraintes temporelles :

Des contraintes d'intégrité simples :

. les contraintes statiques, que l'on analysera dans un autre paragraphe, et qui limitent les modes d'instanciation des objets du système,

. les contraintes de transition, qui limitent les modifications des objets du système, et portent sur des états d'objets :

"Une tache ne peut être mise dans un état 'terminé' que si son état est soit 'commencé', soit 'suspendu'",

Des contraintes événementielles :

. les contraintes qui séquencent les événements, en définissant des règles de successions.

Trivialement :

"Modifier un objet" ne peut se faire qu'après "Créer un objet" ou "Modifier un objet", et ne peut précéder que "Modifier un objet" ou "Détruire un objet".

- Les requêtes relatives au temps :

On peut discerner plusieurs types de besoins d'informations temporelles :

.La connaissance d'un état en fonction de l'instant,

.La connaissance des périodes de validité d'objets dans un état donné,

.La connaissance d'une série d'états pour une période de temps donné,

.La connaissance d'une valeur fonction d'une série d'états pour une période donnée (moyennes, sommations, ...).

Différents formalismes (TRM, TSQL, TQUEL, ...) existent, qui implémentent ces aspects, et dont on trouvera une description dans [SNOD84][KLOP83][NAVA87].

2.5.5.3 Hypothèses dans les BD

La gestion de données hypothétiques n'est pas, *a priori*, l'objectif d'un SGBD, qui a plutôt pour vocation d'être l'image du monde réel. Pourtant, nous l'avons justifié dans notre chapitre introductif, un certain nombre d'univers se préoccupent autant de leur projection dans le futur que de leur état actuel.

Les travaux concernant les systèmes d'aide à la décision traitent, au contraire, de ces problèmes de

simulation : on a déja parlé des travaux de Doyle [DOYL79] et de son TMS (Truth Maintenance System). A notre connaissance, aucun système de ce type n'a fait l'objet de tentative d'implémentation sur un SGBD. Pourtant, dans une optique "base de données généralisée", cette approche peut devenir intéressante.

2.5.5.4 Synthèse

En résumé, nous dirons que la fidélité des systèmes de gestion de bases de données est limitée par la faible quantité de concepts qu'ils sont capables de mettre en oeuvre : la granularité y est restreinte à trois niveaux : le "schéma" abstrait, des tuples que l'on peut identifier par des clés, et des atomes représentant les éléments d'un tuple.

Le contrôle de cohérence est pauvre, limité aux domaines de valeurs et aux types, ainsi qu'à l'unicité des clés pour certains SGBD.

La rémanence d'un SGBD dépendra bien sûr de l'implémentation des concepts présentés en 2.5.3.2.

Les SGBD sont en général incapables de gérer l'exception : leur sélectivité est nulle.

2.5.6 Conclusion et remarques

Les bases de données mettent en oeuvre des techniques et des méthodes que l'on maîtrise désormais raisonnablement. Nous avons pu voir que leurs concepts de base peuvent faire l'objet d'extensions. Nous retiendrons donc que les SGBD procurent un environnement basique intéressant pour une structure de système d'information étendu.

Nous verrons, dans un prochain paragraphe concernant les techniques hybrides, que des travaux visant à l'ennrichissement de leurs possibilités sont en cours et semblent prometteurs.

2.6 Systèmes hybrides

La naissance de systèmes hybrides, prenant leurs sources à la fois dans l'utilisation de techniques orientées objet, logiques, ou issues des bases de données, a eu pour motivation la volonté d'utiliser les avantages de chacunes d'entre elles.

Schématiquement :

- les représentations objets sont riches pour l'expression de la sémantique et la gestion des situations exceptionnelles,

- les outils issus de la logique apportent une formalisation mathématique intéressante des connaissances,

- les bases de données ont pour vocation de gérer de grandes quantité de faits, chose que ne savent pas bien faire les autres techniques citées.

Nous allons nous intéresser ici à quelques unes de ces approches hybrides :

- O.O. et SGBD,
- EAR et logique des prédicats du premier ordre : F1 [CAZI84a],
- relationnel et logique,
- arborescences, EAR et logique : PATRE [ALLE85].

Nous mettrons l'accent sur les plus values apportées par l'hybridation, et nous renvoyons le lecteur aux paragraphes spécifiques des composants de l'hybridation pour plus de détails.

2.6.1 O.O. et SGBD relationnels

2.6.1.1 Les définitions et concepts élémentaires

Dans cette approche, une entité (si l'on se réfère aux définitions EAR [CHEN76]) est doublée d'une classe d'objets (cf. paragraphe traitant des RC) qui incluera ses attributs et des pointeurs vers les autres entités avec lesquelles elle est en relation.

Une instance (un t-uple relationnel) sera une instance d'objet. Quelques équivalences peuvent être présentées :

entité	**classe**
attribut	**slot**
relation	**slot**
t-uple	**instance**

Une classe contiendra aussi la définition des mécanismes d'instanciation, et, par exemple, les contraintes sur les manipulations :

```
(classe employé
    (attribut1 nom)
    (attribut2 age)

    - méthode1:CréerInstance (nom age)
    (si age > 16 alors nouveau (employé nom age))
    - …
)
```

décrit, dans un langage pseudo smalltalk, une classe

"employé" qui ne pourra qu'avoir des instances dont l'attribut "age" sera supérieur à seize.

On remarquera, dans ce domaine, les travaux de [ANDR86], ainsi que ceux de XINHUO [LICH87].

2.6.1.2 Modélisation de connaissances - Avantages du couplage

Cette approche profite :

- des capacités du SGBD à gérer de grandes quantités de faits,

- de la facilité de prototypage en O.O.,

- des possibilités de représentation de la sémantique en O. O. (cf. paragraphe traitant des RCO),

Elle permet de regrouper traitements et définitions statiques au sein de mêmes unités syntaxiques.

2.6.1.3 Qualité des Informations et Hypothèses dans le contexte OO - BD

La représentation de données hypothétiques en milieu O.O. seul a été traitée ailleurs.

Le couplage présenté ici a pour conséquence de gérer deux représentations, en partie redondantes, d'un même monde réel. Il permet d'utiliser le monde O.O. comme "mémoire à court terme", où des incohérences par rapport au réel peuvent exister. On peut donc y manipuler des données hypothètiques.

La base de données joue au contraire un rôle de "mémoire à long terme", et donc de représentation de

référence, dans laquelle on ne répercute que des informations certaines.

Le monde Orienté Objet, lui, supporte une représentation des connaissances beaucoup plus modulaire que l'environnement SGBD, dans la mesure où ses concepts permettent de regrouper aspects statiques et dynamiques d'une même entité, l'objet. Ce type de complémentarité est lui aussi séduisant.

2.6.1.4 Conclusion et remarques

Cette première approche hybride prouve l'intérêt d'utiliser plusieurs formalismes pour tirer le meilleur avantage de chacun d'eux.

La complexification, dûe à l'utilisation simultanée de représentations redondantes est ici positive, puisqu'elle permet d'introduire ces concepts de mémoires à long et court terme.

Certains [ANDR86] ont voulu voir dans cette double représentation deux niveaux, l'un conceptuel, l'autre logique: Nous préfèrerons dire que ce couplage permet une représentation plus riche de la sémantique que ne l'autorisent les formalismes implémentés par les SGBD.

2.6.2 SGBD relationnels et logique

2.6.2.1 Les définitions et concepts élémentaires

Bases de données et systèmes logiques convergent depuis plusieurs années. Gallaire & al [GALL84] ont donné en 1984 un excellent état de l'art en la matière. Depuis, ces travaux continuent [LPDB86][GARD87] dans plusieurs laboratoires.

Les fondements de ce rapprochement sont non seulement théoriques et mathématiques (il existe en effet une isomorphie entre le concept de relation et celui de prédicat), mais aussi pratiques : la logique peut compléter les services proposés par une base de données traditionnelle (non déductive).

L'ensemble des concepts de base qu'utilisent cette approche est fondé sur l'union de ceux des deux éléments du couplage.

L'application de prédicats du monde logique à une base de données sera soit **déductive** (c'est l'utilisation de règles d'inférence pour l'explicitation de faits contenus implicitements dans le modèle), soit **contraignante** (c'est l'utilisation de prédicats comme contraintes d'intégrités).

On peut définir quelques principes fondamentaux :

- la bijection entre identifiant et individu (qui établit qu'un individu réel ne peut être représenté dans le modèle que d'une manière unique),

- la fermeture de l'ensemble des individus modélisés (qui présuppose que le monde réel ne contient pas d'autres individus que ceux identifiés dans la base de faits),

- la fermeture de l'univers situationnel modélisé (qui dit que la non explicitation d'une relation entre individus revient à sa non existence, et non pas à l'ignorance de son occurrence).

L'apport de la logique permet de manipuler des assertions sur des variables :

- la modélisation d'individus au travers de leur existence dans l'expression d'une contrainte (et non plus seulement de leur présence dans la base de faits),

- la modélisation implicite (déductible) de certaines situations,

- la modélisation d'alternatives (A implique (B ou C)).

Nous allons dans la suite montrer comment ces concepts interviennent et peuvent être utilisés.

2.6.2.2 Modélisation de connaissances par un couplage BD-logique. Deux approches [FROS85].

- L'union d'une structure relationnelle complète et d'une Théorie (au sens de la logique).

On considére ici la base de données comme une représentation complète (principe de fermeture de l'ensemble modélisé) de l'univers du discours.

On lie la base de données à des concepts logiques en disant que le schéma de la base (le modèle représenté par les relations, dans un SGBD relationnel) est aussi l'ensemble des prédicats d'une théorie.

Maintenir la cohérence du modèle revient, dans cette approche, à s'assurer que toute manipulation des faits est cohérente avec les formules de la théorie. Toute règle logique est ici considérée comme une contrainte, jamais comme une règle de déduction.

- La "Théorie" seule.

C'est cette fois l'ensemble des faits qui constitue l'ensemble des axiomes d'une théorie qui, a elle seule, donne donc une description complète de l'univers du discours considéré.

Une différence majeure réside ici dans le fait qu'un prédicat est considéré comme un moyen de déduction :

p(x) -> q(x) se traduit par :

"si p(x) alors q(x)" est vrai, et non pas comme une contrainte (c'est à dire l'impossibilité de vérifier sa négation) :

"il existe x tel que p(x) et non q(x)" n'est pas un théorème (dans la théorie considérée).

Une autre difficulté réside dans l'expression de la négation : en logique, et donc dans cette approche, le principe de fermeture du monde modélisé n'est pas explicite, et la non démonstration d'un fait n'équivaut pas à sa négation, mais à son ignorance.

Pour tirer partie de la négation d'un fait, il est

nécessaire :

- soit de définir explicitement sa négation,
- soit d'expliciter un axiome complémentaire, qui exprime que, sauf pour une liste de valeurs données in extenso, un fait est nié (ces deux solutions deviennent rapidement lourdes à manipuler),
- soit d'appliquer le principe de la négation par échec (comme en Prolog), dans le cas de clauses de Horn.

La première des deux approches présentées est applicable dans le cas de bases de données non déductives, mais dont on veut maintenir la cohérence. La seconde apporte cette possibilité.

Cependant, toutes deux sont insatisfaisantes si on veut à l'occasion manipuler des données incertaines (et donc ne pas utiliser systématiquement le principe de négation par l'échec).

La seconde approche ne permet pas non plus d'expliciter simplement des contraintes relatives à l'évolution de la base de faits (des contraintes opérationnelles).

Il serait donc enrichissant d'améliorer la définition des contraintes, et de mettre en place des mécanismes pour la manipulation de données incertaines (qui sont bien utiles pour les outils d'aide à la décision, outils importants dans le domaine du Génie Logiciel qui nous préoccupe).

2.6.2.3 Notion de temps et hypothèses

Aucun travail propre à cette approche couplée n'a été inventorié, et l'on ne pourrait dire ici rien de plus que ce

qui a été dit à propos de chacun des deux éléments du couplage.

2.6.2.4 Conclusion et remarques

L'intérêt d'une telle approche est indéniable, d'un point de vue conceptuel, sous réserve des extensions souhaitées ci dessus.

Des choix d'implémentations différents sont possibles:

- extensions de systèmes, logique vers base de données, et vice versa,

- création de systèmes intégrant l'union des fonctionnalités,

- couplage physique de systèmes existants.

Nous verrons, dans un prochain paragraphe, un exemple (F1) d'une réalisation du second type.

2.6.3 F1

2.6.3.1 Les définitions et concepts élémentaires

F1 est un formalisme d'expression de la connaissance écrit en Lisp et Prolisp, une version Lisp du langage Prolog.

Les concepts manipulés par F1 sont les entités et relations, au sens EAR du terme [CHEN76], ainsi que des lois et des actions. On peut considérer F1 comme une approche dont le fondement théorique est dans la lignée de la précédente, mais l'implémentation quelque peu particulière.

Les instances d'entités et relations constituent une base de faits.

. Une loi est l'expression d'une contrainte, d'une règle vérifiée par une entité ou une relation, ou bien encore de valeurs déductibles des valeurs courantes d'entités et relations, mais que l'on ne souhaite pas mémoriser.

Une action est une opération construite, un ensemble de manipulations de la base de faits, qui présente les caractéristiques suivantes :

- elle n'est exécutée qu'après vérification de préconditions,

- elle n'est validée que sous réserve de la satisfaction de postconditions,

- elle est exprimée dans un langage implémentant les quantificateurs "Il existe" et "Quelquesoit", ainsi que les opérateurs logiques "et" et "ou".

Cette séparation contraintes/déductions est fort

appréciable et va dans le sens des évolutions souhaitées dans le paragraphe où nous avons traité des couplages SGBD et logique.

Enfin, d'autres concepts EAR, comme les cardinalités, les clés, et les domaines complexes, sont utilisables.

De plus, un lien sémantique "sorte-de" permet de définir des sous types d'entité qui héritent des propriétés du type père, et des contraintes statiques peuvent être définies, qui restreignent la définition des types :

```
(type individu (structure (chaine nom)
                          (entier age))
               (clé nom))

(type employé (sorte-de individu)
              (structure (entier salaire))
              (respecte ((qqsoit employé *x)
                         (eval (< (age *x) 60) t))))
```

de [CAZI84a]

2.6.3.2 Modélisation de connaissances

Plusieurs utilisations de F1 [CAZI84b] [CAZI85] ont montré la richesse de ce formalisme pour l'expression des connaissances, grâce à son adaptation fonctionnelle à la manipulation des faits et à la gestion des règles et contraintes.

On remarquera, dans F1, la modélisation séparée du statique et du dynamique (la définition des entités, relations, et lois d'une part, celle d'actions d'autre part), qui est propre aux approches BD.

2.6.3.3 Notion de temps et hypothèses en F1

F1, en ce sens très proche des SGBD traditionnels, ne prend pas en compte la gestion temporelle des faits.

2.6.3.4 Conclusion et remarques

Nous retiendrons de ce formalisme l'approche consistant à intégrer des concepts entité-relation et de la logique, leurs structures des informations, mais aussi leurs langages de manipulation.

Nous remarquerons aussi que, baigné dans un environnement Lisp, F1 profite de toutes les possibilités de manipulation symbolique qu'offre ce langage fonctionnel, et qui viennent augmenter sa capacité à modéliser les comportements dynamiques.

Toutefois, nous soulignerons l'incapacité matérielle de F1 à manipuler des grandes quantités de faits, et ce par manque d'efficacité.

2.6.4 Combinaison de représentations hiérarchiques, EAR et logique : PATRE [ALLE85].

2.6.4.1 Les définitions et concepts élémentaires

L'approche PATRE formalise une notion qui jusqu'ici avait été simplement implicite, à savoir l'existence de plusieurs "niveaux" de représentation de la connaissance :

Elle s'appuie sur la constatation suivante :

- l'être humain manipule volontiers (et c'est caractéristique en Génie Logiciel), des objets stucturés de manière arborescente,

- le langage d'expression des connaissances liées à un univers donné doit être aussi riche que possible, quitte à être redondant (c'est un point de vue qui se discute, mais qui se défend),

- le support "interne" des connaissances doit être aussi formel que possible, afin de permettre un contrôle maximal de leur intégrité.

PATRE inclue donc trois niveaux de représentations des connaissances :

- un niveau externe, fondé sur un outil de modélisation hiérarchique étendu (la représentation interne, RI, de Concerto [LANG85]), qui s'utilise en considérant un modèle de connaissance comme la grammaire d'un langage :

Un objet, de type abstrait "exemple", est défini par des expressions de la grammaire, soit, exprimé en pseudo Bacchus Naur Form :

<exemple> ::= EXEMPLE <nom-exemple> ATTRIBUTS <liste_atts>
COMPOSé_DE <liste-type-comp> ;

<liste-type-comp> ::= <type-composant> <liste-type-comp> |
<type-composant> | nil ;

<type-composant> ::= <type> <liste-objet> ;

<liste-objet> ::= <objet> <liste-objet> | <objet> | nil ;

Une structure d'objets ainsi bâtie permet un premier contrôle, syntaxique, dans la manipulation des relations entre objets, et autorise des manipulations d'aggrégats d'objets.

La vision hiérarchique des objets induit une navigation "naturelle" dans le modèle ainsi constitué.

- un niveau conceptuel, décrit dans le formalisme PATRE, dont les caractéristiques sont présentées dans un paragraphe précédent. Ses points forts sont, on le rappelle, son pouvoir d'expression génériques des relations (ordres, "canaux", …) et des lois F1, ses opérateurs de description ensemblistes [ALLE85], son formalisme graphique.

- un niveau interne, qui s'appuie sur le formalisme F1 présenté par ailleurs.

2.6.4.2 Modélisation de connaissances

L'utilisation simultanée de ces divers types de représentation passe par une spécialisation quant à l'utilisation de chacun d'entre eux.

- F1 (au niveau interne), assure la gestion et le support des faits et connaissances qui sont certains et qui tiennent lieu de référence,

- PATRE est un outil pour la conception d'un modèle, qui autorise des définitions génériques de lois et actions F1,

- la RI est utilisée comme support de la connaissance "à court terme", comme une sorte de brouillon d'un ensemble d'informations qui ne serons références qu'une fois validées.

2.6.4.3 Notion de temps et hypothèses

Le temps n'était pas une préoccupation lors des travaux qui ont mené à la réalisation de PATRE.

La gestion de données hypothétiques n'était pas non plus traitée en tant que telle, mais l'utilisation des objets supportés par la RI, au niveau externe, et leur utilisation comme brouillon, constituait un premier pas dans cette voie.

2.6.4.4 Conclusion et remarques

PATRE a montré que l'on pouvait faire vivre simultanément trois formalismes différents, tous supportés, il est vrai, par un même langage interprété, Lisp. Son apport a été réel tant sur le plan langage et logiciel (les trois formalismes ont été dotés d'un langage de manipulation), que sur le plan méthode (l'expression graphique du niveau conceptuel), et nous essaierons d'en tirer partie dans la suite des présents travaux.

2.7 Evaluation - Choix

2.7.1 Rappel du cadre de l'étude

Nous souhaitons proposer une structure d'accueil de systèmes d'information, plus particulièrement destinée aux SI d'Ateliers de Gestion de Projets Logiciels (sources des exemples cités).

Il est apparu que les connaissances que nous avions à manipuler étaient à la fois :

- des faits (1) :

"Le projet IMPW est en retard",
"La tache WP9 commence le 23 Juin 1988", ...

- des règles de manipulation partagées (contraintes, processus automatiques) (2) :

"Le chef d'un projet doit être choisi avant la méthode de spécification",
"Une équipe ne doit pas dépasser 5 personnes sans être hiérarchisée",
"Dès qu'un employé est malade, sa contribution journalière à toutes les taches auxquelles il est affecté doit être annulée", ...

- des documents (3) :

"Le plan qualité du projet IMPW",
"Le rapport d'avancement de la tache WP9", ...

De plus, nous souhaitons que les SI supportés par notre système soient accessibles par des outils informatiques de types variés :

- des outils algorithmiques traditionnels(4) :

PERT,
Modèles d'estimations algorithmiques (COCOMO, …), …

- des outils de l'intelligence artificielle, écrits dans des formalismes O.O.(5) ou logiques (6) :

Gestionnaire de ressources (O.O.),
Analyseur de risques (O.O.),
Suivi de projet (logique), …

De l'expression de ces contraintes, nous pouvons conclure que nous devons gérer des informations de "petite" taille, élémentaires (7), qui puissent être ensuite structurées par différentes méthodes (sous forme d'objets et d'assertions logiques, par exemple).

D'autre part, un SI supporté par notre structure d'accueil doit satisfaire aux caractéristiques générales d'un système d'information, comme décrites en 0., et que nous résumons ici :

- possibilité de mémoriser les connaissances partagées par les utilisateurs du système, à savoir, pour nous, les différents outils de gestion de projet (8a) : Le concept de "tache" est le même pour l'outil d'estimation, le PERT et l'outil de suivi de projet, même si chacun d'eux les manipulent dans un formalisme différent et à sa manière,

- possibilité de "naviguer" dans le monde des connaissances (8b), et ce de manière aussi complète que

possible : d'une part, notre domaine d'application possède un très grand nombre d'"indicateurs" (consommation de ressources, qualité d'un module, …) dont les liens logiques sont complexes, et d'autre part, il inclut de nombreuses structures hiérarchiques (arbre des taches, structure arborescente d'un logiciel, méthodes de gestion basées sur les arbres [ALLE86b], …),

- capacité à maintenir la cohérence de ces connaissances (9),

- flexibilité, adaptabilité à des contextes d'organisations différentes (10) : La gestion de projets logiciels chef un grand constructeur, dans une SSII, ou bien encore dans un institut de recherche, n'obéit pas toujours aux mêmes règles … [CONC86][CASE87].

2.7.2 Des choix de structure

Comme nous l'avons remarqué en 1., il est des connaissances, de type "dynamique" ou procédural, qui découlent d'un savoir faire. Dans notre approche SI + outils, nous décidons donc de faire en sorte que ces procédures, généralement propres à une activité de la gestion de projet, soient supportées par les outils eux-même (a). Le SI ne supportera que les procédures générales, propres au domaine modélisé dans son ensemble, et ce sous la forme de démons, inspirés des attachements procéduraux présentés en 2. (b).

Les connaissances statiques, plus souvent générales et partagées, seront prises en considération et décrites dans le SI (c), ainsi que l'histoire de ces connaissances (d).

Les métaconnaissances seront partagées entre outils et SI. On cherche à trouver des mécanismes génériques pouvant à

leur implémentation dans ce SI (e).

La gestion de documents doit être prise en compte par le SI, qui doit fournir aux outils des possibilités pour les manipuler (f).

2.7.3 Les choix de techniques

Notre analyse du chapitre 2. a mis en évidence les différentes caractérisques des techniques de représentation des connaissances utilisables.

Il s'avère que les SGBD apportent une réponse très satisfaisante aux besoins (7), (8a), (8b) et (10). De plus, ils offrent l'avantage d'avoir suscité de nombreuses études visant à les étendre pour satisfaire à (3), (9), et à améliorer (10).

De plus, ils sont souvent déja interfacés vers des langages traditionnels qui pourront permettre d'implémenter certaines connaissances procédurales et répondre à (4).

Un SGBD sera donc la pierre angulaire de notre système. Nous le choisirons de type relationnel, qui a l'avantage de s'appuyer sur une théorie formelle.

Pour lui permettre d'approcher les objectifs (2) et (3), il nous apparait essentiel de le "coupler", comme on l'a vu en 2.6, avec une technique beaucoup plus orientée "intelligence artificielle". On peut se reporter [NARA87] pour une très bonne grille d'évaluation de ces techniques.

Il ressort des paragraphes précédents que les adjonctions à notre SGBD devront résoudre les points (b), (c),

(d), (e) et (f).

- Nous laissons à part, pour l'instant, le point (f) qu'aucune technique précédemment étudiée ne satisfait directement.

A - Les points (b) et (c) sont pris en compte de manière très convenable par les outils de représentation basés sur la logique. Le point (e) ne l'est que moyennement [NARA87], mais peut l'être par la conception d'une couche "méta" [LPDB86]. Ces outils ont été l'objet d'études visant à prendre en compte les problèmes d'historique, comme on l'a vu en 2., et augmentent, de ce fait, les chances d'atteindre (d).

De plus, ils permettent des extensions (relativement) faciles de la base de connaissance qu'il supportent (et satisfont en cela à (10)) et s'appuient sur une théorie formelle, comparable à la théorie relationnelle. Ils peuvent ainsi augmenter les capacités d'un SI à atteindre (9), par l'apport de la logique à la vérification de la cohérence, et peuvent augmenter les possibilités de navigation (8b).

B - Les RCO, elles, ont des apports positifs, notamment pour (b) (attachements procéduraux), mais présentent le handicap d'être moins facilement extensibles, moins flexibles, et portent en eux une certaine redondance avec un SGBD (pour ce qui est de la mémorisation des données), tout en n'offrant pas leur capacité de recherche d'information (et donc n'apportant rien à (8b)). Ils ne sont pas très satisfaisant, non plus, du point de vue de (7) :

Sans doute intéressante si elle est utilisée de manière autonome et indépendante, cette technique n'apporte qu'une faible plus value à une structure d'accueil de système d'information.

Au contraire, on a vu en 2.6 qu'elle pouvait apporter beaucoup au niveau d'un interface utilisateur, et donc d'un outil de navigation au sein d'un SI (5).

C - Les RS sont apparus comme d'excellents moyens d'expression de la connaissance, mais on a vu qu'ils souffraient d'un manque d'implémentation manifeste.

D - Les systèmes de production, si on les considère comme des implémentations particulières de systèmes logiques orientés vers la déduction et l'expression des raisonnements, apportent un plus surtout pour (6), et donc en dehors, en fait, de notre structure d'accueil.

Nous décidons ici de choisir une implémentation de système logique (nous verrons plus loin que ce sera un Prolog) **pour être couplé à notre SGBD.**

Revenons à (f). N'ayant pas de solution satisfaisante à priori, il nous parait approprié de nous retourner vers des systèmes aux fonctionnalités étendues mais manipulant des concepts simples, comme un système de gestion de fichier.

Nous procèderons donc au couplage d'un SGF aux éléments que nous venons de choisir. Nous repréciserons ce choix en 3..

Enfin, les points (5) et (6), dont la satisfaction

n'appartient pas à notre structure d'accueil, doivent cependant être favorisés par celle-ci.

Aussi porterons-nous un soin particulier à proposer des interfaces vers des représentations objets et des systèmes logiques à règles de production. Ce dernier point sera bien sûr facilité par le premier choix de couplage entre Prolog et un SGBD.

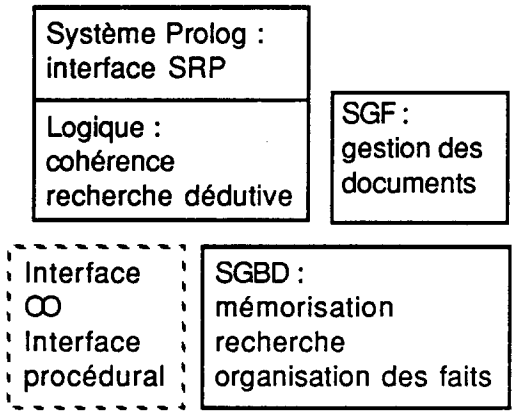


fig. 8 : Les "briques" de base de notre système

3. Etat de l'art complémentaire

Comme suite au choix que nous venons d'effectuer, nous allons étudier les possibilités techniques d'implémentation de systèmes couplés SGBDR et Prolog, puis les différentes méthodes d'acquisition des connaissances qu'ils sont capables de supporter.

3.1 Techniques d'implémentations de systèmes SGBD/Prolog

Trois approches architecturales sont envisageables :

- extensions de SGDB par des procédures déductives et de vérifications de contraintes,

- extensions de langages logiques à la gestion de grandes quantités de données (et donc à la gestion de fichiers),

- couplage, à proprement parlé, d'un SGBD et d'un langage logique.

3.1.1 Extensions de SGBD

Cette approche, si l'on veut qu'elle soit efficace, suppose d'avoir accès au code source du système que l'on souhaite étendre. Elle est menée notamment dans plusieurs centre universitaires qui ont auparavant développé des SGBD traditionnels :

On citera par exemple les projets CAMPUS [MIRA86b], SABRE [GARD86], TIGRE [LOPE83], qui proposent d'ailleurs d'autres extensions (vers des langages fonctionnels par

exemple).

De tels systèmes conservent LDD et LMD comparables au SQL bien connu, mais étendus à d'autres fonctionnalités.

3.1.2 Extensions des langages logiques

On notera l'existence de prologs étendus à la gestion de fichiers sur disque (comme Mu-Prolog [NAIS83], ou bien encore KUL Prolog [VENK85]).

On remarquera aussi que d'autres prologs (comme FOLL-Prolog [DONZ83]) implémentent, pour le stockage des faits, des mécanismes de hashcode qui augmentent la capacité de gestion de faits en mémoire centrale.

Nous nous rangerons derrière G. Gardarin [GARD87] pour dire que cette voie est sans doute difficile, au vu de la quantité de travail nécessaire à la réalisation d'un SGBD.

3.1.3 Les couplages

3.1.3.1 Différentes techniques

Nous voulons parler ici des systèmes qui considèrent les deux composants, l'interpréteur Prolog et le SGBD, comme deux "boites noires". L'interface se situe au niveau du langage de manipulation du SGBD, et les instances de relations mémorisées par celui-ci sont interprétées comme des règles élémentaires par le moteur d'inférence.

Différentes "sous" approches sont encore possibles :

Considérant un certain nombre de prédicats faisant

appel à des données contenues par le SGBD, on peut :

- soit "calculer" la meilleure question globale à poser au SGBD, et intégrer son résultat au monde Prolog (c'est l'approche "question complète" [BOCC87]) (sol1),

- soit faire autant de requêtes qu'il en apparait logiquement dans le programme Prolog (c'est l'approche "question/sous-question" [BOCC87]). Cette deuxième solution peut encore être traitée de deux manières différentes :

.par pré-compilation, en lançant une pré-évaluation du but initial, et en chargeant au fur et à mesure les résultats des requêtes au SGBD, puis en relançant le but sur ce nouvel environnement (sol2),

.par interprétation, en générant, au fur et à mesure qu'elles apparaissent dans la résolution, les requêtes au SGBD, et en récupérant leurs résultats comme autant de choix pour backtracker (sol3).

3.1.3.2 Comparaison

Chacune de ces techniques possède avantages et inconvénients. Quelques études ont comparé différentes approches, notamment dans [CORN87] qui est très claire et synthétique, mais aussi dans [CUPP86] et [BOCC87], et nous en présentons ici une synthèse enrichie de notre propre expérimentation :

(sol1) optimise la quantité d'informations échangées entre le moteur logique et le SGBD (un seul appel, et des résultats limités, en nombre, au résulat souhaité).

(sol1) génère des requêtes complexes, que l'on fera

améliorer par le SGBD. Cette solution est donc intéressante pour un couplage avec un SGBD doté d'un bon optimiseur de requêtes.

Mais (sol1) ne satisfait pas à tous les traitements possibles en Prolog, notamment dans la prise en compte des prédicats évaluables et du "cut".

(sol2) génère plus de requêtes, mais dans une proportion non exagérée, et qui sont plus simples.

Mais (sol2) charge en mémoire un grand nombre de résultats qui ne sont pas toujours utilisés, d'où des risques de dépassements de taille mémoire. De plus, le "cut" n'est pas non plus pris en compte, ni la récursivité terminale que sait pourtant traiter Prolog (même si Prolog ne traite pas tous les cas de récursivité).

(sol3) génère des requêtes simples, qui sont activées progressivement, ce qui élimine certaines requêtes inutiles.

(sol3) permet de traiter (ou plutôt de laisser traiter par Prolog) les prédicats évaluables, le "cut" (moyennant une petite manipulation …), et la récursivité. Ce point est très important, car il garanti le même comportement d'un programme Prolog, qu'il soit éxécuté sur sa propre base de faits ou sur une base de données équivalente.

Mais (sol3) génère beaucoup de requêtes (et donc perd du temps en communication inter processus), regénère parfois les mêmes requêtes (encore que cela puisse être amélioré). (sol3) nécessite donc d'utiliser un SGBD qui répond vite à des questions simples.

Nous tiendrons bien sûr compte de ces remarques lors

de nos choix d'implémentation, qui nécessitent, au préalable, une étude sommaire de nos deux composants, PrologII [GIAN85], et Informix ESQL.

3.1.4 Prolog et Prolog II

3.1.4.1 Introduction

Nous ne présenterons pas ici l'ensemble des principes de ce langage logique. On pourra pour cela se référer à [GIAN85], et, pour ce qui concerne les règles de programmation en Prolog, [COLM83].

Nous nous contenterons de montrer quelques uns des aspects intéressants et spécifiques de Prolog, et de son implémentation Prolog II, que nous utiliserons dans la suite de nos travaux.

Prolog est un langage de la logique des prédicats du prmier ordre restreinte aux clauses de Horn. pour complément d'information sur la logique, on se reportera au paragraphe traitant des logiques comme moyens de représentation de la connaissance.

3.1.4.2 Représentation des connaissances en Prolog

Les faits sont, en Prolog, représentés par des assertions, des règles sans second membre. En syntaxe "de Marseille" (par opposition à la syntaxe d'Edimbourg ...), on écrira par exemple :

tache(codage_traducteur, 871220, 880220) —> ;

le fait que la tache "codage_traducteur" commence le

On pourra aussi exprimer des connaissances abstraites, comme le concept de "tache", sous la forme :

```
attribut(tache, nom_tache, 1) —> ;
attribut(tache, date_de_debut, 2) —> ;
attribut(tache, date_de_fin, 3) —> ;
```

qui décrit la relation modèle du fait exprimé au dessus.

3.1.4.3 Mécanismes de recherche en Prolog

Une telle représentation des connaissances, et le mécanisme d'unification implémenté par le moteur d'inférence de Prolog, permet de lancer des buts avec des variables libres dont l'unification conduit à l'expression des réponses à la question correspondant au but. Par exemple :

```
tache(x, y, 880220) ;
```

retournera les couples (x,y) (i.e. nom_tache et dtae_de_debut) des "tâches" se terminant le 20/02/88.

(Nous ferons ici une parenthèse pour parler du langage Prolog III, en préparation à l'Université de Luminy [COLM87]. La puissance de ce nouveau langage, notamment pour l'expression de contraintes sur les valeurs de variables, permettra de poser des questions du type : Quelles sont les taches qui se terminent AVANT le 20/02/88 ?

En syntaxe Prolog III approximée :

```
tache(x,y,z) {z < 880220} ;)
```

3.1.4.4 Quelques mécanismes de Prolog

La récursivité est partiellement traitée par Prolog : son mécanisme de résolution (en profondeur d'abord) oblige à prendre soin à la façon dont on écrit un prédicat récursif. Dans notre couplage, nous veillerons à ne pas modifier le mécanisme de Prolog, même si il n'est pas entièrement satisfaisant.

Prolog dispose d'un prédicat coupe-choix, le "cut", qui permet d'écrire des prédicats partiellement déterministes.

Prolog dispose aussi de prédicats "évaluables", qui ont des effets de bord sur la base de connaissance.

Dans ces deux cas, nous veillerons à ne pas modifier le comportement des programmes Prolog qui seront couplés à une base de données par notre mécanisme.

3.1.4.5 Particularités de Prolog II : le "freeze", le "dif" et les "arbres infinis"

Le prédicat "freeze" permet de reporter l'effacement d'un but tant que l'on n'a pas restreint les valeurs possibles de l'une des variables qu'il contient :

freeze(v, terme(v)) ;

sera interprété de la manière suivante :

- si la variable "v" est libre, "freeze(v, terme(v))" est effacé, et l'effacement de "terme(v)" est en suspend jusqu'à ce que "v" soit unifiée

- si "v" est liée, alors "terme(v)" s'interprète normalement.

Le prédicat "dif" est une contrainte imposée à deux termes, pour qu'ils soient toujours impossibles à unifier : la vérification de cette contrainte est activée lors de l'"effacement" du prédicat, et reste valable pour la suite de la résolution. Si, par la suite, l'effacement d'un sous-but conduit à l'unification des deux termes, alors il y backtrack et echec sur ce sous-but.

Prolog II, au contraire des autres Prologs, manipule des "arbres infinis" (qui sont des arbres dont un sous arbre pointe vers une racine qui est en fait d'un niveau supérieur, i.e. aussi un "sur" arbre).

3.1.4.6 La communication entre Prolog et d'autres langages

Quand on parle de couplage, on comprendra que ce point est crucial. Il existe différents mécanismes d'interfaces entre langage Prolog et C. On retiendra que Prolog II est capable d'échanger (et ceci dans les deux sens), des informations avec un processus issu d'un programme C. Nous reviendrons sur cette problématique lorsque nous montrerons nos choix d'implémentation.

3.1.5 Informix ESQL

3.1.5.1 Les principes

Nous ne nous étendrons pas ici sur les langages d'accès aux bases de données, comme SQL.

Nous nous contenterons de souligner certains aspects et concepts du langage d'interrogation d'Informix.

- Le concept de vue : une vue regroupe le résultat d'opérations relationnelles (jointures, produits cartésiens, projections, ...) sous la forme d'une nouvelle relation, manipulable en tant que telle, et constamment maintenue (en fonction des variations des relations par lesquelles elle est paramétrée.

- Le concept de relation temporaire permet de définir des "relations variables locales".

- Le mécanisme de "verrouillage" momentané des tables relationnelles autorise la protection de données partagées.

- La notion de transaction donne la possibilité de retrouver un état jugé stable après une série de manipulations.

- Le concept d'index unique permet de définir des attributs comme formant une "clé", tout en augmentant l'efficacité du système dans ses réponses aux questions qui lui sont posées.

3.1.5.2 Les communications entre Informix et C

Ce SGBD possède un langage d'accès "noyable" dans des programmes C.

L'interpréteur du langage est implémenté par un

processus indépendant, qui gère les multiaccès aux bases de données.

3.1.5.3 L'optimisation dans Informix

Informix dispose d'un langage efficace pour ce qui concerne les requêtes simples, mais l'est beaucoup moins dans le cas de requêtes plus complexes. On se souvient que, dans la problématique qu'est le choix d'une méthode d'implémentation du mécanisme de couplage, cette particularité a une grande importance.

3.2 Outils pour la gestion de documents

3.2.1 Introduction

Nous ne traiterons pas ici des systèmes dont l'objectif est de structurer, décomposer, recomposer des documents. Nous essayerons plutôt, dans nos travaux, de nous intéresser à la gestion de documents "complets" (non structurés, mais liés aux connaissances que nous saurons modéliser et qui seront supportées par notre système couplé.

3.2.2 Des systèmes existants

Nous avons vu, dans le chapitre 2, que pas, ou peu de supports de la connaissance prennent en compte les documents. Les seuls travaux conséquents dans ce domaine ont été les études de bases de données généralisées.

Les outils informatiques qui servent couramment à la gestion de documents sont en réalité les systèmes de gestion de fichiers (SGF) et leurs dérivés.

- Le SGF d'Unix [FONT84]

C'est à priori un bon exemple de système de gestion de fichiers, dotés d'outils de bases (structuration arborescente, liens), mais aussi de mécanismes plus complexes, comme SCCS pour la gestion de versions.

- Le Système de Gestion d'Objets de PCTE (SGO)

présentation complète. Le SGO est un système de gestion de fichiers élaboré, basé sur une structuration EAR des fichiers (donc beaucoup plus riche de sémantique que la structuration arborescente d'Unix), qui permet de définir véritablement une "base de fichiers", par analogie aux "bases de données". Lui aussi est doté d'un système de gestion de version.

3.2.3 Conclusion

Notre souci est de permettre la représentation de documents relativement aux autres connaissances.

En ce sens, le SGO est "trop riche", trop orienté SGBD, et, d'une certaine manière, pourrait devenir redondant du SGBD : à notre connaissance d'ailleurs, des travaux ont étés récemment entrepris dans le projet PACT [THOM87], par les centres de recherche d'Olivetti et Systems and Management, à Pise (I). Ils visent à "coupler" le SGO à un langage logique, pour obtenir un système logique de gestion de documents.

Notre choix se portera donc sur l'utilisation du SGF d'Unix pour supporter les documents que notre système permettra de manipuler. Bien sûr, nous mettrons en place une infrastructure qui couplera, en fait, à la fois SGBD, langage logique, et SGF.

3.3 Méthodes d'acquisition de connaissances pour un système couplé relationnel et logique

3.3.1 Introduction

Si aujourd'hui de nombreux travaux sont réalisés qui visent à intégrer SGBD et langages logiques, peu traitent du problème de l'acquisition et la formalisation des connaissances avec pour objectif une implémentation sur un système couplé.

En effet, les possibilités nouvelles introduites, les concepts induits par le couplage (contraintes, démons,...) ou par d'autres de nos préoccupations (gestion du temps, des hypothèses) sont souvent difficilement exploitables. Il manque au concepteur d'un SI les outils, ou la méthode, pour analyser un univers de discours et dégager de celui-ci ses contraintes spécifiques, ses comportements systématiques.

3.3.2 Les briques élémentaires

Les techniques BD sont munies de méthodes de modélisation : les grandes classiques comme Merise [TARD83], Remora [ROLL82] [FOUC82], NIAM (Nijssen), et beaucoup d'autres [GL86] proposent des démarches d'analyse pour mettre en évidence les concepts d'un univers de discours.

Pour simplifier la présentation de ces méthodes, on dira qu'elles sont de quatre types :

d'abord à analyser le processus de traitement, d'en extraire les "données", disons les "connaissances statiques", et de les organiser. Des exemples : SADT (ou $IDEF_0$) [ROSS77], JSD [JACK83].

- approche "formelle" : les méthodes formelles proposent des langages de spécifications, des opérateurs prédéfinis qui doivent servir à décrire l'univers considéré. On citera les travaux menés dans le project SPRAC (Onera - CERT), mais aussi à la faculté d'Orsay.

- approche "données" : la méthode préconise la mise en évidence des connaissances statiques, par des biais divers (nous y reviendrons), puis la modélisation des traitements que l'on va leur appliquer. Un exemple : Merise.

- approche "objet" : ici, on recherche d'abord à isoler des entités, à la fois du point de vue des connaissances statiques que dynamique. Des exemples : "B.J. Cox method" [COX83], "Booch's design method" [BOOC82].

Les deux dernières approches se caractérisent par une phase "d'analyse de texte" qui consiste à extraire de l'énoncé du contenu d'un univers de discours verbes (qui décrivent les actions) et substantifs et pronoms (qui décrivent les connaissances statiques).

Cette phase est ensuite suivi d'une "normalisation" (on parlera des formes normales d'EAR [CHEN76] ou des principes O.O. selon B.J. Cox) qui permet de regrouper de manière adéquate les connaissances pertinentes.

L'approche "formelle" est très orientée vers un processus de génération automatique de codes informatiques : son objectif est donc assez différent de notre domaine, et nous ne faisons que la citer ici, comme une "autre" démarche

de description d'un univers de discours.

L'approche "processus" peut se passer d'une première phase, car la démarche suivie est arborescente et semble correspondre au processus intuitif d'analyse propre à l'humain.

3.3.3 Conclusion

Nous essayerons d'ailleurs de voir si cette approche "processus" ne pourrait pas être préalable à l'approche "données", en remplaçant la phase "d'analyse de texte", ou "d'interview", par une décomposition arborescente des processus, dans le cadre d'une méthode d'analyse adéquate à un outil de représentation logique et base de données.

4.

Les concepts retenus pour notre structure d'accueil

4.1 Préambule à la dernière partie

Comme nous l'avons décidé en 2.7, nous allons étendre les concepts d'un SGBD relationnel en nous appuyant sur les fonctionnalités d'un langage logique, d'un système de gestion de fichier, et du SGBD lui même.

4.1.1 Du chapitre 4.

Dans un premier temps, nous allons, en nous appuyant sur la littérature existante, sur l'état de l'art qui précède concernant les spécifications d'une SA, et sur nos connaissances expérimentales des ateliers de Génie Logiciel (GL), sources de nos exemples, dégager et définir précisémment les concepts que pourra implémenter le système, et les fonctionnalités dont il sera doté : Nous y affinons, en termes de définition et de gestion, les concepts de faits, de connaissances, de documents, dont nous avons déja parlé de manière générale.

Ensuite, nous nous posons la question de savoir comment il est envisageable d'"acquérir", de spécifier les informations que le système peut supporter, et nous apportons un embryon de réponse, ou plutôt une orientation dans la recherche d'une solution, à laquelle nous avons eu l'occasion de réfléchir.

Nous profitons aussi de ce préambule pour préciser que, pour des raisons "historiques", nous désignerons par la suite ce système "Impish", pour "IMP Information System Host", où "IMP" est lui même l'acronyme de "Integrated Management Process workbench", du nom du projet ESPRIT P938 qui a contribué à financer les travaux relatés ici.

4.1.2 Du chapitre 5.

Il sera consacré à la présentation des langages d'accès à notre système : Nous y verrons comment les concepts sont définis et comment leurs manipulations ont pu être implémentées sur un système hybride SGBDR/Prolog/SGF.

4.1.3 Du chapitre 6.

Il présentera l'architecture logicielle du système, les différents interfaces qui permettent d'y accèder, et détaillera les mécanismes internes de base qui assurent l'interprétation des commandes d'Impish, en insistant sur les interfaces qui unissent SGBD, moteur logique, et SGF.

4.2 La représentation de concepts relatifs aux faits

4.2.1 Les faits

Ce concept a été présenté en 1. Nous retiendrons comme définition d'un fait, dans Impish, le résultat véridique d'une observation d'un détail "simple" du monde réel.

En raison des choix technologiques réalisés en 3., nous regroupons et structurons les faits sous la forme de tuples, au sein de relations. Nous dégageons aussi les deux concepts de la modélisation EAR, celui d'entité (dont une occurrence peut exister en tant que telle), et de relation (dont nous dirons si il faut l'entendre au sens "relationnel" ou "EAR"). Toutefois, nous n'en faisons pas les structures de données spécifiques du langage d'accès à Impish, puisque nous conservons une structuration relationnelle, parce que cette solution nous parait plus souple et moins contraignante pour l'expression de faits de faible granularité. Pourtant, nous allons préciser plus bas des types de contraintes (de référence, de clé, de domaines de valeur, …) avec lesquels il est nécessaire de compléter le modèle relationnel pour atteindre la puissance du modèle EAR.

4.2.2 La véracité de faits

Nous choisissons de définir la véracité d'un fait par son existence dans la base d'information : un fait inexistant sera donc interprétable comme "faux" par un utilisateur.

4.2.3 Les faits présents

Tout fait est considéré comme vrai à un instant si il existe dans une base que nous qualifierons de "courante" ou de "présente" à cet instant :

"Le projet IMPW est dans un état 'commencé'".

4.2.4 Les faits historiques

Nous avons aussi vu, dans notre étude du chapitre 2. consacrée aux SGBD et aux systèmes logiques, quelles étaient les extensions possibles à ces systèmes pour leur permettre de gérer des faits historiques passés. Impish implémente l'un d'entre eux, qui est basé sur le concept de "version de fait", que l'on peut définir comme un couple ("fait", intervalle de validité de l'observation). Ce mécanisme peut s'appliquer, pour un tuple donné, lors de la manipulation d'une valeur d'attribut (qui est un "générateur d'historique", ou "historical data", ou HD) : modifier la durée (HD) d'une occurrence de la relation "tache" provoque l'historisation de la totalité du tuple qui la définit :

"Le 15/11/87, IMPW devait durer 36 mois",
"Depuis le 01/03/88, sa durée prévue est de 40 mois".

4.2.5 Les faits futurs

Définis comme en 1.2.2, une information "future" peut être donnée à Impish sous la forme d'un couple (commande de manipulation, date de prise d'effet).

4.2.6 Les faits hypothétiques

Le besoin de se placer dans des situations irréelles mais possibles est fréquent pour des systèmes d'aide à la décision. Or, de nombreux outils, notamment en Gestion de Projets Logiciel, ou en Gestion de la Qualité des Logiciels, sont de ce type. Cette nécessité a été évoquée à plusieurs reprises par [FRAN87] et [FAIR87].

4.2.6.1 Les "contextes" d'hypothèses

Les faits manipulés par un contexte hypothétique (cf. définition en 1.2.3) sont marqués comme appartenant à l'hypothèse associée à une liste nommée d'opérations :

"insertion de 'ressource Bosco affectée à Drive 50% et à Esprit à 50%'",

(alors que le fait réel est : 'ressource Bosco affectée à 100% sur le projet Esprit').

4.2.6.2 Les faits hypothétiques déduits

Lorsqu'un outil se place dans une hypothèse, il travaille :

- d'abord sur les faits appartenant à cette hypothèse : Le plan de charge hypothétique de Bosco sera établi en fonction de son affectation "hypothétique".

- ensuite sur les faits du contexte courant (la base "réelle") dont la clé, aussi bien pour les entités que les relations représentées, n'existent pas dans l'hypothèse choisie : La qualification de Bosco sera trouvée dans la base réelle.

Quand un outil qui travaille dans une hypothèse donnée veut modifier un fait du contexte courant, une copie de celui-ci est "remontée" dans l'hypothèse, où elle est ensuite modifiée.

Une hypothèse qui est vérifiée peut être répercutée dans la base et tous les faits qui la composent sont alors introduits dans le contexte courant.

Le SGBD relationnel est, et on verra comment, le support de ces différents types de faits.

4.3 Les concepts liés aux connaissances

4.3.0 Connaissances, contraintes, démons et raisonnements

Des types de connaissances explicitées en 1., nous choisissons de faire supporter par Impish :

- les connaissances qui sont des contraintes portant sur les faits,

- celles qui sont des manipulations systématiques des faits, que nous appelons "démons".

Au contraire, les raisonnements spécifiques d'un savoir faire ou d'une activité sont supportés par des outils, et sont donc externes au système d'information.

4.3.1 Les contraintes

Le concept de contrainte a été défini en 1.. Son introduction dans les SGBD est une marque de l'évolution des bases de données vers un contrôle d'intégrité sémantique des données beaucoup plus performant que celui assuré traditionnellement.

La formalisation des contraintes, traitée par plusieurs travaux de recherche, notamment [NGUY85], que nous avons pu étudier, nous a permis de sélectionner certains types de contraintes, que nous implantons, en utilisant l'interpréteur Prolog, sur Impish.

4.3.1.1 Les contraintes d'unicité de clé

Pour identifier des objets, en Génie Logiciel comme dans d'autres domaines, il est utile de pouvoir les désigner de manière unique. Pour cela, on définit le concept de clé, associé à l'unicité d'un n-uplet de valeurs d'attributs (de colonnes de tables relationnelles) : Deux taches, instances de l'entité "Tache", ne peuvent avoir le même "nom-de-tache", si on ne veut pas les confondre.

Les clés sont utilisables pour nommer des entités, mais aussi pour désigner de manière unique les occurrences de relations (au sens EAR) entre deux entités.

4.3.1.2 Les contraintes de références

Un contrainte de référence, qui porte sur une colonne de table relationnelle, restreint son domaine de valeurs aux occurrences d'une conjonction d'autres colonnes : Tous les tuples d'une table "travaille-sur" doivent être des couples "(nom-d-employé, nom-de-tache)" élément du produit cartésien des valeurs d'attributs "nom-de-famille" de la table "Employé" et des valeurs d'attribut "nom-de-tache" de la table "Tache".

4.3.1.3 Les contraintes de dépendence fonctionnelle

Une valeur d'attribut peu dépendre fonctionnellement d'un certain nombre de valeurs d'autres attributs : une telle situation résulte parfois d'une "mauvaise" modélisation du réel (par exemple, il existe une fonction simple entre la longueur d'un intervalle et les valeurs de ses deux bornes, mais on trouve souvent, dans des modèles, les trois attributs présents, par exemple "(début-de-tache, fin-de-tache, durée)",…), mais une telle redondance est parfois nécessaire.

Une dépendance induit des conditions préalables à la manipulation de valeurs des attributs contraints.

Les trois types de contraintes ont des définitions statiques, mais ils induisent des contraintes opérationnelles, c'est à dire vérifiées au moment des manipulations de faits.

4.3.1.4 Les contraintes opérationnelles

On peut aussi vouloir explicitement conditionner une manipulation précise (insertion, suppression, modification) par une vérification préalable ou *a posteriori* :

Le fait que, par exemple, une ressource ne peut être affectée qu'à une date postérieure à la date de sa dernière journée de travail prévue, et est représentable par une contrainte opérationnelle qui doit être vérifiée avant l'insertion d'un tuple qui représente une affectation.

4.3.1.5 Les domaines de valeurs

Les domaines de valeurs ont pour intérêt de réduire les possibilités d'instanciation des faits, et visent à augmenter le contrôle d'intégrité des données de la base.

Nous voulons être capable de gérer des domaines définis en extension, comme en intention, et vérifier l'appartenance d'une valeur à un domaine lors de l'insertion ou la modification d'un fait : La "taille" d'une "Equipe" ne doit pas dépasser 5 personnes, ou bien encore : L'"état" d'une "Tache" est dans la liste "(projeté, commencé, qualifié, terminé)".

4.3.2 Les démons et les raisonnements sur la base de faits

L'introduction des démons modifie sensiblement la philosophie d'une base de donnée, puisque ces démons lui confèrent un caractère dynamique qu'elle n'a pas naturellement.

En Génie Logiciel, les procédés systématiques sont appréciés. Pour donner simplement un exemple, les méthodes qui permettent de définir l'ordonnancement des taches d'un planning peuvent être automatiques (c'est la cas dans l'atelier que nous réalisons), et l'introduction d'un nouveau composant logiciel à réaliser (appelons-le toto) peut provoquer la définition de toutes les taches qui conduiront à cette réalisation. Un démon peut, automatiquement, être activé pour "croiser" un cycle de vie (par exemple l'enchainement des phases "spécification", "conception", "codage", "intégration"), avec "toto", pour générer et introduire dans la base les nouvelles taches, successives, que sont :

"spécification_de_toto",
"conception_de_toto", ….

4.4 Les concepts liés aux documents

Un document est, plus qu'une information, un support de l'information. Son objectif est de faciliter la communication entre les différents acteurs d'un processus, à savoir, pour nous, celui de la production de logiciels.

4.4.1 Les documents générés par des outils de gestion de projets

4.4.1.1 Documents et versions de documents

Un document généré par un outil est considéré comme un produit fini de l'atelier : inséré dans le système d'information, en fonction de paramètres que l'on va définir, il ne pourra plus être modifié sans qu'une nouvelle version, en fait, soit générée (et gérée par le système) : C'est, par exemple, la description, au 10 juin 1986, du composant C31 idendifié par "Desc.10.06.86.C31".

4.4.1.2 Les "faits clés" d'un document

L'objet d'un document peut être varié : synthèses, rapports, descriptions (sous forme textuelle ou codée) d'un certain nombre d'autres informations, plus "concentrées" et atomiques, il se doit d'être en harmonie avec elles. Ainsi, le document qui décrit textuellement une "tache", à l'instant T, et qui est généré par un outil de planification, s'il contient des indications sur la date de fin de cette tache qui ne sont pas celles prévues, à ce même instant, pour la tache, doit être considéré comme partiellement obsolète.

nous permettons la définition d'un document produit par un outil (un "tool doc.") comme paramétrable par des "faits clés", faits contenus par la base définie plus haut dans ce chapitre.

Cette paramétrisation est graduable : la cohérence avec un ou plusieurs faits peut être de type "fort", "moyen", "faible" : La description "Desc.10.06.86.C31" dépend fortement de "01/06/86", date de la dernière modification du code de C31.

De plus, ces "faits" clés peuvent servir de mots clés dans la recherche des documents.

4.4.1.3 Combinaison d'incohérences

L'accumulation des incohérences doit aussi être prise en compte : un document qui dépend "faiblement" de N faits, et pour lequel on détecte N/10 incohérences "faibles", n'aura pas la même crédibilité qu'un document pour lequel on en dénombre 9N/10.

Pour document donné, il est doit donc être possible d'obtenir une loi entre sa crédibilité et une combinaison du nombre d'incohérences fortes, moyennes, faibles entre le document et la base au moment de l'utilisation du document.

4.4.2 Les documents descriptifs

Le concept de document descriptif est associé à celui d'un attribut dont le type serait un peu particulier, et pourrait être stocké sous une forme quelconque (comme un long texte ASCII décrivant les objectifs d'une tache, une photographie numérisée pour accompagner un CV, ...).

Les documents sont supportés par le SGF, et les mécanismes de gestion des "faits clés" implémentés sur le SGBD relationnel.

4.5 Les manipulations élémentaires

Nous reviendrons dans le prochain chapitre sur le détail des opérations simples (insertions, suppressions, modifications) réalisables sur et avec les concepts choisis et définis ci-dessus.

Toutefois, nous présentons ici, dans leurs grandes lignes, les objectifs que nous nous sommes fixés, à propos de la manipulation :

- Des tables relationnelles

Dans Impish comme dans un SGBDR, elles supportent les faits courants. Toutefois, elles doivent aussi supporter les faits historiques, et les faits hypothétiques.

Leur création, suppression, modification, sont du domaine du Langage de Définition de Données (LDD) et ne sont pas accessibles aux outils applicatifs, mais seulement à l'"administrateur" d'Impish.

- Des faits courants

Leur création, suppression, modification, sont du domaine du Langage de Manipulation de Données (LMD) et sont accessibles aux outils.

Toutefois, elles sont soumises à la vérification des contraintes, et peuvent générer l'activation des démons.

La recherche de faits courant comme elle est possible

dans un SGBD traditionnel est enrichie par des possibilités de filtrage logique et récursif sur lequel nous reviendrons longuement.

- Des faits historiques

Leur création, suppression, modification appartiennent au LMD.

Elles héritent des caractéristiques des opérations sur les faits courants, mais gèrent les "versions" de faits.

Leur recherche est paramétrable par le temps : on peut retrouver telle valeur de fait à telle date, par exemple.

La manipulation des faits hypothétiques a déja été introduite.

- Des contextes

Un contexte peut être validé (on vérifie sa cohérenc eavec la base courante) et "réalisé" (on le substitue au contexte réel).

- Des contraintes et démons

Leur création, suppression, modification, sont du domaine du Langage de Définition de Données (LDD) et ne sont pas accessibles aux outils, mais seulement à un administrateur.

Les contraintes sont implicitement utilisées au moment de la manipulation des faits, que ce soit par le LMD ou

par un démon qui a été activé .

- Par les démons

En fin de session (utilisation de la base par un outil), les démons activables sont exécutés, et la base peut ainsi être modifiée.

Les démons sont considérés comme des "commandes système", comme les mécanismes d'intégration des faits futurs.

- Des types de documents

Leur création, suppression, modification, sont du domaine du Langage de Définition de Données (LDD) et ne sont pas accessibles aux outils.

- Des documents descriptifs (ou référencés)

Le type "document référencé" étant assimilable à un nouveau type de données, la manipulation de ces documents est, à quelques exceptions près que nous préciserons, assujettie à des règles semblables à celles qui s'appliquent à la manipulation des faits.

La manipulation des "tool" documents a aussi déja été abordée.

4.6 Un embryon de méthode de spécification des informations supportées par Impish

4.6.1 La problématique

La question que se pose un concepteur de système d'information est la suivante : Comment vais-je trouver les informations que j'ai besoin de gérer, et avec quel concept de mon outil de représentation (pour nous, Impish) vais-je faire supporter telle ou telle connaissance ?

Des méthodes existent, qui, de l'entretien avec l'utilisateur du futur système d'information, à l'analyse de documents (cf. 3.), aident à la mise en évidence de certains concepts comme des entités et des relations, pour prendre un exemple.

Lorsque, comme avec Impish, l'on dispose de concepts plus variés, se pose le problème de définir une méthode qui permette de tirer partie, au mieux, de cette richesse supplémentaire dont on peut disposer.

Notre approche, qui est beaucoup plus un point de départ qu'une recherche achevée, s'appuie sur la thèse suivante : un système d'information pour un atelier, quel qu'il soit, gère les données que manipulent les outils de cet atelier; définissons le processus qu'implémentent ces outils, et les données qui y apparaîtront seront celles qu'il faudra modéliser.

Premier point donc : partir de la modélisation des traitements pour obtenir celle des données.

Deuxième point : extraire des données ce qui est

fait, contrainte, document, donnée historique, et, au niveau des traitements, ce qui est comportement systématique et besoin de simulation.

4.6.2 Description d'un processus d'analyse basé sur l'analyse des traitements

Un exemple : $IDEF_0$

Cette méthode permet de définir les spécifications graphiques d'un processus, en autorisant l'expression des différentes actions (mais sans prendre en compte leur synchronisation) et des échanges de données entre ces actions (*actigram*).

L'analyse est hiérarchique : la méthode consiste à découvrir les actions d'un certain niveau, puis de les détailler en les décomposant, ….

Du flux de données échangées entre ces actions, qui sont de type "données en entrée", "données en sortie", "données de controle", on peut extraire un dictionnaire de données, et une représentation montrant, pour chacune d'entre elles, les actions qui les manipulent (*datagram*).

4.6.3 Observation des données

De ce dictionnaire, enfin, on peut penser extraire les concepts que devra supporter le système d'information concerné : Des entités, des relations, mais, on peut aussi l'espérer, des contraintes (notamment au niveau des données de contrôle), des domaines de valeurs (en analysant les données consommées et modifiées par une action), les besoins en

supports de type "documents", ….

- Des données de contrôle qui sont utilisées par une action pour réaliser un test, mais qui ne sont pas modifiées, on peut déduire des informations sur :

.leur domaine de valeur,

.des liens, notamment de dépendance, avec d'autres données.

- De certaines des données en entrées et sortie, on peut connaître les différents états qui leur sont éventuellement associés et comprendre que leur histoire est importante:

Par exemple, quand une donnée "passe" par une action de validation, elle y entre "existante" et en sort "valide". On peut alors pressentir que l'histoire de ces faits a une importance pour le processus, surtout si ils sont utilisés, ensuite, comme données de contrôle.

On peut aussi leur associer un type, et, pour certaines d'entres elles, qui apparaissent le plus souvent dans les niveaux supérieurs de la hiérarchie des actions, se rendre compte que ce sont des données complexes de type "document".

- Dans la sémantique des verbes qui définissent les actions, enfin, on peut mettre en évidences des démons et le besoin de gérer certains faits comme des hypothèses :

Des actions aux noms évocateurs, comme "simuler", "lancer", …, on pourra tirer quelques explications à ce sujet.

4.6.4 Conclusion

Notre étude n'avait pas pour but de définir une méthode de spécification adaptée aux concepts d'Impish : il peut s'agir là plutôt du sujet d'une autre thèse, dans la suite de celle-ci, et beaucoup plus orientée "méthodes de conception".

Nous tenions toutefois à dire qu'une telle approche, brièvement survolée ici, pouvait apporter un plus dans la modélisation, notamment des contraintes, problème qui reste, à ce jour, l'un des plus cruciaux dans le domaine de la spécification des systèmes d'information.

Il nous a enfin semblé important de nous préoccuper quelque peu de ces problèmes : Atteindre, comme nous nous le sommes fixé, un objectif de synthèse sur les stuctures d'accueil de SI, passe par une telle démarche.

Mais, après cette courte escapade dans le monde des méthodes, revenons à notre outil de représentation, et parlons des moyens qu'il offre pour supporter ces différents concepts présentés ci-avant.

5. Les langages de définition et de manipulation de données d'Impish

5.1 Introduction

Nous allons montrer ici les fonctionnalités d'Impish à travers ses différentes commandes.

Chaque fois que cela présentera un intérêt particulier, nous expliciterons comment, en termes fonctionnels, Impish utilise ses trois composants. Toutefois, nous ne nous étendrons pas sur les interfaces entre ces composants qui seront détaillés en 6..

Pour structurer cette présentation, nous avons choisi de décrire :

- d'abord le langage de définition (à savoir, les commandes d'insertion et de suppression des bases Impish, des tables, des types de documents, des contraintes et des démons, utilisables par un "administrateur"),

- ensuite, le langage de manipulation (à savoir, les commandes d'insertion, de modification et de suppression des faits et documents, utilisables par des outils applicatifs).

5.2 Le LD d'Impish

Le Langage de Définition d'Impish est interprété en PrologII, et active, via des interfaces qui seront définis en 6., à la fois des commandes ESQL et du SGF d'Unix. Il gère aussi de l'information en PrologII dans un sous monde de notre interpréteur "model".

5.2.1 Définition des bases

Une base Impish est constituée de :

* une base ESQL,
* une hiérachie de répertoires et fichiers Unix,
* un monde PrologII.

- Le *create database nom-base* d'Impish envoie à ESQL :

un *create database nom-base,*

et un :

create table doc (doctype char(128), docname char(128), facttype char(128), factkey char(128), level char(8), flag char(1))

qui servira à gérer les "faits clés" des documents produits par les outils : on y conserve le type du document (doctype), son nom (docname), le nom de la table à laquelle appartient le fait (facttype), la concaténation de sa clé (factkey), le type de dépendance, faible, moyen ou fort (level), la marque d'une manipulation éventuelle du fait

(flag).

- Par appel de Prolog à Unix, il crée la hiérarchie suivante, sous le répertoire repéré par une variable $IS :

$IS/nom-base/REF_DOC (pour les documents descriptifs)

$IS/nom-base/TOOL_DOC (pour les documents produits)

$IS/nom-base/TEMP (pour stocker les documents en cas de transactions)

$IS/nom-base/USER (où l'on copie les documents sélectionnés)

$IS/nom-base/FUTURE (pour les documents descriptifs de faits futurs) .

$IS/nom-base/OLD (pour les documents descriptifs de faits obsolètes)

- Enfin, il initialise un sous monde "model" de l'environnement PrologII dans lequel est écrit Impish, en y insérant le prédicat **program-number(0) ->;** (initialisation des programmes de contraintes), en assignant la valeur nom-base à une variable globale PrologII **current-base**, et en créant un fichier "$IS/nom-base/nom-base.pro".

- La sauvegarde du modèle y sera faite lors d'un *close database* Impish, en même temps que l'envoie d'un *close database* à ESQL.

- Une base peut, comme en ESQL, être détruite par un *drop database*, qui sous entends dans Impish la destruction de la hiérarchie $IS/nom-base et celle du monde "model".

5.2.2 Définition des tables

Une table Impish est représentée par une table relationnelle, un ensemble de répertoires Unix, un certains nombres de prédicats dans le sous-monde "model" de l'environnement PrologII d'Impish.

5.2.2.1 La création des tables relationnelles :

- création d'une table "nom-table_sys" ESQL composée des colonnes définies dans le *create table nom-table...* Impish et des colonnes systèmes *status*, *false*, *sdv*, et *edv* dont on verra l'utilité dans le paragraphe consacré au LM d'Impish,

- création d'une vue ESQL "nom-table", sur cette table système, composée des seules colonnes définies conceptuellement, et ne regroupant que les tuples pour lesquels *status* vaut "c", pour "courant" (cf. gestion des faits historiques),

- par souci d'améliorer les performances, création d'un index ESQL, sur la colonne *status* de "nom-table_sys".

5.2.2.2 La création des répertoires Unix

Si'il apparait dans la liste des colonnes des "liens" (dont le type est *link*), alors pour chacune d'elle est créé le répertoire "$IS/nom-base/REF_DOC/nom-table/nom-colonne".

De plus, si la table contient des colonnes d'"historical data", repérées par le mot clé *hd*, sont aussi créés les répertoires :

"$IS/nom-base/FUTURE/nom-table/nom-colonne",

"$IS/nom-base/OLD/nom-table/nom-colonne".

5.2.2.3 La création d'une table dans le monde PrologII "model"

Pour simplifier l'interpréteur Impish, il s'est avéré nécessaire de conserver en Prolog, sous forme de prédicats, des informations telles :

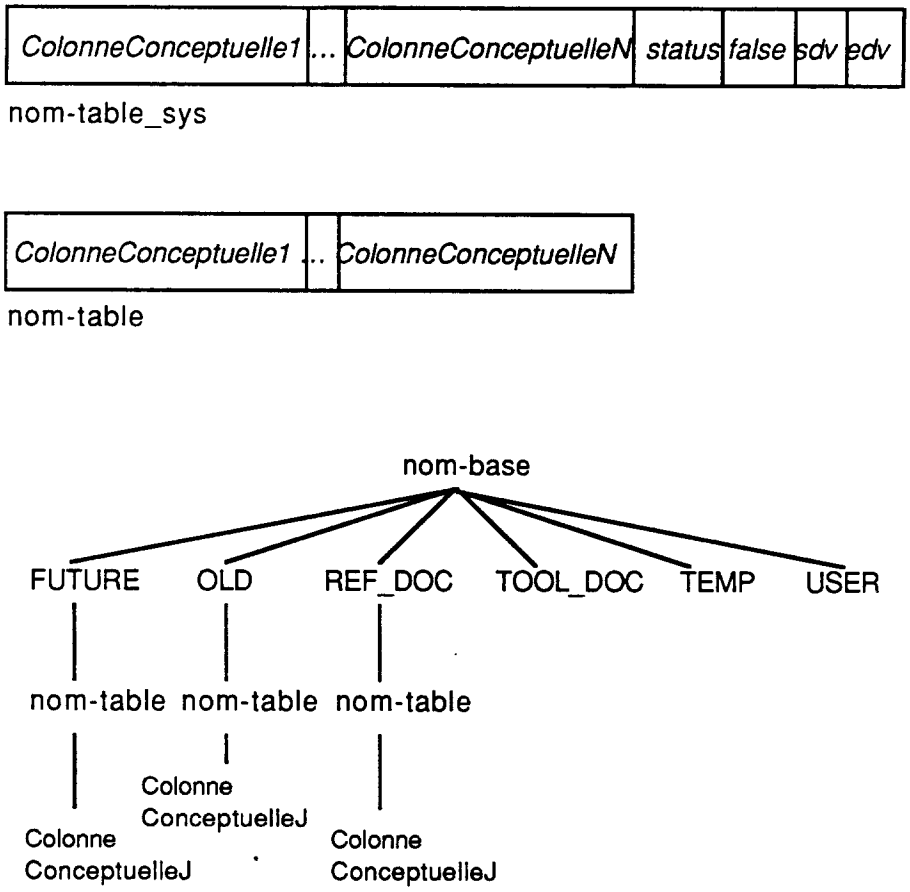
- la trace de chacun des liens :
  link(table-name,link-name)->;

- la trace de chacune des colonnes historiques :
  hdata(table-name,column-name)->;

- la trace de chacune des colonnes non *link* :
  attribute(table-name,column-name,domain-name,range)->;

- la trace de chacune des clés de la table, repérée dans la commande par le mot clé *key* :
  key(table-name,column-name,range-in-the-key-list)->;

Ces diverses assertions d'ordre 0 seront utilisées par le LM d'Impish, qui accèdera aux multiples informations qu'elles contiennent par unification.




```
attribute(nom-table,ColonneConceptuelle1,Domaine1,1)->;
...
attribute(nom-table,ColonneConceptuelleN,DomaineN,N)->;
key(nom-table,ColonneConceptuelle1,1)->;
hdata(nom-table,ColonneConceptuelleK)->;
link(nom-table,ColonneConceptuelleJ)->;
```

PrologII

fig. 9 : Les trois "sous-bases" d'Impish après la création d'une table

5.2.2.4 Suppression d'une table Impish

Un drop table Impish permet de détruire, si elle ne

contient plus d'informations, une table, c'est à dire les tables créées par ESQL, les répertoires Unix correspondant à ses liens, et les prédicats du monde "model" s'unifiant à attribute(table-name,x,y,z), link(table-name,u), hdata(table-name,v), ou à key(table-name,w).

5.2.3 Définition des contraintes et démons

- Les contraintes d'unicité de clé sont exprimées dans la définition de la table et donc simplement déclarées au niveau de PrologII.

- Une référence d'un attribut table1.col1 vers table2.col2 est simplement représentée par une assertion :

```
reference(<"table1","col1">,<"table2","col2">) ->;
```

En cas de conjonction (col1 référant à col2, col3, ...), on insère autant de règles que de termes dans la conjonction.

- Une contrainte opérationnelle est définie par un programme PrologII, dont le point d'entrée est un prédicat paramétrable par des noms d'attributs.

Sa création a pour conséquences l'insertion dans le monde Prolog "model" des clauses suivantes :

* les règles résultats de l'intégration du programme dans le monde "model" d'Impish par le mécanisme décrit en 6.3,

* une assertion :

program(nom-du-prédicat-point-d-entrée, identifiant-de-programme) ->;

qui permet de relier programme et point d'entrée pour une contrainte.

* une trace de la contrainte :

constraint (nom-de-table,nom-d-opération,moment-d-activation,
identifiant-de-programme,nom-du-prédicat-point-d-entrée)->;

* une trace de tous les paramètres du prédicats, :

parameter (<nom-de-table,nom-de-colonne>,rang-du-parametre,
identifiant-de-programme, nom-du-prédicat-point-d-entrée)->;

On verra en 5.3 comment sont utilisées ces traces.

- Une dépendance fonctionnelle d'un attribut table.col envers un n-uplets d'autres attributs est définie par un prédicat que doivent vérifier les valeurs de cet attribut et du n-uplets (n + 1 variables), dans certaines circonstances que nous allons préciser.

Sa création suit le même processus que dans le cas précédent, si ce n'est que :

* la trace de la contrainte n'est plus un prédicat constraint mais un :

dependence(<t-able,c-ol>,
nom-du-prédicat-point-d-entrée,
identifiant-de-programme)->;

où <t-able,c-ol> représente l'attribut dépendant, le "résultat" de la "fonction", et où l'identifiant du programme est celui fourni par le mécanisme d'intégration.

* l'on génère les traces des contraintes opérationnelles à vérifier (au même moment que les autres contraintes opérationnelles "après" que l'on a pu définir explicitement, cf. cas précédent), mais qui sont déduites de la dépendance fonctionnelle. Ce sont, pour une dépendance donnée de Y envers x1,…,xN, les contraintes suivantes :

.après insertion de Y,

.après modification de Y,

.après modification de chaque xI tel que xI n'appartient pas à la même table que Y,

.après suppression de chaque xI tel que xI n'appartient pas à la même table que Y.

On devra prendre soin d'écrire le programme Prolog qui décrit la contrainte de manière à ce que toutes les configurations d'appels possibles i.e. (1) Y et les autres valeurs de la table d'Y instanciées et xI libres, (2) un certain nombre d'xI de la même table instanciés et Y et les autres xJ libres, conduisent à des résultats. L'administrateur fera notamment attention en utilisant les fonctions évaluables de PrologII. Il prendra soin, aussi, de ne pas passer deux fois la même colonne en paramètre : le système ne saurait pas si il doit essayer de les unifier ou pas, et du coup fait échec.

- Un domaine de valeurs est défini par un programme PrologII, dont le point d'entrée est un prédicat dont l'unique paramètre s'applique aux valeurs de la colonne contrainte. Un domaine est toujours un sous domaine d'un type de base ESQL

("pair" sera un domaine sous domaine de "entier", par exemple).

Sa création se traduit par :

* l'intégration du programme,

* l'assertion d'un prédicat program...

* l'assertion d'un prédicat :

domain (nom-du-prédicat-point-d-entrée, identifiant-de-programme, type-ESQL-de-base) ->;

Le nom du domaine est déduit, par le système, du nom du prédicat point d'entrée.

- Un démon est défini par un "déclencheur" qui est une manipulation de tuples d'une table (un *insert* ou *delete*, nous n'autorisons pas la définition de démons sur des *update* : lors d'un *update*, on appliquera les démons des *insert* et des *delete correspondants*), d'un moment (avant ou après la manipulation), et d'une action contenue dans un programme Prolog intégré au monde "model". Cette action est accessible par un prédicat point d'entrée qui peut contenir des variables correspondant à des attributs de la table manipulée.

La condition d'activation d'un démon est l'unification du tuple à manipuler avec un prédicat effaçable dans la base (dont l'identifiant est un nom de table).

Par exemple, le démon sur tab1 qui a quatre colonnes:

create deamon after
insert tab1 (1,2,x,4)

do do-something-with(col1,col2)

sera déclenché, on le verra plus loin, après l'insertion de tout tuple dont les valeurs de colonnes seront respectivement (1,2,quelconque,4).

Dans tous les cas, le programme contient, pour être qualifié de démon, des prédicats à effets de bord sur la base de données comme ceux présentés en 6. (db-insert, db-delete, ...).

Sa trace est de la forme :

deamon(nom-table,déclencheur,moment,c-lause-id, identifiant-de-predicat)->;

où "moment vaut avant ou après.

Les paramètres du prédicats point d'entrée, comme dans le cas des contraintes, sont aussi répertoriés.

5.2.4 Définition des types de documents produits (*tool-doc*)

La création d'un type de document, par *create doctype*, génère la création d'un répertoire Unix :

"$IS/nom-base/TOOL-DOC/nom-du-type-de-document".

Si la commande est paramétrée par un "path" Unix, le contenu de ce fichier est considéré comme un document typique (plan type, exemple, ...) et copié dans un fichier :

"$IS/nom-base/TOOL-DOC/nom-du-type-de-document/draft".

Ces deux résultats sont obtenus par l'appel, au niveau de l'interprète d'Impish, à deux nouveaux prédicats ajoutés à ceux que propose PrologII.

Une trace du doctype est conservée dans le monde "model" sous la forme :

tool-doc(nom-du-type-de-document)->;

Le *create doctype* peut être inversé par un *drop doctype*.

5.2.5 Définition d'une hypothèse

La définition d'une hypothèse commence par un *begin hypothesis*, qui met une variable globale Prolog hypothesis à defining, se poursuit par une série de manipulations (qui ne peuvent être historiques, et qui sont des manipulations de faits hypothétiques décrites en 5.3). Elle se conclut par un *commit hypothesis*, qui relâche la variable Prolog hypothesis. Dans Impish, et pour des raisons simplificatrices, on n'autorise la définition que d'une seule hypothèse.

On remarquera que ce sont là des opérations du LMD qui sont utilisées. Mais, à notre sens, les hypothèses concernant tous les utilisateurs de la base, leur définition initiale, la validation de leur cohérence par rapport à la base courante, et la "réalisation" (c'est à dire la répercussion de leur contenu dans cette même base courante), doivent être commandées par un "super-utilisateur" que l'on assimilera à l'administrateur. Leur manipulation, au contraire, est ouverte aux outils.

La validation et la "réalisation" d'une hypothèse seront présentées dans le LMD, dans la mesure où son mécanisme est très proche de celui qui implémente le travail d'un outil dans une hypothèse.

5.3 Le LM d'Impish

5.3.1 Manipulation des faits courants

5.3.1.1 Insertion

Un tuple (x1,…,xN), de type T, qui représente un fait courant est enregistré sous la forme T_{sgbd}(x1,…,xM,"c") dans une table système T_sys (cf. 5.2), pour ce qui concerne ses éléments atomiques de types entier, chaine de caractères, réel, date ou "argent" qui sont reconnus par le SGBD, et sous la forme T_{unix}(xI,…,xJ) dans une hiérarchie "../REF_DOC/T" de fichers Unix pour tous ceux de type "documents de référence" (ou "lien").

Le "c" instancie la colonne *status* et définit le caractère courant du fait. Est utilisé, pour l'insertion, un *<insert statement>* de ESQL.

Les fichiers représentant les liens sont identifiés, sous "../REF_DOC/T/xI/cléI", …, "../REF_DOC/T/xJ/cléJ". Les cléI, …, cléJ sont le résultat de la concaténation des valeurs $T_{clé}$(xC1, …, xC2) du tuple qui correspondent aux colonnes clés (cf. 5.3.2 et 5.2). Notons que, si des liens existent, $T_{clé}$ ne peut pas être vide.

Le passage des valeurs T_{unix}(xI,…,xJ) peut être réalisé soit par référence (xI = chemin d'accès d'un fichier dont la commande va faire une copie), soit par valeur (xI = VAL "chaine de caractères de taille quasiment illimitée") où VAL est un mot clé et où "chaine de caractères de taille quasiment illimitée" va être stocké comme contenu du lien.

Exemple :

insert into exemple
(clé1, clé2, col3, lien1, lien2)
values ("A", "B", "C", "/usr/CETE/bosco/monDoc",
val "voila ce que je veux mettre dans lien2")

qui génère :

- l'insertion dans les colonnes (*clé1*, *clé2*, *col3*, *status*) de la table *exemple_sys* du tuple ("A", "B", "C", "c"),

- la création des fichiers Unix :

"../nomBase/exemple/lien1/AB" et
"../nomBase/exemple/lien2/AB",

où :

"../nomBase/exemple/lien1/AB"

contient ce que contient "/usr/CETE/bosco/monDoc".

Au contraire, le contenu de :

"../nomBase/exemple/lien2/AB" vaut :

voila ce que je veux mettre dans lien2

5.3.1.2 Modification

Le *update* d'un tuple Impish se comporte comme celui d'ESQL à deux nuances près :

- il autorise la modification d'un lien (par écrasement du contenu du fichier qui le représente) par passage soit du chemin d'accès au nouveau contenu, soit de la valeur du contenu sous forme de chaine de caractères. La modification est obtenu par simple appel à un *cp* d'Unix intégré à un prédicat Prolog, cf. 6.3.4..

- son mécanisme de sélection des tuples à modifier (la WHERE-clause d'ESQL) est étendu par une WITH-clause :

Une WITH-clause utilise un prédicat Prolog comme un filtre : ce prédicat doit en effet être vérifié par les tuples, comme doit l'être la clause conditionnelle de la WHERE-clause.

Exemple :

```
update exemple
set clé1 = "U",
    lien1 = "/usr/CETE/boi/sonDoc",
    lien2 = val "voilà le nouveau contenu de lien2"
where clé2 = "B"
with voyelle(exemple.clé1)
```

- recherche d'abord tous les tuples de *exemple* qui satisfont à :

```
where clé2 = "B"
```

par utilisation d'un *select* ESQL. On notera au passage que la WHERE-clause ne peut porter sur des liens.

- filtre ensuite ces tuples par le prédicat :

voyelle(exemple.clé1)

pour ne rendre, on l'a deviné, que ceux dont la valeur pour la colonne *clé1* de *exemple* est dans {"A", "E", "I", "O", "U", "Y"}. On remarquera ici que la WITH-clause ne peut porter sur des liens. L'activation du prédicat *voyelle* est réalisé par métainterprétation d'un programme qui a été préalablement intégré au système : ces mécanismes sont détaillés en 6.3.3.

- effectue, pour les tuples ainsi retenus, les modifications souhaitées, c'est à dire ici le changement de la valeur de *clé1* (*update* ESQL reconstruits et activés), et pour les liens, à la fois le changement d'identifiant (puisque leur clé a changé) et de contenu.

5.3.1.3 Suppression

Le *delete* d'Impish est, comme *update*, étendu à la manipulation des liens, et utilise le même mécanisme de filtrage.

En cas d'échec de l'une de ces trois commandes simples, la restauration de la base de faits est assurée par le SGBD. En effet, quand les causes d'erreurs proviennent du SGBD (problèmes de types, …), il suffit de commencer par activer les effets de bords sur la base (les commandes ESQL), et terminer par ceux sur les liens, pour que ce soit le SGBD qui détecte les commandes erronées. Schématiquement :

Impish-command(c-ommand) ->
    ESQL-command(c-ommand)
    Unix-command(c-ommand);

Le SGBD empêche l'exécution dans la base, et le prédicat **ESQL-command** qui implémente cette commande partielle fait échec. De ce fait, le prédicat **Unix-command** qui implémente l'action sur les liens, et qui est le but suivant dans l'exécution de la commande, n'est pas effacé.

Quand les erreurs proviennent de la base Unix (il peut s'agir de problèmes de droits d'accès, puisque la méconnaissance d'un type de lien passé en paramètre à la commande est traitée préalablement, comme les autres causes d'erreur éventuelles), l'état de la base de faits est rétabli par le *rollback work* du SGBD.

5.3.1.4 Recherche

Le *select* d'Impish est étendu par la WITH-clause. La WHERE-clause permet, comme en ESQL, l'expression des jointures et projections sur des colonnes non *link*.

Il permet aussi la recherche des liens : quand un nom de lien apparait dans la liste de sélection,

- les clés entières des tuples sélectionnés (c'est à dire filtrés par les WHERE et WITH-clause) sont récupérées,

- les chemins d'accès aux liens reconstruits,

- les liens ainsi retrouvés recopiés dans un espace "../nomBase/USER" sous des noms calculés et renvoyés, à leur place, dans les tuples résultats de la requête.

La convention "*" peut être utilisée, comme en ESQL. Le caractère "*" représente l'ensemble des colonnes de type

différent de *lien*. Les noms de colonnes *liens* doivent apparaitre explicitement avant, après, ou avant et après l'"*".

Exemple :

select lien1, *, lien2 *from* exemple *with* voyelle(exemple.clé1)

peut renvoyer :

("../nomBase/USER/lien1.UB","U","B","C","../nomBase/USER/lien2.UB")

5.3.1.5 La WITH-clause

La syntaxe de la WITH-clause est la suivante :

with
<identifiant_de_prédicat>
(<liste_de_noms_de_colonnes_ou_valeurs>)

La composition et l'exécution du but PrologII est réalisée par le prédicat IS-select-plus :

```
IS-select-plus   (s-elect-list,p-red-arg-list,f-ull-list-att,p-redicate-name,
                  t-uple.k-db-result,  t-uple.result)->
                 compose-goal(p-redicate-name,  f-ull-list-att,  p-red-arg-list,
                    t-uple,g-oal)
                 /compose le but PrologII en remplaçant les noms de colonnes arguments dans
                 la with-clause par les valeurs correspondantes du t-uple dans k-db-result
                 obtenu après filtrage par la where-clause. Voir aussi plus bas c
                 ompose-constraint-goal/
                 program(p-redicate-name,  c-lause-id)
                 /repère le paquet de clauses c-lause-id sous lequel le programme incluant
                 p-redicate-name a été intégré (cf. 6.3.3)/
```

```
run (normal, c-lause-id, g-oal)
/lance la métainterprétation du g-oal par le métainterpréteur normal, dans
le paquet désigné par c-lause-id/

IS-select-plus(s-elect-list,p-red-arg-list,f-ull-list-att,p-redicate-name,
        k-db-result,result)->;
```

5.3.2 Utilisation des contraintes. Transactions.

5.3.2.1 Introduction des contraintes

Nous avons présenté différents types de contraintes, qui sont prises en compte par les commandes de manipulation. Nous distinguerons :

- les contraintes d'unicité de clé, et les contraintes portant sur les domaines de valeur qui ne sont utiles que pour l'*insert* et l'*update*,

- les autres contraintes, valables pour les trois manipulations.

5.3.2.2 Vérification des contraintes d'unicité de clé

Lorsqu'une clé a été définie sur une table, avant tout *insert* ou *update* est testée la non existence d'un tuple ayant la même clé, et ce grâce à un prédicat prolog qui lance un *select* statement à ESQL : Le résultat de ce *select* doit être *nil* puisque sa WHERE-clause contient la restriction des valeurs des colonnes clés aux valeurs introduites :

```
verify-key(t-able, l-key) ->
/l-key a été préalablement extraite du tuple à insérer/
```

```
extract-key-doublet(t-able, l-key, l-doub, 1)
/extrait dans l-doub les couples <nom-de-colonne, valeur>/
KDB-compose-where-key-clause(t-able, l-doub, w-here,1)
/compose une chaine du type " col1 = val1 and ... and colN = valN"/
if-then(dif(l-doub, nil),
    list-conc("select * from ".table." where ".w-here.nil, command).
     ex(c-ommand, nil, l-length, t-length, l-type, e-rror));
  /exécute la commande "select ..."/
```

On note que le cas où il n'y a pas de clé est testé et traité. On remarque aussi l'utilisation du prédicat "ex" défini plus loin en 6.3.2.

5.3.2.3 Vérification des contraintes de domaines de valeurs

Pour un tuple donné, à chaque nouvelle valeur de colonne est appliqué, avant l'insertion, le prédicat :

```
verify-domain(t-able, v-alue, n) ->
    attribute(t-able, c, d-omain-predicate,n)
    domain(d-omain-predicate, p-redef-type, c-lause-id)
    /
    error-if (not (run (normal, c-lause-id, <d-omain-predicate,
                                v-alue>)), 900.v-alue.d-omain-predicate);
verify-domain(t-able, v-alue, n) ->;
```

Le nom du domaine est récupéré dans la variable d-omain-predicate, par unification du prédicat attribute(t-able,c, d-omain-predicate,n) dans le monde PrologII "model" (cf. la représentation des tables en PrologII par *create table* dans le chapitre consacré au LD d'Impish).

Si ce d-omain-predicate n'est pas un type de base ESQL, résultat obtenu par unification du prédicat

domain(d-omain-predicate, p-redef-type, c-lause-id) dans le monde PrologII "model" (cf. la représentation des domaines en PrologII par *create domain* dans le chapitre consacré au LD d'Impish), la métainterprétation du programme définissant le domaine (et identifié par c-lause-id) par le métainterpréteur normal du d-omain-predicate est activée par run sur la valeur v-alue dont on teste le domaine.

Si ce d-omain-prédicate est un domaine de base ESQL, c'est ESQL qui est automatiquement chargé de la vérification du domaine.

On remarquera le traitement de l'erreur qui renvoie le nom du domaine qui n'est pas vérifié et la valeur qui ne le vérifie pas, ce qui permet des messages expressifs, comme par exemple :

"Error : 11 does not belong to the domain even".

5.3.2.4 Vérification des contraintes de références

Pour vérifier les références lors d'une insertion, quand la colonne insérée réfère à une autre (le test est réalisé par unification de reference(<t-able,c>,<t-able1,c1>) dans le monde "model", cf. le *create reference*), un *select* est composé, avec une WHERE-clause restreignant la valeur de c1 (colonne de la table référée) à l'égalité à la valeur v-al que l'on veut introduire. Si le résultat de son activation vaut nil, alors la référence n'est pas satisfaite.

```
verify-reference("INSERT", t-able,v-al,n) ->
    attribute(t-able,c,t,n)
    all-the(
    reference(<t-able,c>,<t-able1,c1>).
    compose-reference-select(t-able1,c1,v-al,r-esult).
```

```
        error-if(eq(r-esult,nil),901.v-al.t-able.c.t-able1.c1))/;
    verify-reference("INSERT", t-able,v-al,n) ->;

    compose-reference-select(t-able1,c1,v-al,r-esult) ->
        list-conc("select unique".c1."from".t-able1."where".c1."=".v-al.nil,
                c-md)
        ex(c-md, r-esult, l-length, t-length, l-type, e-rror);
```

Pour vérifier les références lors d'une suppression de tuple, quand la colonne supprimée est référée par d'autres (le test est réalisé par unification de reference(<t-able,c>,<t-able1,c1>) dans le monde "model, cf. le *create reference*), des *select* sont composés, avec une WHERE-clause restreignant la valeur de c (colonne de la table référante) à l'égalité à la valeur v-al que l'on veut supprimer. Si le résultat de son activation ne vaut pas nil, il existe encore des valeurs référantes, et la suppression ne peut avoir lieu. A ce test près, le code du verify-reference("DELETE",t-able,v-al,n) est le même que celui portant sur l'*insert*.

Pour vérifier les références lors d'une modification de tuple, pour tous les cas où une nouvelle valeur de colonne est différente de l'ancienne (connaissance déduite de l'expression de la commande, par comparaison de la SET-clause et de la WHERE-clause), on applique successivement les deux mécanismes précédents.

Toutes les vérifications présentées jusqu'ici sont faites avant la manipulation proprement dite.

5.3.2.5 Vérification des contraintes opérationnelles générales

Quand une contrainte a été définie en utilisant le *create constraint*, elle est vérifiée avant ou après (cela est explicitement contenu dans sa définition) la manipulation qu'elle doit restreindre (cela aussi est explicitement contenu dans sa définition). Le détail du mécanisme se trouve, sous forme de commentaires, dans l'exemple présenté ci-après.

```
verify-general-constraint("AFTER", "INSERT", t-able, t-uple) ->
    all-the(constraint(t-able, "AFTER", "INSERT", c-lause-id,
            p-redicate).
    /identifie toutes les contraintes concernées, cf. create constraint, par
    unification dans le monde "model"/
    compose-constraint-goal(c-lause-id, p-redicate, t-able, t-uple,
                                                    g-oal).
    error-if(not(run(normal, c-lause-id, g-oal)),902.g-oal));

compose-constraint-goal(c-id, p-red, t-able, t-uple, g-oal) ->
/compose le but Prolog dont l'effacement correspond à la satisfaction de la
contrainte/
    compose-constraint-goal'(c-id, p-red, t-able, t-uple, 1, g-oal')
    make-goal-shape(p-redicate.g-oal', g-oal);
    /simple mise en forme de but Prolog/

compose-constraint-goal'(c-id, p-red, t-able, t-uple, m, v-al.g-oal') ->
    parameter(<t-able,c>,m,c-lause-id,p-red)
    /récupère, dans le monde PrologII "model", le nom c de la colonne de
    t-able qui entre comme m-ième variable dans le prédicat p-red/
    /
    attribute(t-able,c,t,n)
    /récupère, dans le monde PrologII "model", la place n de la colonne c
    dans t-able, qui est aussi la place de v-al dans t-uple/
    range-number-of-in(n,v-al,t-uple)
    /recupère v-al et la met dans la liste les valeurs de paramètre du but
    que l'on construit/
    plus(m,1,m1)
```

```
compose-constraint-goal'(c-id, p-red, t-able, t-uple, m1, .g-oal');
/cherche le paramètre suivant/
compose-constraint-goal'(c-id, p-red, t-able, t-uple, m, nil);
```

Bien entendu, les "contraintes avant" sont vérifiées avant manipulation, les "contraintes après" après.

Le cas des contraintes après pose une difficulté : en cas d'échec, il faut pouvoir revenir dans un état de la base de faits et de documents qui soit celui qui précèdait la manipulation. Il devient impossible de confier automatiquement cette tache au SGBD (alors que nous l'avions fait dans le cas des manipulations sans contraintes). Nous traiterons de cela dans le paragraphe traitant des transactions dans Impish.

5.3.2.6 Vérification des dépendances fonctionnelles

Elle consiste en la vérification des contraintes opérationnelles "après" générées lors de leur définition :

Comme pour les contraintes générales, le but Prolog est recomposé : Chaque fois, du tuple manipulé sont extraites les valeurs des colonnes de la même table et qui sont paramètres du prédicat. Les autres paramètres du prédicat, qui correspondent à des colonnes d'autres tables, sont remplacées par des variables libres.

Si le but ainsi reconstitué est effacé, alors la contrainte est satisfaite.

5.3.2.7 Les micro-transactions

Nous avons choisi, dans une commande Impish, d'encapsuler les appels à ESQL dans une transaction, et de

conclure par un ESQL *commit work*, ou un ESQL *rollback work* selon que les contraintes étaient ou n'étaient pas vérifiées. Pour éviter que la base de documents (documents sur lesquels, on le répète, ne portent aucune contrainte) ne soit touchée, les manipulations du SGBD et les vérifications des contraintes sont préalables à toute intervention sur la base de documents.

5.3.2.8 Les transactions

Les contraintes introduisent une complexité supplémentaire dans la gestion de la base de donnés : il devient notamment nécessaire d'autoriser des incohérences momentanées lors d'une succession de manipulations, l'essentiel étant d'aboutir à un état cohérent.

Pour rendre cela possible, nous utilisons le mécanisme de transaction d'ESQL qu'il a fallu enrichir pour gérer l'activation des vérifications de contraintes d'une part, et les transactions sur les documents d'autre part.

En ce qui concerne les contraintes, au sein d'une transaction, ne sont vérifiées au moment de l'activation des commandes que les "contraintes avant". La vérification de l'ensemble des "contrainte après" est regroupée en fin de transaction, : au lieu d'être effacés comme précédemment, les buts PrologII, une fois calculés, sont stockés et leur run déclenché au moment de l'Impish *commit work*.

Une transaction Impish commence par un *begin work*, homologue de son homonyme ESQL, mais qui met en place les structures nécessaires à une éventuelle reprise, à savoir :

- une exécution d'un ESQL *begin work*,

- un buffer "ss.log" PrologII qui contiendra les

commandes à la base de documents qui auront été lancées,

- un buffer "after.log" qui contiendra les "contraintes après" à effacer,

- une variable PrologII globale transaction qui vaudra current ou not-current.

Un échec lors d'une vérification de "contrainte avant" au sein d'une transaction arrête celle-ci. Les "contraintes après" jusque là apparues sont alors vérifiées.

Si toutes les "contrainte après" sont vérifiées, alors l'ensemble des opérations validées au niveau des "contraintes avant" peuvent être répercutées par un *commit work*. Au contraire, si l'une des contraintes après n'est pas vérifiée, un *rollback work* est appliqué à l'ensemble de la transaction.

Au cours d'une transaction Impish (transaction à current), les micro-transactions sont débrayées.

Enfin, les modifications de la base de documents, dont les commandes ont été stockées, sont réalisées au moment du *commit work*, dans le cas ou celui-ci est possible ("contraintes après" vérifiées).

Nous tenons à remercier ici M. Gibelli, qui est l'auteur de ce mécanisme de gestion de transactions (réalisé en PrologII) qu'il nous fallait détailler pour la bonne compréhension des problèmes.

5.3.3 Manipulation des faits historiques

5.3.3.1 Insertion d'un fait historique

Un fait historique doit être un tuple d'une relation "HD" (cf. *create table*). Le caractérisent ses attributs conceptuels et deux attributs "temporel", sa *sdv*, "start date of validity" et *edv*, "end date of validity", qui déterminent la période pendant laquelle le fait est valide.

L'utilisateur d'Impish, lorsqu'il veut insérer un tuple dans une table "HD", à l'éventail de choix suivants :

- ne pas préciser de *sdv* ni de *edv* : le fait tuple est manipulé avec comme *sdv* la date de l'insertion, et comme *edv* une valeur supposée infinie, "9999999999",

- préciser un *edv* qui doit être postérieur à la date d'insertion, et dont on verra dans un prochain paragraphe la prise en compte possible,

- préciser un *sdv* antérieur à la date d'insertion. Dans ce cas, le fait garde cette date passée comme *sdv*, mais la seule garantie offerte par Impish est la satisfaction des contraintes à la date d'insertion : c'est à l'utilisateur d'assumer le fait qu'à la date *sdv* donnée, l'insertion eut pu être impossible.

Dans tous ces cas, le *status* des tuples est instancié à "c".

- préciser un *sdv* postérieur à la date d'insertion. Dans ce dernier cas, le fait est considéré comme futur (On verra comment il est alors pris en compte dans un prochain paragraphe), les contraintes sur son insertion ne sont pas vérifiés, et son status vaut "f". Un attribut de type "lien" est stocké sous un répertoire "../FUTURE/table/lien" avec pour nom de fichier le résultat de la concaténation de la *sdv* et de

la liste des clés du tuple auquel il appartient.

Bien sûr, chaque fois que *sdv* et *edv* sont précisés simultanément, l'antériorité de *sdv* par rapport à *edv* est vérifiée.

5.3.3.2 Suppression d'un fait historique

Lorsque la suppression d'un tuple d'une table "HD" est souhaitée par un utilisateur d'Impish, le système conserve le tuple avec un *status* égal à "o", pour "old".

La suppression ne peut porter que sur un fait courant (*status* égal à "c"). Les contraintes doivent être vérifiées.

```
KDB-delete-history(t,nil)  ->;
KDB-delete-history(t,t-uple.l-tuple)  ->;
     KDB-compose-where-clause(t,t-uple,w-here,1)
     KDB-compose-delete-history-statement(t,w-here)
     KDB-delete-history(t,l-tuple);

KDB-compose-delete-history-statement(t,w-here)  ->
     conc-string(t,"_sys",t-sys)
     /on travaille sur la table système/
     get-date-hour(d-ate)
     /renvoie la date courante/
     informix-date(d-ate,  i-date)
     /mets la date au format choisi, dans les exemples "AAAAMMJJHH"/
     /
     /slash, et pas commentaire .../
     list-conc("update".t-sys."set status = ""o"", edv = ".i-date.
                    "where status =  ""c""".w-here.nil, c-md)
     ex(c-md,  e-rror);
     /lance la modification/
```

Les attributs de type "lien" sont déplacés du répertoire "../REF_DOC/table/lien" au répertoire "../OLD/table/lien". Son nom sous "OLD" devient le résultat de la concaténation de sa *sdv* au nom qu'il avait sous "REF_DOC".

5.3.3.3 Modification d'un fait historique

Il n'y a modification d'un fait avec sauvegarde de sa valeur ancienne que si la valeur d'un attribut qualifié de "HD" est modifiée. Sinon, le *update* est un *update* Impish normal.

La modification ne peut porter que sur un fait courant (*status* égal à "c"). Les contraintes doivent être vérifiées.

Les anciens liens sont stockés sous "OLD" et remplacés par les nouveaux.

La figure suivante présente l'état de la base de données après quelques manipulations historiques :

col1	status	false	sdv	edv
"je suis un fait courant sans date de fin de validité"	c		1988062410	9999999999
"je suis un fait courant avec une date de fin de validité"	c		1988010100	1989052900
"je suis un vieux fait"	o		1986010100	1987010100
"je suis un fait futur"	f		1992010100	9999999999

fig. 10 : Un ensemble de tuples historiques

5.3.3.4 Recherche d'un "vieux" fait

Les WHERE-clause et WITH-clause d'un *select history* peuvent porter sur les attributs conceptuels (comme pour le *select* normal) et sur les attributs temporels *status*, *sdv* et *edv*.

La liste des noms de colonnes sélectionnées peut aussi contenir ces attributs temporels.

Alors que le ESQL *select* envoyé par le *select* normal d'Impish au SGBD porte sur les tuples dont le *status* vaut "c", celui envoyé par le *select history* porte sur les tuples dont le status vaut "c" ou "o" :

```
KDB-select-history(u-nique, l-col, s-ys-l-tab, w-here, a-nswer) ->
    list-conc("select".u-nique.l-col,c1)
    list-conc(c1."from".s-ys-l-tab,c2)
    /de la cuisine/
    if-the-else(dif(w-here,nil),
        conc-string("where status = ""c"" or status = ""o"" and ",
                w-here, s),
        eq(s,"where status = ""c"" or status = ""o"""))
    /selection sur status courant ou passé (old)/
    conc-string(c2,"  ",s1)
    conc-string(s1,s,c-md)
    /de la cuisine/
      ex(c-md,a-nswer,l-length,t-length,l-type,e-rror);
    /envoi de la requête au SGBD/
```

De même, il est possible d'accèder aux *sdv* et *edv* d'un fait courant en précisant, dans la WHERE-clause du *select history* :

"where status = "c"".

5.3.4 Prise en compte du futur

Il existe dans Impish deux types d'informations définies comme "futures" : les faits courants avec une *edv* précisée, et les faits futurs.

Tous deux sont pris en compte lors de l'activation d'une commande *progress*, qui porte soit sur une liste de table (*progress table*), pour des faits restreints par une WHERE et une WITH-clause, soit sur une base entière (*progress database*).

Le *progress table* sélectionne, dans les tables considérées, les tuples courants où *edv* est antérieur à la date du jour (et qui satisfont aux WHERE et WITH-clauses), et lance sur eux un *delete history* qui les fait passer à un *status* "o" et qui déplace les liens sous "OLD".

Ensuite, il sélectionne les tuples futurs où sdv est antérieur à la date du jour. Pour ceux d'entre eux dont la clé n'existe pas encore, il active un *insert history*. Pour ceux, dont la clé, au contraire, existe déja, il procède à un update history. Les liens, présents sous "FUTURE", sont déplacés sous "REF_DOC" dans les deux cas.

Le *progress database* active le *progress table* sur toutes les tables de la base.

On peut voir, dans la figure qui suit, quel aurait été l'application d'un progress table sur la table de l'exemple précédent, le 5 Janvier 1992 :

col1	status	false	sdv	edv
"je suis un fait courant sans date de fin de validité"	c		1988062410	9999999999
"je suis devenu un vieux fait"	©		1988010100	1989052900
"je suis un vieux fait"	o		1986010100	1987010100
"je nesuis plus un fait futur"	©		1992010100	9999999999

fig. 11 : Intégration de faits futurs

5.3.5 Activation de démons

Lors de toute manipulation "manipulation" (*insert*, *delete*, ou *update*) sur une table "table", un test est réalisé pour identifier les démons activables.

5.3.5.1 Dans le cas d'un *insert*, ce test consiste :

- à recomposer, à partir des valeurs de colonnes manipulées c1,…,cN un prédicat table(c1,...,cM) (où l'on remplace les valeurs non manipulées par des variables libres),

- puis à essayer d'unifier dans le monde "model" le but :

deamon("table", table(c1,...,cM), "manipulation", c-lause-id,p-redicat)

Pour tous les démons trouvés, (et ce dans un ordre quelconque, car il ne nous a pas été évident d'en définir un à priori…), le système repère les paramètres du prédicat point d'entrée et les remplace soit par des variables libres, soit par leur valeur si elle apparait dans le tuple manipulé.

Enfin, on efface le but prolog ainsi obtenu.

5.3.5.2 Dans le cas d'un *delete*, les démons sont déclenchés pour les tuples à supprimer qui s'unifient avec le déclencheur.

5.3.5.3 Dans le cas de l'*update* sont déclenchés les démons des *insert* et *delete* correspondants

On remarquera que le mécanisme s'apparente à ceux utilisés pour vérifier les contraintes, mais que, dans le cas des démons, les clauses du programme prolog activé peuvent contenir des prédicats à effets de bord sur la base, comme ceux présentés en 6. (db-insert, db-delete, ...).

Si la manipulation est incluse dans une transaction, tous ces effets de bord sont réversibles. De plus, les vérifications de contraintes "après" les prennent en compte.

5.3.6 Manipulation de documents produits (*tool-doc*)

5.3.6.1 Stockage d'un document

L'enregistrement d'un document, *store tool-doc*, est accompagné d'une liste de dépendances :

```
store tool doc "myDocSource" doctype "DocName"
        strong from table1
            where col11 > 10
            with pred2(col12),
        low from table2
            where col21 < 22
```

Il provoque la copie de "myDocSource" sous "$IS/base/TOOL-DOC/doctype/DocName" et l'insertion dans la table *doctable* des tuples de dépendance :

Dans l'exemple, si *strong from …* renvoie les clés de tuples suivantes :

(11."string".nil).(12."string".nil).nil
(forme d'un résultat de *select* sous Impish)

et *low from …* renvoie :

(20.nil).nil

les tuples insérés sont :

(doctype, DocName, table1, "11string", "strong",y)
(doctype, DocName, table1, "12string", "strong",y)
(doctype, DocName, table2, "20", "low",y)

Les commandes *update* et *delete*, que nous avons déja détaillées, provoquent le basculement du flag ("y" devient "n") en cas de modification ou de suppression du tuple de dépendance.

5.3.6.2 Suppression d'un document

Par *delete tool-doc* on détruit un document et les dépendances.

5.3.6.3 Recherche d'un document par son nom

get tool-doc par1 par2 renvoie le ou les documents de type *par1* qui coïncident, selon la convention étoile d'Unix,

avec *par2*, et leur "niveau de confiance".

5.3.6.4 Recherche d'un document par ses mots clés

On peut remplacer *par2* par une liste de dépendances : le *get tool-doc* retourne alors les documents avec leur "niveau de confiance" qui satisfont, au moment de l'interrogation, à ces dépendances : si une dépendance n'est plus vérifiée (flag à "n"), le document concerné n'est pas sélectionné.

5.3.6.5 Obtention du "niveau de confiance" à accorder au document

Ce niveau de confiance est calculé comme suit :

- recherche du triplet de nombres de dépendances fortes, moyennes, faibles (nF,nM,nf) encore satisfaites,

- recherche du triplet de nombres de dépendances (nFi,nMi,nfi) définies au départ,

- calcul de :

$$(3.nF + 2.nM + nf)/(3.nFi + 2.nMi + nfi) = C.$$

Si C = 1, alors confiance "absolute", si 0.70 < C < 1, confiance "high", si 0,40 < C < 0.70, confiance "medium", et "low" si C < 0.40.

Un tel modèle demande à être vérifié par l'expérience, et l'utilisation d'Impish par des outils de gestion de projets aidera à en apprécier le bien fondé.

5.3.7 Manipulation des faits hypothétiques

5.3.7.1 Travail dans un contexte hypothétique

Un outil voulant travailler dans le "contexte hypothétique" d'Impish le déclare par un *start hypothesis* qui met la variable Prolog **hypothesis** à **using**.

A partir de là, les commandes de manipulation d'Impish utilisées par l'outil ont un comportement particulier que nous décrivons ci-dessous. Notons au passage que la gestion des historiques n'est pas assurée au sein du contexte hypothétique. Ajoutons que ces notions ne sont valables que dans le cadre d'une base où toutes les tables ont une clé.

Un outil peut, à tout moment, en sortir et revenir dans le contexte réel par un *close hypothesis* qui relâche **hypothesis**.

Dans la suite du paragraphe 5.3.7, toutes les manipulations, si cela n'est pas précisé, sont entendues dans le cadre du contexte hypothétique (H).

5.3.7.2 Insertion

L'*insert* d'un tuple est réalisé, après vérification de la clé sous H et des autres contraintes (cf. plus bas), avec un *status* égal à "h". Il n'apparait donc pas dans le contexte courant où *status* vaut "c", "o" ou "f".

Si le tuple existe intentionnellement nié sous H (cf. juste après), son expression négative (où *false* = "f") est effacée.

5.3.7.3 Suppression

Supprimer un tuple revient à :

- chercher dans le contexte H si il existe, c'est à dire si un tuple qui a la même clé existe (*status* = "h" et *false* à "null"),

- si oui, mettre sa colonne *false* à "f" et terminer (Cette colonne, qui est une colonne système définie au moment de la création de la table, sert à nier "intentionnellement" un fait, sans pour autant l'effacer).

- si non, chercher dans le contexte courant,

- si le fait y est, le recopier avec un *status* égal à "h" et *false* égal à "f" et terminer.

- sinon, terminer.

col1	status	false	sdv	edv
"je suis un fait courant sans date de fin de validité"	c		1988062410	9999999999
"je suis un vieux fait"	o		1986010100	1987010100
"je suis un fait hypothétique "	h		1	9999999999
"je suis un fait hypothétique nié "	h	y	2	

fig. 12 : Un exemple de contexte hypothétique

5.3.7.4 Modification

Modifier un tuple revient à :

- chercher dans le contexte H si il existe (*status* = "h" et *false* à "null"),

- si oui, le modifier et terminer.

- si non, chercher dans le contexte courant,

- si le fait y est, le recopier avec un *status* égal à "h", le modifier et terminer.

- sinon, terminer.

Pour chacune de ces manipulations, le ou les tuple(s) qui en résultent sont marqués par ordre de manipulation (dans la colonne sdv qui n'est pas utilisée puisque le mécanisme d'historisation est débranché sous H). Ce numéro d'ordre servira à la validation de l'hypothèse.

5.3.7.5 Recherche

Lors d'une sélection, les tuples renvoyés sont ceux obtenus par le processus suivant :

- la recherche des tuples qui satisfont aux WHERE et WITH-clauses (cf. note ci-après) de la sélection, à status = "h" et false = "null", qui retourne un ensemble T1 de tuples,

- la recherche des tuples qui satisfont aux WHERE et WITH-clauses (cf. note ci-après) de la sélection, à status = "h" et false = "f", qui retourne un ensemble T2,

- la recherche des tuples qui satisfont aux WHERE et WITH-clauses (cf. note ci-après) de la sélection, à status = "c" qui retourne un ensemble T3 de tuples,

- la suppression de T3 de tous les tuples dont la clé existe comme clé de tuples de T2 (ceux ci sont faux dans H) ou de T1 (ceux là sont différents dans H) pour obtenir T4.

L'ensemble des tuples retournés est l'union de T1 et T4.

5.3.7.6 Note sur la vérification des contraintes

- Le test de clé sur le contexte hypothétique se fait donc sur les tuples tels que *status* = "h" et *false* = "null".

- Dans H, la WITH-clause du *select* et la vérification des contraintes font appel à un métainterpréteur (cf. 5.3.1.5 et 5.3.2.4). Ici, ce n'est pas le métainterpréteur **normal** qui est utilisé mais un autre dans lequel l'interprétation du prédicat **db-erase** (qui étend la base de faits PrologII à la base de données dans laquelle travaille l'outil, cf. 6.), est quelque peut différente :

Alors que dans le travail courant, le **db-erase** efface des faits dont le *status* vaut "c", il efface ici les faits sélectionnés selon le mécanisme du *select* tel qu'il est décrit précédemment dans ce paragraphe (recherche des status = h et false = "null",...).

5.3.7.7 Note sur le traitement des colonnes liens

Les documents liés à des faits hypothétiques sont stockés sous les mêmes répertoires "…/REF_DOC/…" Unix que ceux

des faits courants. Leur nom, par contre, est le résultat de la concaténation de la clé du fait et de ".H".

5.3.7.8 Retour sous le contexte courant

L'outil peut retravailler dans le contexte courant : il va donc modifier l'état de la base courante. Les faits hypothétiques ne sont pas pris en compte lors de la vérification des contraintes, et la base courante reste donc toujours cohérente avec le réel.

Au contraire, elle n'est pas forcément cohérente avec l'hypothèse, et cette incohérence peut être détectée par une non vérification de contrainte dans le contexte H, dû à un tuple du contexte courant qui n'a pas été "redéfini" dans H. Impish impose donc à l'hypothèse de rester cohérente avec l'état courant, ou d'expliciter les incohérences.

5.3.7.9 Validation d'un contexte hypothétique

La validation, on l'a dit, est demandée par l'administrateur. Elle consiste en l'activation, au sein d'une transaction, de l'enchainement suivant :

- tout tuple vrai (false = "null") sous H est inséré dans le contexte courant (avec status = "c") si il n'y existe pas,

- si il existe déja, il est modifié avec les valeurs du tuple de l'hypothèse,

- tout tuple faux (false = "f") de H, si il existe avec la même clé dans le contexte courant, y est supprimé,

- si il n'existe pas, rien n'est fait.

- enfin, le contexte H est "purgé", par suppression de tous ses tuples.

Lors de toutes ces manipulations, qui sont regroupées sous une transaction, les contraintes autres que la clé sont cette fois vérifiées dans le contexte courant.

Les manipulations sont faites dans l'ordre croissant des numéros des tuples de H.

La validation d'une hypothèse peut donc entrainer la mise en évidence de son incohérence avec le contexte réel. C'est aux outils de les lever avant que cette hypothèse puisse être confirmée par un *realise hypothesis* qui ordonne le basculement de l'hypothèse dans le monde réel (et qui est en fait le *commit work* de la transaction, si tout c'est bien passé).

5.4 Conclusion

Voila donc présentées les commandes d'Impish : Avant de décrire, dans le chapitre suivant, les mécaniques internes qui permettent à Impish d'interpréter toutes ces commandes, faisons quelques remarques sur l'apport de ces langages.

D'abord, on aura pu noter, pour les quelques exemples donnés, leur compatibilité avec les commandes ESQL. Toutefois, cette compatibilité grammaticale, qui permet de faire "tourner" des applications ESQL sur Impish (à condition que le modèle relationnel sous-jacent au modèle Impish soit le même) est relativisée par les différences sémantiques introduites par les vérifications de contraintes et les activations de démons, par exemple : les programmes ESQL tournent sur Impish, mais pas forcément de la même manière que sur une base ESQL, puisque, à modèle relationnel égal, un modèle Impish est plus riche qu'un modèle ESQL.

Nous avons expliqué leur sémantique, et nous nous sommes attachés, dans ce chapitre, à traiter de la représentation des différentes connaissances relatives aux concepts d'un modèle Impish dans les monde PrologII, Unix et ESQL qui le compose : on trouvera en annexe le résultat de l'exécution d'une commande *status* qui montre l'état de ces différents monde après la définition d'un modèle.

Les "plus" apportés par Impish par rapport à un système relationnel sont exprimés dans le tableau qui suit.

Concepts ajoutés	*Commandes ou expressions nouvelles*
DOCUMENT	CREATE/DROP
CONTRAINTE	CREATE/DROP
DEMON	
HYPOTHESE	START/CLOSE/VALID
HISTORIQUE	HD (CREATE TABLE)
TYPE "LIEN"	LINK (CREATE TABLE)

Commandes enrichies	*Fonctions supplémentaires*
INSERT, DELETE, UPDATE	Gestion de versions de tuples Manipulations avec contraintes Gestion des faits hypothétiques
SELECT	Recherche récursive Recherche de documents

fig. 13 : Les apports d'Impish

6. Architecture et interfaces

6.1 L'architecture d'Impish : Introduction

Le système s'appuie sur trois composants :

- le langage logique PrologII*,
- le SGBDR Informix**,
- le système de gestion de fichier d'Unix***.

C'est un programme Prolog qui est maître du système, et qui gère l'utilisation des trois supports des informations manipulées par Impish, à savoir :

- le SGBD pour les faits,
- Prolog pour les contraintes et démons,
- Unix pour les documents.

En plus d'une interface textuelle sur laquelle nous ne nous étendrons pas, qui permet d'interpréter des commandes écrites dans la syntaxe des LD et LM d'Impish, deux interfaces vers des langages de programmation sont proposés :

- le premier pour des programmes écrits en PrologII, qui permet aux outils de type "système expert", développé dans ce langage logique, de puiser des informations dans la base d'Impish. Ces outils accèderont aux données par de nouveaux prédicats intégrés au système PrologII.

- le second vers des applications plus traditionnelles, algorithmiques et modulaires puisque développées dans un langage orienté objet (Objective-C****).

---

*PrologIA, **RDS, ***AT&T, ****PPI

Ce deuxième interface permettra aux outils, conçus comme des objets, d'accèder au système d'information vu lui même comme un objet, en communicant par échange de messages.

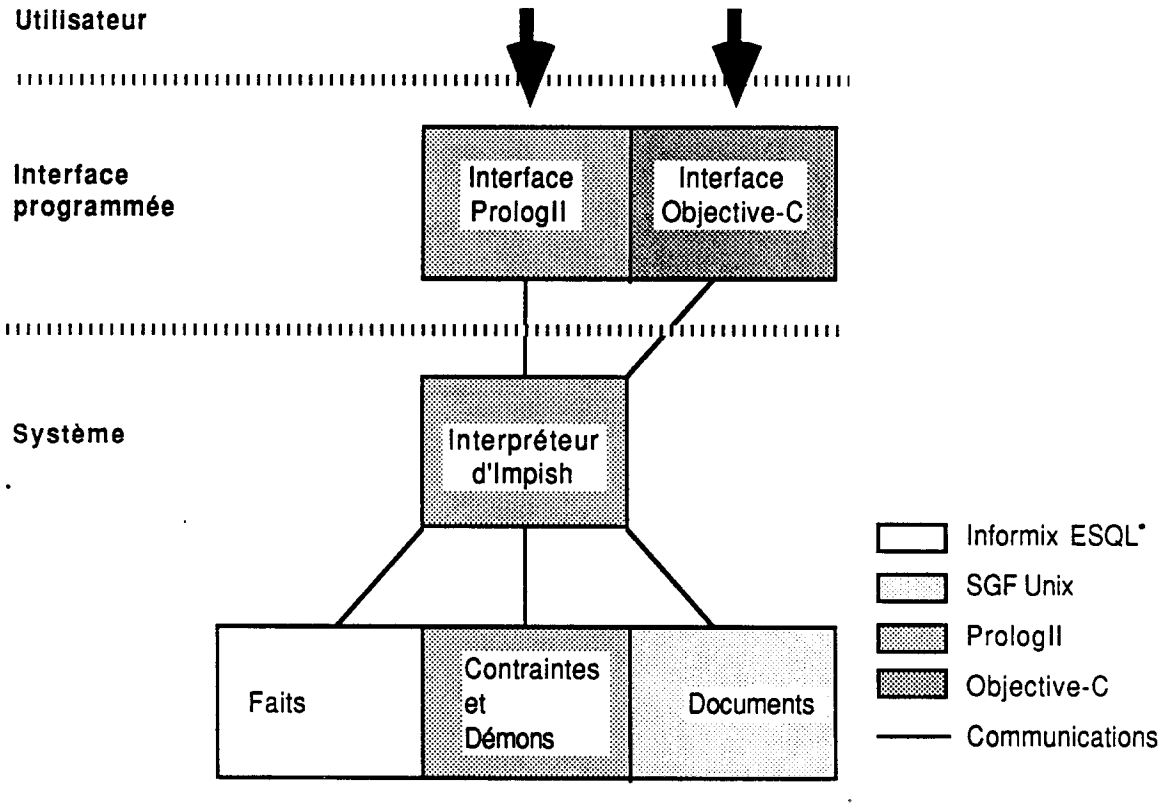


fig. 14 : L'architecture d'Impish

Nous allons, dans la suite, traiter des points suivants :

- les interfaces vers des langages de programmation, tels qu'ils ont été conçus (6.2).

- l'organisation du monde PrologII en tant que langage d'implémentation du système (6.3.1), et

- les interfaces qu'il a, à ce titre, avec :
  .le SGBD (6.3.2),
  .Prolog lui même en tant que support des informations (6.3.3),
  .et le SGF (6.3.4).

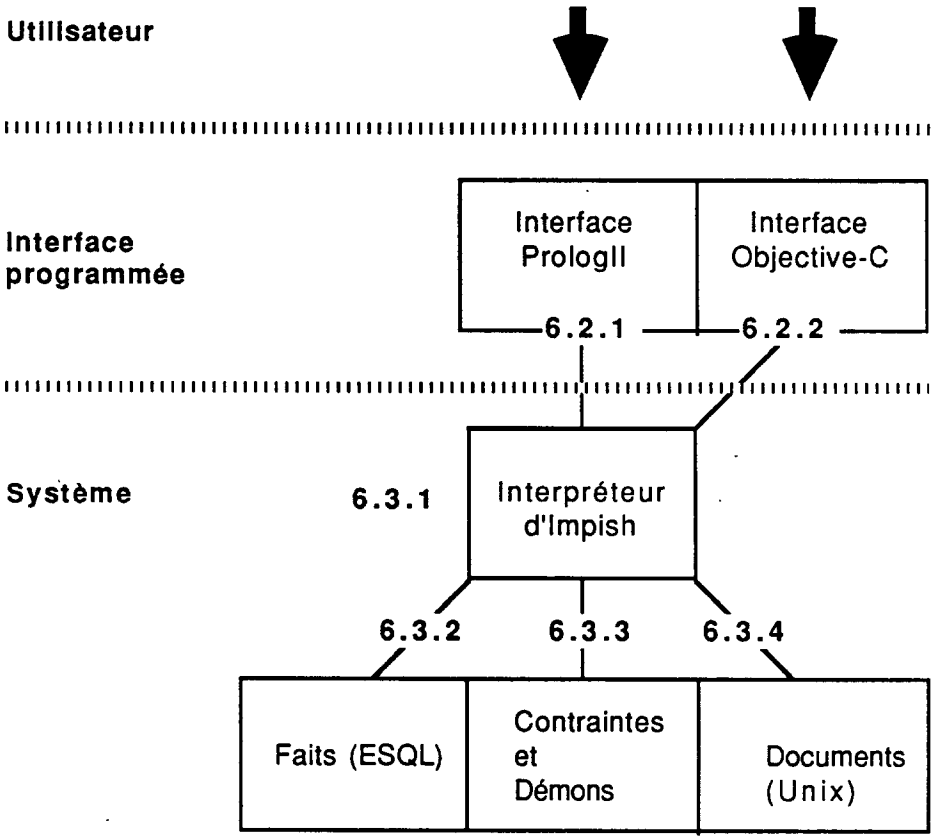


fig. 15 : Le plan du chapitre 6.

6.2 Les interfaces à Impish

Un SI construit avec Impish, est, dans le cadre d'un atelier, le serveur d'outils qui peuvent y accèder, soit en le considérant comme un processus PrologII, soit en tant qu'objet Objective-C.

Le schéma qui suit montre les voies de communication possibles, et les deux prochains paragraphes en détaillent les principes.

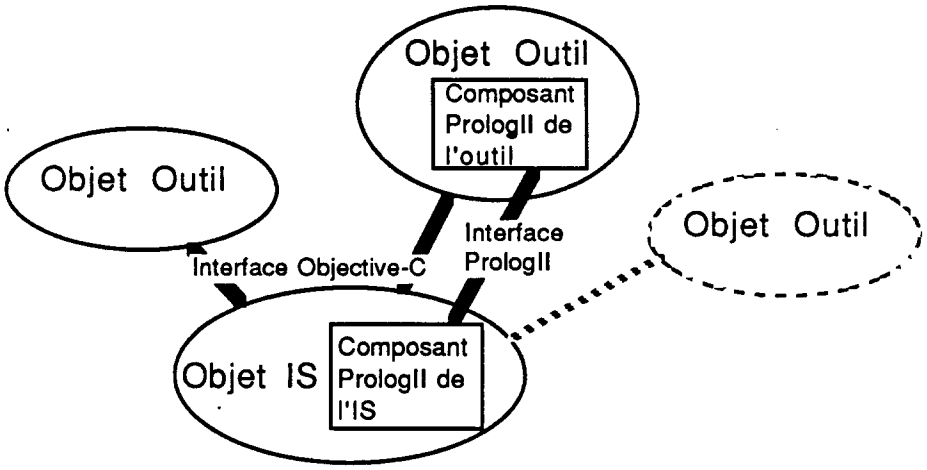


fig. 16 : L'IS dans un atelier

6.2.1 L'interface vers des programmes écrits en PrologII

Un prédicat "activate" permet de lancer une commande Impish à partir de son arbre syntaxique : Un programme prolog qui aurait construit l'arbre syntaxique "t-ree" d'une commande Impish peut dans l'exécuter directement par simple effacement

du prédicat "activate" :

```
activate(t-ree,a-nswer) -> ... ;
```

Toutefois, cette voie d'accès à Impish est extrèmement peu conviviale : elle nécessite de connaitre la définition des arbres de la syntaxe du langage d'Impish, et est uniquement utilisée dans le code de l'interprète d'Impish.

Il est aussi possible d'utiliser un prédicat IS-interprete(q-uery,a-nswer,e-rror) où q-uery est une chaine de caractère exprimant une commande Impish (même type d'interface que le "ex", voir en 6.3).

Cependant, pour rendre Impish accessible plus simplement par des programmes écrits en Prolog, nous avons défini une série de prédicats dont la sémantique est (presque) équivalente aux commandes du langage de manipulation de données d'Impish.

L'expression de ces prédicats s'appuie sur la similitude remarquée précédemment entre un prédicat et une relation :

```
predicat(v-ar1, v-ar2, v-ar3)
```

définit, en Prolog, la structure de la relation de nom "predicat" qui est composée de trois colonnes. On dira que ce prédicat est un MRPT (pour "model relevant prolog term"), puisqu'il réfère une relation du modèle de la BD.

Chacun des prédicats présentés ci-après sont conçus de manière identique : ils vont construire l'arbre syntaxique nécessaire au prédicat "activate" déja présenté, en tirant la sémantique de la requête de l'identifiant d'un MRPT, des valeurs de variables si elles sont affectées, et de leur non

affectation dans le cas contraire.

Nous allons maintenant détailler ces prédicats.

En plus de deux prédicats élémentaires, "db-database" et "db-close-database", plusieurs classes de prédicats ont été implémentées :

6.2.1.1 Recherche dans la base : db-erase

Db-erase travaille sur les faits courants (et non historiques) et le contexte courant.

- La syntaxe de ce prédicat :

  db-erase(*p*)

où *p* est un MRPT, un terme de la forme :

  nom-de-table(x1, ..., xn).

- La sémantique de *p* est interprêtée de la sorte :

.les variables représentent, dans l'ordre défini dans le modèle, les attributs (colonnes) de types non "link". Toutes les colonnes sont sélectionnées. La requête qui va être activée est un :

"select * from nom-de-table <where expression>"

.les variables liées au moment de la tentative d'effacement du prédicat *p* sont prises en compte dans la "where expression" comme des contraintes d'égalités : Si x1, qui représente la colonne "totocolumn", vaut "totostring", alors la "where expression" correspondante est :

… where nom-de-table.totocolumn = "totostring" …

.les variables libres sont unifiées en sortie avec les valeurs correspondantes qui, dans la base, satisfont aux contraintes exprimées par les variables liées.

.le "backtrack" naturel de Prolog continue donc sur les tuples réponses qui sont vus comme l'ensemble des substitutions possibles à la liste des variables. Le comportement normal de Prolog reste ainsi préservé. On notera que le db-erase essaie d'abord d'unifier le prédicat dans le monde Prolog : l'univers des faits est donc en réalité l'union de la base de données et de la base de faits Prolog.

.les valeurs nulles renvoyées par le SGBD sont exprimées par l'unification de la variable concernée avec le terme 'null' (de type Prolog identifiant).

- L'activation de la requête est lancée par le prédicat "ex", présenté plus haut, qui réalise le couplage "physique" entre Prolog et le SGBD.

Nota : L'utilisation du db-erase est délicate : on en verra plus bas le pourquoi, et les outils proposés dans Impish pour en faciliter l'utilisation.

6.2.1.2 Manipulations élémentaires de la base

Ces manipulations portent sur les faits courants (et non historiques) et le contexte courant.

Insertion dans la base : db-insert

- La syntaxe de db-insert est la même que celle de

db-erase.

- La sémantique en est toutefois quelque peu différente :

.le prédicat paramètre est interprêté comme un tuple que l'on veut insérer dans la table correspondant à son identifiant.

.les variables libres sont, en général, considérées comme non ayant des valeurs nulles, au sens des bases de données.

.les variables libres correspondant à des colonnes de type "serial" sont interprêtées comme non précisées, ce qui équivaut à la valeur 0 pour le SGBD, et sont donc incrémentées.

.la valeur des variables liées est prise comme une valeur de colonne :

db-insert(nom-de-table(x1, x2, x3, x4))
{x1 = "toto", x2 = 1, x3 = 2}

est traduit par l'arbre de la commande Impish :

```
"insert into nom-de-table
      values ("toto", "1", "2", NULL)"
```

.l'insertion se fait donc selon la sémantique Impish, c'est à dire avec vérification des contraintes éventuelles (cf. les extensions Impish d'ESQL). En cas de non vérification de ces contraintes, l'effacement du prédicat db-insert se conclut sur un échec.

- L'activation de la commande est exécutée par le prédicat "activate" (voir plus haut), et c'est donc l'arbre de la commande qui est construit par le prédicat db-insert.

Suppression dans la base : db-delete

- La syntaxe de db-delete est la même que celle de db-erase (cf. plus haut).

- Sa sémantique est la suivante :

.le prédicat paramètre est considéré, comme dans le db-erase, comme une restriction de la requête de suppression, aux tuples qui sont unifiables avec ses variables.

.le prédicat :

db-delete(nom-de-table(x1, x2, x3, x4))
{x1 = "toto", x2 = 1, x3 = 2}

supprimera de la base les mêmes tuples que la requête Impish (c'est à dire avec vérification de contraintes) :

"delete from nom-de-table
where <col1> = toto and <col2> = 1 and <col3> = 2"

.en cas de non vérification des contraintes, l'effacement du prédicat db-insert se conclut sur un échec.

- L'activation de la commande est exécutée par le prédicat "activate" (voir plus haut), et c'est donc l'arbre de

la commande (et non le texte) qui est construit par le prédicat db-delete.

Modification dans la base : db-update

- La syntaxe de db-update :

db-update(p, q)

où *p* et *q* sont deux MRPT de même identifiant.

- La sémantique du second prédicat est la même que celle du paramètre de db-delete. Celle du premier est plus spécifique :

.ses variables libres sont considérées comme représentant des colonnes que l'on ne veut pas modifier.

.ses variables liées donnent les nouvelles valeurs de colonnes, si elle sont de type chaine, entier ou réel.

.ses variables liées à un identifiant égal à "null", "Null", ou "NULL" définissent les nouvelles valeurs nulles (au sens du SGBD) .

- La commande préparée est un update Impish, et les contraintes de l'update vérifiées.

- La commande préparée est un arbre passé à l' "activate".

Nous n'avons pas traité le cas des manipulations de

données historiques : ils ne présentent pas vraiment d'intérêt théorique particulier, dans la mesure où écrire des prédicats pour cela revient simplement à combiner les choix réalisés ici (notamment ceux portant sur la sémantique des valeurs nulles) et une paramétrisation étendue aux attributs "temporels" qui ont été présentés en 5.

6.2.1.3 Manipulation de documents

Pour rester homogène avec le db-erase, nous avons conçu les db-insert, delete et update comme des prédicats ne manipulant que des attributs de type différent de "link".

La manipulation de ces derniers est toutefois rendue possible par le db-update-doc que nous présentons ici :

- La syntaxe :

db-update-doc(p, d-oc-type, v-aleur)

où *p* est un MRPT, d-oc-type un identifiant, et v-aleur soit une chaine, soit un terme de la forme :

val(chaine)

- La sémantique :

La variable *p* permet de définir la "where clause" (comme le second paramètre du db-update), d-oc-type permet de retrouver le lien concerné, v-aleur décrit la valeur à affecter. Comme en ImpishSQL, cette valeur peut être soit un "pathname" (auquel cas c'est une chaine), soit le contenu du document (auquel cas c'est un terme comme décrit ci-dessus).

- L'utilisation permet de modifier des liens existants, mais aussi d'en définir des nouveaux, quand il est utilisé conjointement à un db-insert :

```
toto -> db-insert(Tache(x1, x2, "Toto"))
        db-update-doc(Tache(x1, x2, "Toto"),
                      description,
                        val("cette tache est une tache alimentaire"));
```

insère les tuples satisfaisants avec des liens "description" qui contiennent la phrase "cette tache …".

La règle suivante :

```
toto -> db-insert(Tache(x1, x2, "Toto"))
        db-update-doc(Tache(x1, x2, "Toto"),
                      description,
                        "/usr/CETE/bosco/prog/myDoc");
```

insère les tuples et les liens correspondants dont le contenu sera recopié du fichier Unix "/usr/CETE/bosco/…".

L'accès aux "tool-doc", comme définis en ImpishSQL, à partir de Prolog, n'a pas été réalisé.

6.2.1.4 Transactions sous Prolog

Pour permettre une bonne utilisation des commandes Impish, et notamment mettre à profit les contraintes d'un modèle, il est important de pouvoir, en Prolog, définir des transactions.

Aussi avons nous défini les prédicats sans paramètres "db-begin-work", "db-commit-work" et "db-rollback-work", dont

la sémantique est la même qu'en ImpishSQL.

Leur utilisation mérite quelques remarques. Soit la séquence :

```
toto -> db-insert() db-delete() ;
```

que l'on souhaite regrouper au sein d'une transaction. Si l'on veut qu'en cas d'échec de l'un des deux prédicats, la séquence entière soit annulée, on écrira :

```
toto -> db-begin-work db-insert() db-delete() db-commit-work;
toto -> / db-rollback-work ;
```

Si l'on souhaite, au contraire, que tout ce qui est possible soit répercuté, et que la vérification des contraintes "après" (cf. paragraphe sur les contraintes) ne soit pas cause d'échec, on écrira plutôt :

```
toto -> db-begin-work db-insert() db-delete() db-commit-work;
toto -> / db-commit-work ;
```

6.2.1.5 Note sur l'utilisation de "db-erase"

Nous disions en a) que l'utilisation du db-erase était quelque peu délicate, et nous nous expliquons ici :

- L'effacement d'un MRPT dans la base de données doit pouvoir être décidé sciemment par le concepteur du programme Prolog. Il est facile, lorsqu'on écrit un programme, d'insérer ce db-erase aux endroits voulus explicitement. On écrira à volonté :

```
(1) toto -> db-erase(Tache(x1, x2, "Toto"));
```

.où (1) définit un prédicat toto qui sera vrai si il existe, dans la base de faits Prolog étendue à la base de données, un tuple dont la valeur du troisième élément vaut "Toto",

.et où (2) définit un prédicat effaçable dans le seul monde Prolog.

- Le problème apparait lors de l'utilisation de variables pouvant s'unifier à un terme Prolog complexe. Le cas typique est celui du "not" définit en Prolog par l'échec :

```
not(p) -> p / echec;
not(p) -> ;
```

Avec une telle définition du "not" :

```
not(Tache(x1, x2, "Toto"))
```

ne s'effacera que dans le monde prolog, et pas dans la base de faits étendue à la base de données. Pour qu'il en soit ainsi, il faudra écrire :

```
not(p) -> db-erase(p)/ echec;
not(p) -> ;
```

- Prolog et Impish et les hypothèses

Cet interface n'a pas été implémenté, mais sa conception est simple : avec des prédicats start-hypothesis et close-hypothesis, on peut encadrer le travail dans le contexte

hypothétique, pour que la session d'un outil se passe dans celui-ci. Le métainterpréteur utilisé alors est tel que l'interprétation des db-insert, db-delete, db-erase, … se fait selon les processus implémentés par les commandes relatives aux hypothèses et décrites plus haut dans le LM d'Impish.

6.2.2 L'interface vers des programmes Objective-C

6.2.2.1 Motivation

Impish doit être le serveur d'applications écrites en C (il doit donc pouvoir être C-embedded), et en langage orienté objet (dont nous choisirons un exemple judicieux, Objective-C).

L'interface à C ne sera présentée qu'à travers l'interface Objective-C, dont l'aspect conceptuel (traitant des rapports objets/relations) est plus intéressant.

Une validation de cette interface, comme de celle à Prolog, est actuellement en cours par l'intégration d'outils de gestion de projets logiciels sur un système d'information supporté par Impish.

6.2.2.2 La problématique

Nous ne reviendrons pas ici sur les concepts de l'orienté-objet, ni sur la syntaxe d'Objective-C (pour cela, on se réfèrera à [GOLD83] et [COX83]. La lecture de tout ou partie de ce dernier ouvrage pourra être nécessaire à la compréhension des détails d'implémentation).

Nous remarquerons simplement que l'on peut assimiler les faits contenus dans la base comme l'ensemble des parties statiques (les structures de données, les attributs, les "slots") d'objets correspondants :

Considérons par exemple le type d'objet, manipulé par un outil d'aide à la planification, appelé "Tache". Celui-ci a trois attributs : "nom de tache", "durée", "date de début". Il a aussi un ensemble de "méthodes", qui décrivent son

comportement.

Considérons aussi la relation, représentée par une table, "Tache", dont les colonnes sont "nom de tache", "date de début", "date au plus tard", "durée", "nom de l'événement de début".

L'outil de planification, avant de travailler avec ses taches, doit puiser dans le système d'information les valeurs qui lui servent et qui ont été instanciées par ailleurs. Pour cela, il lance une sélection portant sur un sous ensemble des colonnes de la table, et doit instancier les "slots" de ses objets avec les valeurs retournées.

De plus, selon les principes O.O., chaque outil est implémenté sous la forme d'un objet. Il communiquera donc avec le système d'information par échange de messages [COX83].

6.2.2.3 Un SI supporté par Impish vu comme un objet

Créé par un administrateur, le modèle d'un système d'information est instancié et ces instances manipulées par les outils.

Une instance d'objet ("monSI", par exemple) représente le SI manipulé. Pour communiquer avec lui, un outil lui envoi le message :

```
[monSI interpretSel: <select-expression>]
```

pour générer l'exécution d'une sélection, ou bien :

```
[monSI interpretOther: <autre-expression>]
```

pour une autre commande.

Ces appels mettent un objet "flag" dans un état valide ou invalide, qui permet de s'assurer du bon déroulement de la requête.

Ensuite, une expression en Objective-C :

```
uneCollection = [monSI answer]
```

permet de récupérer le résultat dans un objet Objective-C, de type "collection", et qui contient des objets t-uples. Charge reste ensuite à l'outil de récupérer les éléments des t-uples et … d'en faire ce qu'il veut.

Dans ce mécanisme, l'entité de plus petite "taille" qui est échangée entre les deux systèmes est la valeur. La généricité de l'interface réside uniquement au niveau de la gestion automatique du type de ces valeurs au sein de la collection qui contient les différents tuples, et de l'interprétation des valeurs nulles.

6.2.2.4 Points d'implémentation

La solution choisie pour traiter le premier point (la gestion des types) consiste simplement à transformer les types ESQL-C en types C, et non en classes Objective-C qui seraient les classes chaines, entier, … . Nous verrons un peu plus bas qu'une solution plus complexe mais plus homogène est envisageable.

Les valeurs nulles du SGBD, elles, sont traduites par des valeurs réservées, selon leur type.

6.3 Le système : Prolog, langage maître

Ce paragraphe est consacré à l'organisation du monde PrologII dans lequel est implémenté Impish, et aux travaux réalisés pour donner accès, en Prolog, au langage de requête du SGBD natif (accès utilisé par le langage étendu (Impish) que nous avons déja présenté).

6.3.1 Le monde PrologII

Impish utilise le monde Normal de PrologII sous lequel un sous monde "is" est créé qui contient le code PrologII de l'interpréteur d'Impish. Un sous monde "model" y est défini, qui contient la représentation PrologII de la base courante.

Les identifiants de prédicats susceptibles d'exister dans ce monde "model" (attribute, link, hdata,...), sont, pour qu'ils puissent y être reconnus, déclarés sous "is"

6.3.2 Interface "physique" entre Prolog et ESQL

Un prédicat "ex" permet d'exécuter une requête sous forme de chaine de caractères.

L'accès au langage du SGBD à partir de Prolog est possible si l'on utilise l'environnement dans lequel ils baignent tous les deux, à savoir Unix, et leurs interfaces au langage C.

Pour réaliser ce couplage, notre système maintient actifs deux processus :

- un processus Prolog,
- un processus qui gère l'accès à la base de données, ESQL-C.

Pour permettre l'accès, par un programme Prolog, aux données de la base, il faut lui donner la possibilité de communiquer avec ce deuxième processus.

Pour cela, un prédicat évaluable, "ex", manipule trois variables :

- l'expression de la requête (variable de type chaine, en entrée), qui a été construite au préalable,
- la réponse à la requête (variable liste en sortie),
- le code d'erreur éventuelle (variable entière en sortie).

6.3.2.1 Expression de la requête

On discernera deux types de requêtes : les sélections (qui attendent une réponse complexe constituée de tuples, ou bien un rejet) et les autres (qui n'attendent, en fait, qu'une accréditation ou un rejet).

Le langage dans lequel sera exprimée cette requête sera ici celui du SGBD choisi comme base de notre système, ESQL Informix.

6.3.2.2 Activation d'une sélection

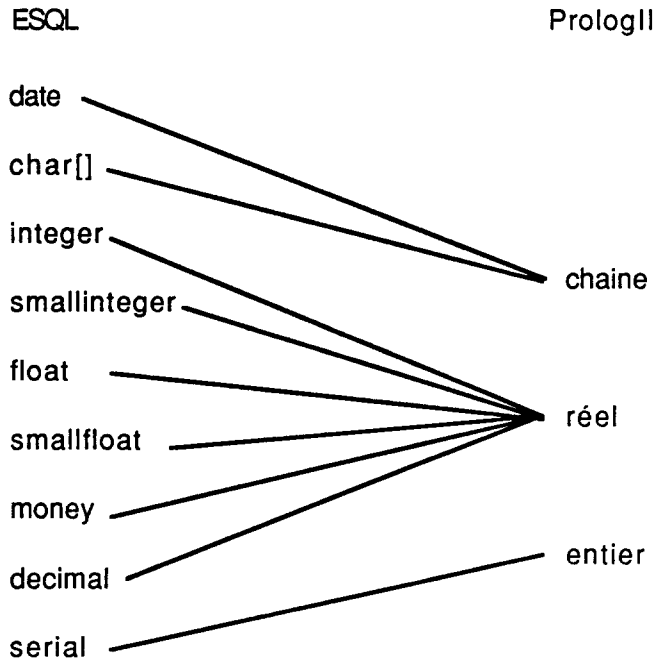
L'activation d'une sélection consiste en la séquence suivante :

- transmission du processus Prolog au processus

ESQL de la chaine représentant la requête, via un programme C,

- activation du mécanisme ESQL Informix qui l'exécute et retourne :
  .le nombre de tuples réponse,
  .la longueur de chaque tuple,
  .le type (ESQL) de chaque élément du tuple et,
  .un à un, ces tuples (ou un code d'erreur en cas d'échec),

- récupération au niveau de Prolog, via C :
  .des deux indications sur les longueurs des réponses et les types, puis,
  .en parallèle avec l'autre processus, des tuples, avec conversion des types ESQL en types Prolog,



Nota : la faible utilisation du type Prolog "entier" est due à son incapacité à inclure les entiers négatifs. Cette lacune sera comblée par les versions compilées de PrologII.

fig. 17 : Table des correspondances de types

- récupération éventuelle d'un code d'erreur généré par le SGBD.

6.3.2.3 Activation d'une autre commande

Elle consiste aussi à un échange interprocessus, mais avec deux différences essentielles :

- il n'y a, en retour de ESQL à Prolog, qu'une accréditation,
- ESQL réalise des effets de bords sur la base.

Les étapes du mécanisme précédent qui consistaient en la réception des réponses n'existent donc plus, et la récupération de l'accréditation est semblable à celle d'un code d'erreur.

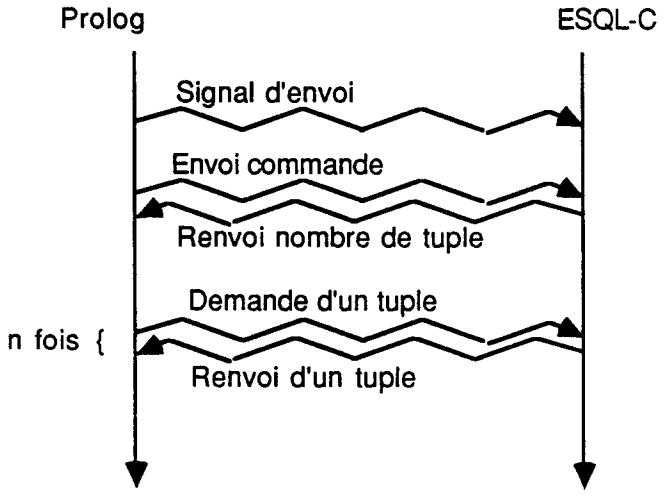


fig. 18 : Echanges interprocessus

Le mécanisme décrit ci-dessus est implémenté à la fois en Prolog et en C (ou, plus exactement, en ESQL-C, qui est un préprocesseur autorisant l'inclusion, dans un programme C, de commandes précompilables permettant de récupérer le résultat d'une activation de commande ESQL dans des structures C).

- Le programme Prolog décrit la logique du mécanisme. On en trouvera un extrait ci-après :

```
execute-sql-select(c-ommand, a-nswer,l-length,t-length,l-type,e-rror) ->
        Send-sql-select
        send-sql-command(c-ommand)
        Receive-sql-error(e-rror)
        receive-sql-answer(a-nswer,l-length,t-length,l-type);
```

/extrait du fichier coupling.pro en annexe/

On remarquera les prédicats, qu'une convention nous fait écrire avec une lettre majuscule et qui sont de nouveaux prédicats évaluables ajoutés au Prolog de base via deux fichiers, "prouser.c" [PROL87] , où ils sont déclarés pour C, et un fichier de règles Prolog, où ils le sont pour Prolog.

```
...
case 504:
     send_sql_select(err, err_nb);
     break;
/* extrait de prouser.c */
...
```

```
Send-sql-select     ->     /?504  /?99 ;
```

"et prédicat évaluable n°504."

- Des programmes C définissent, par des fonctions, le corps de ces prédicats. Ils incluent un certain nombre de fonctions C propres à l'environnement Prolog et qui réalisent les entrées/sorties (get_integer, put_integer, … [PROL87]).

En se rapportant à l'annexe, on remarquera, dans le fichier "coupling.ec" (fonction "send_sql_select"), l'usage simultané des fonctions d'interface et du préprocesseur ESQL-C (qui interprète les $ prepare, $ declare, … [ESQL86]).

- Le traitement des conversions de types ESQL en types Prolog est réalisé en deux étapes :

A l'activation de la requête, le résultat est sauvé, sous forme de chaine de caractères, dans une variable globale de type tableau.

Au moment de la réception de ce résultat (de la lecture par le prédicat "receive-sql-select", voir plus haut dans le code Prolog), ces chaines sont retransformées en types Prolog. On peut lire le détail de ces transformations dans "coupling.ec", en annexe, dans la fonction "put_value".

6.3.3 Interface entre Prolog-Interpréteur et Prolog-Outil de représentation de la connaissance

6.3.3.1 Prolog, outil de modélisation

Considérons un système d'information, défini avec un

modèle spécifique, et supporté par Impish :

Un ensemble d'outils (des "applications" selon les termes consacrés par les langages de quatrième génération) peuvent l'utiliser. Si l'on s'intéresse à plusieurs outils écrits en Prolog, il va falloir, en tenant compte des spécificité de Prolog (sa - faible - modularité implémentée par le concept de "monde" [PROL87], son mécanisme de recherche des règles applicables au moment du backtrack, …), les faire vivre contemporainement. De plus, le corps des contraintes sont autant de programmes Prolog qu'il faut gérer de la même façon.

Si l'on suppose que ces programmes sont écrits séparément les uns des autres, des risques de réécriture de règles existent. A charge du système que nous élaborons de gérer ces risques.

De plus, on a vu que l'utilisation de certains prédicats Prolog - Impish (cf. le db-erase et les commentaires correspondants) était délicate, et il nous parait intéressant de l'automatiser partiellement.

Nous appelons ce traitement de programmes Prolog, destinés à s'exécuter sur une base Impish, l'intégration.

Nous détaillons, dans les lignes qui suivent, ce mécanisme d'intégration, qui a donc un double but :

- charger des programmes indépendants dans des modules différents,

- permettre, si cela est souhaité, l'effacement des prédicats MRPT dans la base de données Impish.

a) La modularité :

Pour identifier une règle comme appartenant à un programme donné, plusieurs solutions existent :

- le renommage des règles :

```
toto -> titi(x);
```

devenant :

```
toto12 -> titi12(x) ;
```

pour le programme numéro 12, qui pose différents problèmes :

.si titi1 est un identifiant de règle pour le programme numéro 2, on aura là un autre titi12 ...,

.pour les prédicats prédéfinis, même s'il est vrai que cela peut être testé à l'intégration.

- l'encapsulation, sous la forme de triplets :

```
<clause12, toto, titi(x).nil> -> ;
```

qui est ensuite "métainterprété" (par la commande *run* que nous verrons ci après). Le problème des identifiants de règles ne se pose plus, et les prédicats prédéfinis ou partagés sont pris en compte par le métainterpréteur, dont nous verrons ausi les autres avantages.

b) L'extension de la base de faits :

Dans les deux cas, le prédicat "db-erase" peut être

inséré là où un MRPT, ou une variable pouvant s'unifier à un MRPT (cf. l'exemple du not plus haut), est identifié, et ce de manière systématique (dans la commande *integrate*), ou pas (dans la commande *integrate local*) qui n'insère pas les db-erase.

C'est la deuxième solution que nous choisirons.

c) La sauvegarde du comportement de Prolog

Nous avons déja dit que nous souhaitons laisser au programmes intégrés un comportement analogue à leur comportement naturel, à l'exception près de l'extension de la base de faits.

L'insertion du db-erase, telle que nous la pratiquons, conserve à notre effacement dans la base de données les mêmes qualités qu'un effacement dans le monde Prolog : il y a échec si il y a absence de tuple.

Notre étude, dans le chapitre 3, a toutefois mis en évidence une autre difficulté, le traitement du "cut", le "/" de PrologII. Notre solution consiste à traiter le premier "/" rencontré dans une règle par sa substitution par son code interne : "0", comme dans l'exemple suivant :

```
toto -> titi /;
```

devient :

```
<clause12, toto, titi.0.nil> -> ;
```

Quant au reste de la clause, il doit être interprété de manière déterministe : Pour qu'il en soit ainsi, nous avons donc défini un prédicat "deterministic" qui sera

métainterprété, utilisé comme dans l'exemple :

```
toto -> titi / tutu / tata;
```

devient :

```
<clause12, toto, titi.0.tutu.deterministic(tata.nil).nil> -> ;
<clause12, deterministic(x), erase(x).0.nil> -> ;
```

où le "erase" efface une liste de prédicats selon le mode (LOCAL ou pas) choisi lors de l'intégration. Nous tenons à souligner que la métainterprétation du prédicat coupe choix était, à notre connaissance, jusqu'à ce jour non réalisée [MOLL87], et que notre réalisation est le sujet d'une publication à paraître [BOSC88b] : le succès de cette solution est dûe, en fait, à la capacité de PrologII de gérer le "cut" dans une liste de termes, chose que ne sait pas faire un grand nombre d'autres Prolog.

Exemple.

Soit le programme (qui représente une contrainte pour une partie d'un modèle de gestion de projets logiciel, avec Task et ResWorkTask comme MRPT) :

```
Task-ResWorkTask-min1(x1) ->
    Task(x1,x2,x3,x4,x5,x6) /
    ResWorkTask(y1,x1,y3,y4,y5);
Task-ResWorkTask-min1(x1) ->;
```

Son intégration donne :

```
<clause14, deterministic(a), erase(a).0.nil> ->;
<clause14,Task-ResWorkTask-min1(a),db-erase(Task(a,b,c,d,e,f)).0.
```

```
<clause14, Task-ResWorkTask-min1(a), nil> ->;
"(on remarque qu'ici le deterministic n'est pas utilisé, et qu'en fait"
"il ne l'est que très rarement)"
```

6.3.3.2 Exécution de programmes Prolog

a) La métainterprétation

Ce mécanisme est représenté dans [STER84] par les règles suivantes :

```
meta(true) ->;
meta((Q1,Q2)) -> /  meta(Q1) meta(Q2);
meta(Q) -> clause(Q, Body) meta(Body);
```

Toute sa puissance est en fait fournie pas les mécanismes de backtrack et d'unification de Prolog.

Les possibilités d'extensions existent dans la définition du **meta(true)**. C'est à ce niveau que l'on peut définir, sans changer quoi que ce soit au programme d'origine, des fonctionnalités particulières d'interprétation : on pourra aisément implémenter des mécanismes de type "explicatifs", ou "query-the-user", ..., modifier le mécanisme d'inférence (faire par exemple du chainage avant, ...), modifier les mécanismes de contrôle, définir des outils (dévermineurs, traceurs, ...).

Par exemple, voilà, tel qu'il est implémenté dans Impish, le "métatraceur" qui permet de tester des programmes Prolog intégrés sur un SI :

```
execute-program-list1(c-lause-id,nil)  ->;
execute-program-list1(c-lause-id,p.l)  ->
```

```
        execute-program-list1(c-lause-id,l);

execute-program1(c-lause-id,p)  ->
        <c-lause-id,p,q>
        outl(p)
        if-then-else(contains(q,0),
                        /contains(l,x) est vrai si l contient x/
                        special(q,l-av,l).
        /special (q,l-av,l) extrait de q la liste l-av des termes apparaissant
        avant 0 et l, qui est la liste des listes, entrecoupée de 0, des séquences de
        termes différents de 0 dans le reste de q :
        special("a".0."b"."c".0."d".nil,"a".nil,("b"."c".nil).0.("d".nil).nil)/
                        execute-program-list1(c-lause-id,l-av).
                        0.
                        execute-program-list1(c-lause-id,l),
                        execute-program-list1(c-lause-id,q));
execute-program1(c-lause-id,erase(p))  ->
        erase(p)
        outl(erase(p));

execute-program1(c-lause-id,db-insert(p))  ->
        db-insert(p)
        outl(db-insert(p));
execute-program1(c-lause-id,db-delete(p))  ->
        db-delete(p)
        outl(db-delete(p));
execute-program1(c-lause-id,db-update(p,q))  ->
        db-update(p,q)
        outl(db-update(p,q));
execute-program1(c-lause-id,erase(p))  ->
        execute-program1(c-lause-id,p)
        outl(p);

execute-program1(c-lause-id,list-of(a,b,c,d))  ->
```

```
        list-of(a,b,execute-program1(c-lause-id,c),d)
        outl(list-of(a,b,c,d));
execute-program1(c-lause-id,draw-tree(a)) ->
        draw-tree(a);
execute-program1(c-lause-id,p) ->
        predefined(p)
        p
        outl(p);
```

On notera que les prédicats prédéfinis et spéciaux (db-xx,draw-tree,…) sont traités séparément.

b) L'appel d'un programme

Cet appel est réalisé par la commande Impish *run*, à laquelle sont passés en paramètres :

- le prédicat "point d'entrée" du programme (le "to-begin", en fait),

- le nom du métainterpréteur choisi pour cette exécution (nom qui est codé de manière interne : dans l'exemple ci-dessus, "metatraceur" est associé à "execute-program1"). Si ce nom n'est pas donné, c'est le métainterpréteur courant (le dernier choisi, ou, à défaut, celui utilisé pour la vérification des contraintes), qui est utilisé :

run metatraceur *of* toto

6.3.3.3 Synthèse des accès à un SI offerts par notre

structure

Pour se résumer, les différents niveaux d'accès à un SGBD, à partir de programmes Prolog, et qui sont offerts par Impish, sont les suivants :

1- accès à ESQL Informix, par le prédicat "ex",

2- accès à ImpishSQL, par le prédicat "activate",

3- accès à Impish par le IS-interprète et des prédicats "built-in" (cf. 6.2), les db-xx, qui sont utilisés par le programmeur Prolog, comme des prédicats systèmes, et dont la liste suit :

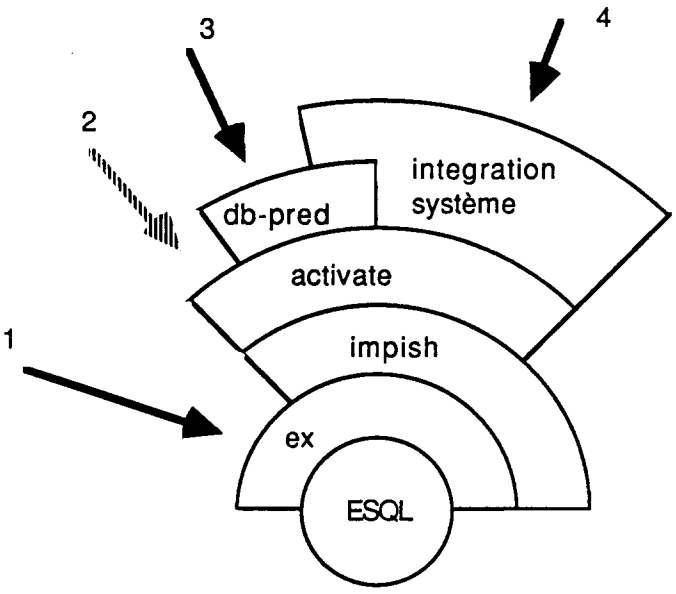
db-database,
db-close-database,
db-erase,
db-insert,
db-update,
db-delete,
db-update-doc,
db-begin-work,
db-commit-work,
db-rollback-work.

4- accès à Impish par des prédicats "built-in", avec intégration et exécution "système" par les commandes Impish :

integrate,
run.

Le premier niveau est intéressant pour des applications sur le SQL natif. Le deuxième est un niveau interne, peu utilisable directement. Les deux derniers sont les plus intéressants et sont utilisés par les outils "Systèmes Experts" de notre atelier de gestion de projets logiciel [JENK87].

fig. 19 : Les différents niveaux d'interfaces Prolog - Base de Données



6.3.4 Interface entre Prolog et le SGF

6.3.4.1 Introduction

Les commandes d'Impish, lorsqu'elles manipulent des documents (que ce soient des "liens" ou des "tool-doc"), génèrent des effets de bords (création, suppression, modification de fichiers) sur le monde des fichiers Unix. Cela est réalisé par une provédure C qui inclut elle même des appels à des fonctions du SGF.

6.3.4.2 Prédicats Prolog à effet de bord sur les documents

La liste qui suit est extraite du fichier "prouser.c" [PROL87] :

```
                    /**********************/
                /*      procedures impish      */
                /**********************/

                create_base(err, err_nb);
                ...
                drop_base(err, err_nb);
                ...
                create_ref_doc(err, err_nb);
                ...
                drop_ref_doc(err, err_nb);
                ...
                create_tool_doc(err, err_nb);
                ...
                drop_tool_doc(err, err_nb);
                ...
                store_tool_doc(err, err_nb);
                ...
```

Ces fonctions correspondent à des "built-in" prédicats Prolog.

Un extrait de corps de fonction C :

```
create_base(err, err_nb)
int *err, *err_nb;
{
    char  base[STRSIZE],
          ref_dir[STRSIZE],
          tool_dir[STRSIZE],
          user_dir[STRSIZE],
          temp_dir[STRSIZE];

    get_string(1, base, err);
    /* interface PrologII/C */

    strcpy(base, (cheminlocal("IS",base)));
    mkdir(base, 0777);

    strcpy(ref_dir, base);
    strcat(ref_dir, "/REF_DOC");
    mkdir(ref_dir, 0777);

    ...

    strcpy(temp_dir, base);
    strcat(temp_dir, "/TEMP");
    mkdir(temp_dir, 0777);
}
```

6.3.4.3 Recherche de documents

Des documents peuvent être recherchés (lors d'une sélection), et, on l'a vu, leur liste retournée.

Un algorithme calcule le nom des documents souhaités, en se basant sur la clé d'une entité pour un "ref_doc", et sur l'identifiant et la version du document pour un "tool_doc". Le document est alors repéré, son chemin d'accès recomposé, et une copie réalisée sous le répertoire .../USER de la base Unix.

Le renvoi des chemins d'accès de ces copies est assuré, de C à Prolog, par un échange de valeurs entre les deux processus. Des mécanismes analogues sont décrits en détails dans le paragraphe consacré à l'interfaçage interne entre Prolog et ESQL-C.

6.4 Conclusion

Le projet IMPW, et les outils de gestion de projets logiciels qu'il réalise, utilise à la fois l'interface vers PrologII et l'interface Objective-C.

PrologII s'est avéré un choix judicieux, notamment pour certaines de ses fonctionnalités :

- son interprétation du code interne du "cut" dans une liste de terme (qui permet la métainterprétation du coupe-choix, très difficile à réaliser sinon),

- ses interfaces efficaces avec le langage C,

- son prédicat "freeze" utilisé pour l'analyse syntaxique des langages d'Impish,

ont rendu nos travaux plus aisés.

Objective-C a aussi beaucoup apporté au niveau de la définition des communications entre outils et système d'information.

Les problèmes posés par le système se situent sans doute au niveau de l'efficacité de l'interprète. A cette critique, fort justifiée, nous répondrons que notre objectif était de démontrer la faisabilité d'implémenter certains principes et concepts, pas de les optimiser. De plus, ce qui a été écrit en PrologII, peut, en partie, être réécrit dans un langage plus efficace, et le reste pourra être "compilé" dès que PrologII sera compilable …. Il est certain qu'alors les performances seront acceptables.

Sur le plan technique, on peut dire qu'Impish est le

royaume des interfaces et des interconnexions entre systèmes … Sa complexité en est, de fait, assez grande, mais l'utilisation forte des potentialités des différents composants d'Impish en fait aussi sa richesse, et, seule, a permis l'implémentation assez rapide des nombreux concepts ajoutés aux concepts des SGBD relationnels.

Nous espérons aussi que l'utilisation d'un SI basé sur Impish, dans le cadre d'un logiciel "grandeur" réelle, montrera le bien fondé des différents choix techniques, et notamment de la technique de couplage que nous avons réalisé : bien souvent critiquée (parce que peu efficace, peu "intégrée"), nous pensons avoir démontré son intérêt et son "utilisabilité" réelle, d'autant plus que nous n'avons pas fait porter nos efforts sur l'optimisation de ce couplage (l'appel tuple à tuple que fait le **db-erase** n'est pas ce que l'on fait de mieux en la matière …).

Enfin, le choix de la métainterprétation, acceptable pour la vérification des contraintes, est toutefois criticable pour des outils "systèmes experts" de taille considérable, dont la rapidité d'exécution sur Impish est très, très faible …

7. Conclusion générale et perspectives pour Impish

7.1 Les objectifs atteints

Si l'on se réfère au paragraphe 2.7 où nous définissions nos objectifs, il apparait, à la fin de cette étude, que l'essentiel en a été atteint :

Les types d'informations manipulés par Impish sont multiples : faits, faits historiques, faits hypothétiques, documents, sont autant de concepts de base qui enrichissent les capacités de modélisation des SGBD relationnels. On se rapportera au tableau de la conclusion du chapitre 5. pour en avoir une représentation complète et synthétique.

Les contraintes, définies et utilisées par le système grâce à l'extension PrologII, apportent leur contribution au contrôle de l'intégrité des informations que l'on vient de citer. Le couplage avec le moteur logique permet aussi de doter Impish de mécanismes de recherche d'informations récursifs et complexes.

Notre volonté d'ouverture du système vers des langages traditionnels et de l'Intelligence Artificielle a aussi atteint son but : Impish permet l'accès aux données qu'il manipule à des programmes Objective-C, et PrologII. Ces propriétés sont actuellement validées par son utilisation dans le projet d'Atelier de Gestion de Projets Logiciels IMPW. On trouvera d'ailleurs dans l'annexe numéro 2 un modèle Impish de cet atelier.

Nous rappellerons enfin la compatibilité SQL de nos langages d'accès au système.

7.2 Les approndissements souhaitables

Deux approfondissements important doivent, à notre sens, compléter ces travaux :

Le premier consiste à implémenter dans l'environnement d'Impish un outil interactif qui s'appuierait sur les concepts présents dans l'annexe. Outre l'enrichissement fonctionnel du système lui-même, cette extension permettrait la mise en place sur Impish d'un interface complètement orienté objet et viendrait ainsi compléter les concepts supportés par notre structure d'accueil.

Le second, très important aussi, consisterait à définir aussi formellement que possible des règles d'extraction de la connaissances, dans la continuité de la méthode ébauchée en 4., pour que la définition des contraintes soit facilité : le projet IMPW montre aujourd'hui que c'est là un besoin crucial et mal satisfait par les méthodes connues jusqu'alors.

Nous espérons que ces travaux, dans la continuité et la dynamique de ceux présentés ici, pourront se développer, et qu'ils conduiront à de nouvelles thèses.

7.3 Les perspectives pour Impish

L'avenir proche, c'est tout d'abord la validation complète du système - et sans doute son amélioration - grâce à son utilisation dans le projet ESPRIT IMPW : une douzaine d'outils, constituants d'un atelier de getsion de projets, sont ou vont être intégrés à un système d'information, au modèle complexe et supporté par Impish.

Ensuite, Impish, conçu *a priori* pour le Génie Logiciel, sera, nous l'espérons, utilisé dans d'autres domaines.

Enfin, et parce qu'il apparait que le seul langage Prolog ne suffit pas à la réalisation de systèmes experts, mais que des "générateurs" sont nécessaires, nous pensons qu'il sera fort appréciable d'intégrer l'interface PrologII à un générateur basé, lui aussi, sur PrologII, et développé dans nos laboratoires depuis quelques années.

Impish se situe dans le cadre d'une approche d'intégration. D'autres recherches sont actuellement menées qui pourraient suggérer des évolutions du système, comme par exemple les travaux de l'ECRC [BOCC87] sur les extensions par des capacités déductives du modèle relationnel, de l'Université de Nice [LETH86][MIRA88] sur les extensions de ce modèle aux objets encapsulés, ou bien encore ceux de l'équipe du GIP-Altaïr [BANC87] sur l'extension aux bases de données de langages orientés objets.

Références bibliographiques

[ALLE85] Allez F., Boi L., Bosco M., "PATRE, un langage pour la modélisation des données", *Journées BIGRE : Nouveaux langages pour le Génie Logiciel*, Evry, 1985

[ALLE86a] Allez F., Boi L., Bosco M., "Des outils orientés systèmes d'information dans le poste de pilotage de Concerto", in [CONC86]

[ALLE86b] Allez F., Boi L., Bosco M., De La Motte Collas Y., Benoît S., "PATRE : le poste de pilotage de l'atelier Concerto", *TSI*, à paraitre

[ANDR84] André E., Moreau B., Rougeot B., "Vers un atelier flexible et intégré de logiciel, le Projet CONCERTO", *L'Echo des recherches*, 1984

[ANDR86] André J., Perrin p., "Augmentation de la sémantique du modèle relationnel à partir du modèle objet", *MBD n°4*, septembre 1986

[BANC87] Bancilhon F., "Présentation du GIP ALTAïR", in [BDIA87]

[BDBC87] Des Bases de Données aux Bases de Connaissances, Ed. P.S.I., Paris, 1987

[BDIA87] Bases de données et Intelligence Artificielle, Ed. FIRTECH-Masi, Paris, 1987

[BOCC87] Bocca J., Decker H., Nicolas J.M., Vieille L., Wallace M., "Some step towards DBMS based KBMS", in [BDIA87]

[BOEH83] Boehm B.W., "Les facteurs du coût du logiciel", *TSI*, 1983

[BOGO83] Bogo G., Richy H., Vatton I., "Proposition pour la manipulation de documents", *Rapport Tigre n°3 - Bull*, Grenoble, 1983

[BOOC82] Booch G., "Object Oriented Design", *lettres Ada n°3*, USA, avril 1982

[BOSC85] Bosco M., "Contribution aux spécifications du poste de pilotage de l'atelier de génie logiciel Concerto", *Rapport de DEA Université Paul Sabatier*, Toulouse, juin 1985.

[BOSC88a] Bosco M., Gibelli M., Terranova B., "Objective-C pour une gestion orientée objets de fichiers Unix", *Rapport interne CETE en préparation*, Aix en Provence

[BOSC88b] Bosco M., Gibelli M. "De la métainterprétation du prédicat coupe choix en PrologII", *à paraitre*

[BOUB85] Boubenider Y., "Un système interactif d'aide à la planification de projets logiciels", *Thèse de docteur ingénieur ENSAE*, Toulouse, 1985

[BRAC77] Brachman R.J., "What's in a concept : structural foundations for semantic networks", *International Journal of Man-Machine Studies*, Vol. 9, mars 1977

[BUBE77] Bubenko J., "The temporal dimension in information processing, in *Architecture and Models in Database Management*, Ed. North Holland, 1977

[CARB70] Carbonell J.R., "An artificial intelligence approach to computer-aided instruction", *IEE Transactions on Man-Machine Systems*, 1970

[CARM87] Carmo J., Sernadas A., "A temporal logic framework

for a leyered approach to systems specification and verification", *TAIS*, Sophia Antipolis (F), 1987

[CASE87] CASE'87 : Computer-Aided Software Engineering, Cambridge, Massachusetts, 1987

[CAZI84a] Cazin J., Jacquart R., Michel P., "Manuel de référence F1", *Doc. Onera CERT*, Toulouse, juillet 1984

[CAZI84b] Cazin J., Jacquart R., Michel P., "Dévelopments sur le système d'information de Concerto", *Doc. Onera CERT*, Toulouse, décembre 1984

[CAZI85] Cazin J., Jacquart R., Verfaillie G., "Une expérience d'utilisation du formalisme F1 : modélisation du système de gestion d'objets de la structure d'accueil Emeraude", in *Bigre n°45 : Nouveaux langages pour le Génie Logiciel*, Evry (F), octobre 1985

[CHEN76] Chen P.P., "The entity-relationship model : towards a unified view of data", *ACM TODS 1*, 1976

[CHRI83] Chrisment C., Crampes J.B., Zurfluh G., "The BIG Project", *ICOD - 2*, Cambridge, Massachusetts, 1983

[CHRI85] Chrisment C., Crampes J.B., Zurfluh G., "Bases d'informations généralisées", Ed. Dunod, Paris, 1985

[COLM83] Colmerauer A., Kanoui H., Van Caneghem M., "Prolog, bases théoriques et développements actuels", *TSI*, 1983

[COLM87] Colmerauer A., "Notes sur Prolog III", in [ESPR87]

[CONC86] CONCERTO : Actes des Journées de synthèses, Ed. CNET, Perros Guirec, 1986

[COPE84] Copeland G.P., Maier D., "Making SMALLTALK a Database System", *ACM SIGMOD*, 1984

[CORN87] Cornily J.M., "Couplage Prolog/Bases de Données : comparaison de trois approches classiques et d'une approche parallèle", in [BDBC87]

[COX83] Cox B.J., "The Message/Object Programmin Model", *IEEE - Softfair*, USA, juillet 1983

[CUPP86] Cuppens F., Demolombe R., "A prolog-relational DBMS interface using delayed evaluation", in [LPDB86]

[DELA86] De La Motte Collas Y., Benoît S., "La méthode Prysme" in [CONC86]

[DELI79] Deliyanni A., Kowalski R. A., "Logic and semantic networks", *CACM 22*, 1979

[DONZ83] Donz P., "FOLL : Un extension de Prolog", *Documentation CRISS*, Grenoble, novembre 1983

[DOYL82] Doyle J., "A truth maintenance system", *Artificial Intelligence 12*, 1982

[DUBO87] Dubois D., "Théorie des possibilités : quelques applications", in *Séminaire sur la logique floue - Documentation LSI UPS*, Toulouse, janvier 1987

[ESPR87] ESPRIT'87 Achievements and Impact, Ed. CEC & North Holland, Bruxelles, 1987

[ESQL86] ESQL/C-Informix, Manuel de référence, *INFORMIX*

Software, Inc., USA, février 1986

[FAHL75] Fahlman S.E., "A system for representing and using real-world knowledge", *MIT Artificial Laboratory Thesis Progress Report*, 1975

[FAIR87] Fairley R. E., "Database Requirements for Comprehensive CASE Support Environments", in [CASE87]

[FAJO88] Fajon M., Corby O., "ERASME", *Avignon 88*, Avignon (F), 1988

[FEIG71] Feigenbaum E.A., Buchanan B.G., Lederberg J., "On generality and problem solving : a case study using the dendral program", *Machine Intelligence 6*, 1971

[FERN87] Fernstroëm C., Paris J., Doize M.S., "PIMS Information System Archtecture", *Rapport PIMS Cap Sogeti Innovation*, Grenoble, 1987

[FONT84] Fontaine A.B., Hammes P., "UNIX : Système et environnement", Ed. Masson, Paris 1984

[FOUC82] Foucault O., "Modèle et outil pour la conception des SI dans les organisations", *Thèse de Doctorat*, Nancy, juin 1982

[FRAN87] Frank A.U., Position Paper in [CASE87]

[FROS85] Frost R.A., "Using semantic concepts to characterise various knowledge representation formalisms", *The computer Journal 26*, 1983

[FROS86] Frost R.A., "Introduction to Knowledge Base Systems", Ed. Collins, Londres, 1986

[GALL84] Gallaire H., Minker J., Nicolas J.M., "Logic and databases : a deductive approach", *ACM computing Surveys 16*, 1984

[GARD86] Gardarin G. & al, "SABRINA : un SGBDR issu de la recherche", *TSI 5*, novembre 1986

[GARD87] Gardarin G., Simon E., "Bases de données déductives : langages de règles et récursivité", in [BDIA87]

[GIAN85] Giannesini F., Kanoui H., Pasero R., Van Caneghem M., "Prolog", Ed. InterEditions, Paris 1985

[GL86] Génie Logiciel, Revue, numéro 4, avril 1986

[GOLD83] Goldberg A., Robson D., Smalltalk-80, Ed. Allison Wesley, 1983

[HEND75] Hendrix G.G., "Expending the utility of semantic networks through partitioning", *Fourth IJCA*, Tiblis, 1975

[HURS86] Hurst R.S., "SPMMS - Information structures in software management", *Software Engineering Journal*, janvier 1986

[ILIN86] Iline H., "Un environnement orienté objet en Prolog : LAP", *CIIAM 86*, Marseille, 1986

[JACK83] Jackson M., "System Development", extraits, Ed. Prentice Hall, New Jersey, USA, 1983

[JENK87] Jenkins J., Verbrüggen R., Bosco M., "Integrated Management process Workbench (IMP Workbench) : Intelligent assistance for the Software Project Manager", in [CASE87]

[KLOP83] Klopprogge M., "TERM, an approach to include the time dimension in the ER model", in *Entity Relationship Approach*, ED. Chen, 1983

[KOWA75] Kowalski R.A., "A proof procedure using connection graphs", *JACM 22*, 1975

[LANG85] Lang B., "The Virtual Tree Processor, *Documentation Concerto - CNET*, décembre 1985

[LATO86] Latour P., "La controverse entre SGBD classique et relationnel n'est elle pas un faux débat ?", *MBD n°2*, janvier 1986

[LAUR82] Laurières J.L., "Représentation et utilisation des connaissances", *TSI*, 1982

[LEDI86] Le Dizes J.M., "PHIDIAS, manuel de référence", *Doc. CETE*, Aix en Provence, 1986

[LEMO73] Le Moigne J.L., Les systèmes d'informations dans les organisations, Ed. P.U.F., 1973

[LETH86] Le Thanh N., "Isodépendances et modèles B-relationnel", *Thèse d'Etat de l'Université de Nice*, avril 1986

[LEWI32] Lewis C.I., "Alternative systems of logic", *The Monist 42*, 1932

[LICH87] Li Chu-Min, Barthes J.P., "XINHUO : un système réalisant l'intégration d'un KBMS et de la programmation orientée objet", in [BDBC87]

[LOPE83] Lopez M., Palazzo J., Velez F., "The TIGRE Data

Model", *Documentation IMAG*, Grenoble, novembre 1983

[LPDB86] Workshop on integration of logic programming and databases, *ESPRIT Project 530*, Venise, 1986

[LUND82] Lundberg B., "An axiomatization of events", *BIT 22*, 1982

[MACC69] Mac Carthy J., Hayes P.J., "Some philosophical problems from the standpoint of artificial intelligence", *Machine Intelligence*, Ed. Edinburg University Press, New York, 1969

[MACC79] Mac Call J.A., Software Quality Measurement, USA, 1979

[MACD80] McDermott D.V., Doyle J., "Non monotonic logic I", *Artificial Intelligence 13*, 1980

[MELL87] Meller A., Cornille J.M., Ruhla J., "Le système expert de maintenance PEDRO", in *Les systèmes experts et leurs applications*, Avignon, mai 1987

[MINS75] Minsky M., "A framework for representing knowledge", in *The Psychology of Computer Vision*, Ed. McGraw-Hill, New York

[MIRA86a] Miranda S., Busta J.M., "L'art des bases de données", 2 tomes, *Eyrolles*, 1986

[MIRA86b] Miranda S., "Promenade avec Campus", *MBD n°2*, janvier 1986

[MIRA88] Miranda S., Le Thanh N., Borelli E., Bonfils G., "Présentation du modèle B-relationnel", *MBD*, mars 1988

[MOLL87] Moll G. H., "Un langage pivot pour le couplage de PROLOG avec des bases de données", *Thèse de doctorat IUT Lyon I*, Lyon

[MONT73] Montague R., "The proper treatment of quantification in ordinary English", *Approaches to Natural Languages*, Ed. Dordrecht, RFA

[NAIS83] Naish L., MU-Prolog reference manual, Melbourne University, 1983

[NARA87] Narat V., Lochet P.Y.,"Les différentes techniques de représentation des connaissances utilisées en I.A.", *MBD n°6* Paris, juin 1987

[NAVA87] Navathe S.B., Ahmed R.,"TSQL, a language intreface for history databases", *TAIS*, Sophia Antipolis (F), 1987

[NGUY85] Nguyen G.T., Olivares J., "Sycslog, système logique d'intégrité sémantique", *Rapport TIGRE n°26*, IMAG, Grenoble, 1985

[PENA84] Penado M.H., "Prototyping a Project Master Database for Software Engineering Environments", *IEEE Transaction on Software Engineering*, 1985

[PROL87] PROLOG II, version 2.4, manuel de référence, *PrologIA*, Marseille

[QUIL68] Quillian R., "Semantic memory", *MIT Press*, Cambridge Massachusetts, 1968

[RAPH68] Raphael B., "A computer program for semantic information retrieval", *MIT Press*, Cambridge

Massachusetts, 1968

[RECH87] Rechenmann F., "Les représentations de connaissances centrées objets", *Documentation IMAG*, 1987

[ROLL82] Rolland C., Richard C., "REMORA", *IFIP Conference on Comparative Review of Information Systems Design methodologies*, Ed. North Holland, 1982

[ROSS79] Ross D.T., "Structured analysis" (deux articles), *IEEE Transactions on Software Engineering*, USA, 1977

[SERN80] Sernadas A., "Temporal aspects of logical procedure definition", *Information systems* 5, 1980

[SHAP77] Shapiro S.C., "Representing and locating deduction rules in a semantic network", *ACM/SIGART Newsletter 63*, 1977

[SIMO69] Simon H.A., The science of the artificial, extraits, *MIT Press*, Cambridge, Massachusetts, 1969

[SMIT77] Smith J.M., Smith D.C.P., "Database abstractions : aggregation and generalization", *ACM Transactions on Database Systems*, juin 1977

[SNOD84] Snodgrass R., "The temporal Query Language TQUEL", *ACM symposium on Database systems*, Waterloo (CN), avril 1984

[STER84] Sterling L., "Expert System = Knowledge + Meta-interpreter", *Weizmann Institute of Science Tech. Rep. CS84-17*, Rehovot (Israel), 1984

[TAPI78] Tapiero C.S., "Time, Dynamics, and the process of management modelling", *TIMS studies in management*

science 9, 1978

[TARD79] Tardieu H., Nanci D., Pascot D., "Conception d'un système d'information", Ed. les éditions d' organisation, Paris, 1979

[TARD83] Tardieu H., Rochfeld A., Coletti R., "La méthode Merise", Ed. les éditions d'organisation, Paris, 1983

[THOM87] Thomas I., "The PCTE Initiative and the PACT Project", in [ESPR87], 1987

[ULLM80] Ullman J.D., "Principles of Database Systems", *Ed. Computer Science Press*, 1980

[VASS79] Vassiliou Y., "Null values in Database Management", *ACM SIGMOD*, 1979

[VENK85] Venken R., "The interaction between Prolog and Relational Databases", *ESPRIT project 107*, 1985

[WILL86] Williams C., "ART : Conceptual Overview", *Doc. Inference Corporation*, Los Angeles, 1985

[ZANI86] Zaniolo C. & al, "Object Oriented Database Systems and Knowledge Systems", in *Expert Database Systems*, Ed. CPC, 1986

Plan bibliographique

Chapitre 0

0.1	[LEMO73]	[TARD79]		
0.2	[ANDR84]	[BOSC85]	[PENA84]	[JENK87]
	[HURS86]	[THOM87]		

Chapitre 1

1.1	[FROS87]	[CHEN76]	[HUSR86]	[CONC86]
1.2	[MACC79]	[BOEH83]		

Chapitre 2

2.1	[RAPH68]	[QUIL68]	[HEND75]	[SHAP77]
	[DELI79]	[FAHL75]	[BRAC77]	[FROS86]
	[KOWA75]	[CARB70]		
2.2	[RECH87]	[MINS75]	[GOLD83]	[ZANI86]
	[COPE84]	[FERN87]	[MELL87]	[BOSC88a]
2.3	[COLM87]	[DUBO87]	[LEWI32]	[MACC69]
	[MACD80]	[DOYL82]	[MONT73]	[BOUB85]
	[SERN80]	[LUND82]	[ILIN86]	
2.4	[SIMO69]	[FEIG71]	[LAUR82]	[WILL86]
	[FAJO88]	[JENK87]		
2.5	[ULLM80]	[LATO86]	[MIRA86a]	[ALLE86]
	[CHEN76]	[ALLE85]	[DELA86]	[SMIT78]
	[LEDI86]	[CHRI83]	[CHRI85]	[BOGO83]
	[VASS79]	[BUBE77]	[TAPI78]	[CARM87]
	[SNOD84]	[KLOP83]	[NAVA87]	[DOYL79]
2.6	[CAZI84a]	[ALLE85]	[CHEN76]	[ANDR86]
	[LICH87]	[GALL84]	[LPDB86]	[GARD87]
	[FROS85]	[CAZI84b]	[CAZI85]	[ALLE85]
	[LANG85]			
2.7	[ALLE86b]	[CONC86]	[CASE87]	[NARA87]
	[LPDB86]			

Chapitre 3

3.1	[MIRA86b]	[GARD86]	[LOPE83]	[NAIS83]

	[VENK85]	[DONZ83]	[GARD87]	[BOCC87]
	[CORN87]	[CUPP86]	[GIAN85]	[COLM83]
	[COLM87]			
3.2	[FONT84]	[THOM87]	[CAZI85]	
3.3	[TARD83]	[ROLL82]	[FOUC82]	[GL86]
	[ROSS77]	[JACK83]	[COX83]	[BOOC82]
	[CHEN76]			

Chapitre 4

4.2	[FRAN87]	[FAIR87]	[NGUY85]	

Chapitre 6

6.2	[GOLD83]	[COX83]		
6.3	[PROL87]	[ESQL86]	[MOLL87]	[BOSC88b]
	[STER84]	[JENK87]		

Chapitre 7

7.3	[BOCC87]	[LETH86]	[MIRA88]	[BANC87]

Annexe1

	[CONC86]	[CAZI84a]	[ALLE86a]	[ALLE86b]

Bibliographie

Références du texte :

[ALLE85] [ALLE86a] [ALLE86b] [BOSC85] [BOSC88a] [BOSC88b] [JENK87]

Autres publications :

par L. Boi, M. Bosco, M. Gibelli :

"LARKS : a Logic And Relational Knowledge System" *ESPRIT International Workshop*, Venice, Italy, 1986

par M. Bosco, M. Gibelli :

"Impish, A RDBMS Extended to Handle Logical Rules and Documents" in *ESPRIT '87 Achievements and Impact*, Ed. CEC & North Holland, 1987

"Impish, une extension de SGBDR à la logique et à la gestion de documents" in *JISI'88, Les bases de données: état de l'art et perspectives*, Tunis, Tunisie, Avril 1988

"An overview of Impish, an intelligent I.S. Host" in CASE'88, Cambridge, MA, Etats Unis, Juillet 1988

Documents ESPRIT P938 :

"Specification and Design of the KDB", CETE-WP2-S-004, Septembre 1986

"IS Design", CETE-WP2-D-004, Mars 1987

"Impish User Manual", CETE-WP2-M-013, Novembre 1987

"An outline of an Impish Demonstration Scenario"

Annexes

Annexe1

Une interface interactive orientée objet pour Impish

1 Introduction

Une interface interactive à Impish, basée sur les concepts d'objets, a été partiellement spécifiée, sans être implémentée. Nous en présentons ici les principes majeurs qui serviront de base à une réalisation.

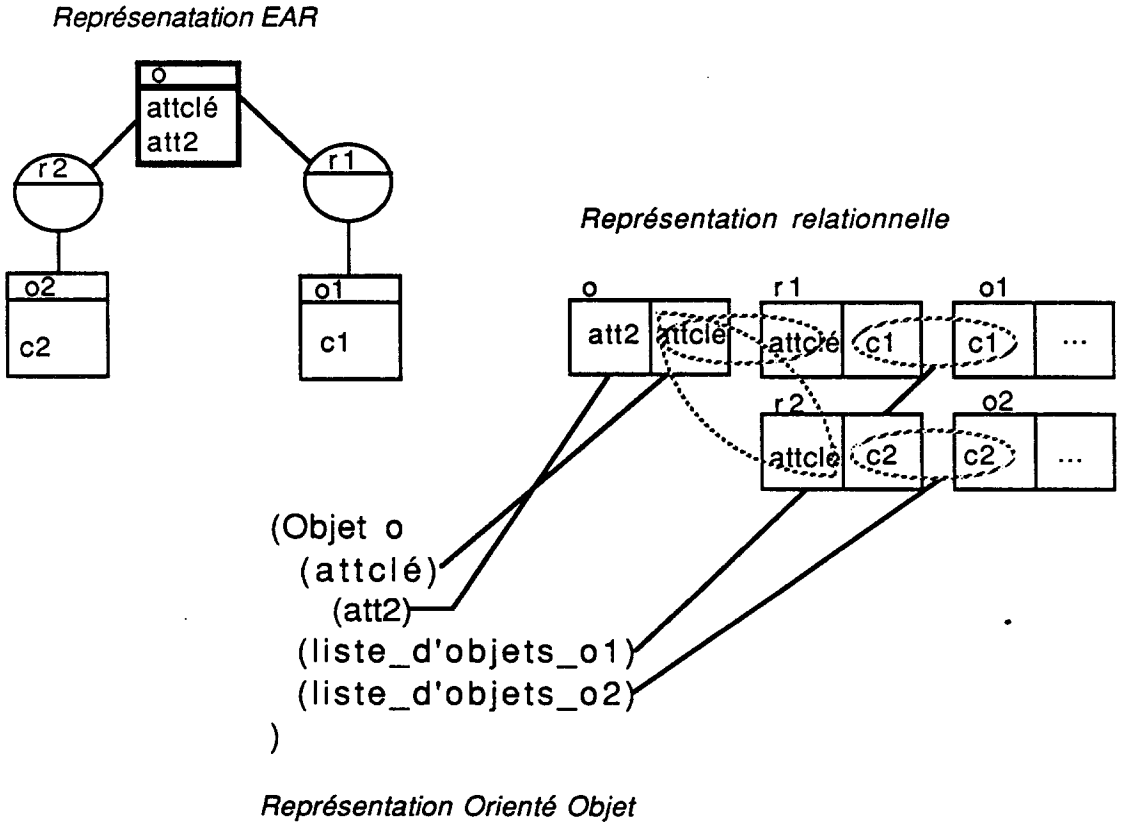
2 Définition d'un objet pour l'interface

Un objet est assimilé à une entité (au sens EAR), dont les slots auraient consisté en ses attributs (au sens EAR) et en l'ensemble des images de ses relations avec d'autres entités .

Cette solution a été partiellement automatisée, mais abandonnée, dans un premier temps, comme structure d'objets interfaces vers des outils. Mais elle a été abandonnée parce qu'elle nuit à la flexibilité dans la conception du système d'information : en effet, chaque application, chaque outil, pour l'utiliser, doit définir ses méthodes d'accès aux objets. Ces méthodes doivent ensuite être toutes regroupées.

Ce choix est au contraire très satisfaisant pour un outil indépendant des autres (et qui fait en réalité partie d'Impish, puisque il en est le "browser", l'outil de navigation et d'exploration). Il présente l'avantage de

conduire à une interface très naturelle et ergonomique.



Conversion EAR / Objet

3 Création d'un objet

Nous supposons que notre point de départ est un modèle représenté en EAR, et notre objectif la manipulation des objets que l'on peut associer aux entités de cette représentation (cf. chapitre 2.6).

Nous devons pour cela générer, pour chaque entité du diagramme EAR (en fait, les règles d'équivalences attributs et

rappelées dans un schéma du chapitre 2.6), une classe d'objet Objective-C, dont les "slots" sont :

```
= Entité1 : Object

//instance variables
{
id  attribut1,
    attribut2,
    …
    liste_des_objets_en_relation_par_rel1,
    liste_des_objets_en_relation_par_rel2,
    …;
}
```

Ces "slots" sont tous de type "id" (i.e. représentent des objets), même quand il s'agit de types simples d'Impish (comme des chaines de caractères, des entiers, des dates, mais aussi des "links") : ils désignent des occurrences des classes chaines, entiers, dates, … . Dans le cas des images par des relations (liste_des_…), ces Id représentent des collections d'objets.

Chacune de ces classes contient une "factory" méthode:

```
+ createObject
```

qui permet de définir une instance pour laquelle on définit une méthode d'instanciation :

```
- instanciateWith:aCollection
```

dont l'argument est l'une des réponses à une "sélection

d'objets" dans Impish. Le corps de cette méthode peut être défini automatiquement :

```
- instanciateWith:aCollection
{
anIdArray = [aCollection asIdArray];
[self attribut1:[anIdArray at:1]];
[self attribut2:[anIdArray at:2]];
…
[self [liste_des_objets…rel1:[anIdArray at:…]]]
[self [liste_des_objets…rel2:[anIdArray at:(…+1)]]]
…
}
```

ainsi que, pour chaque attribut, les méthodes d'instanciation et d'interrogation :

- attribut1:anId (qui donne pour valeur à la variable d'instance "attribut1" l'objet dont l'identifiant est passé en paramètre),

- attribut1 (qui renvoit la valeur courante de la variable d'instance "attribut1").

Ces méthodes s'appuient sur les méthodes duales implémentées dans les classes de bases qui transforment les types de bases Impish (chaines, réels, entiers, …, dates, et "links") en classes d'objets.

Ces différents éléments sont regroupés dans la classe Entité1, bien sur, et stockés dans un fichier "Entité1.m" qui est compilé lors de la configuration des outils sur le système d'information.

Les outils peuvent, s'ils le désirent, utiliser le mécanisme d'héritage d'Objective-C pour se créer des objets

plus spécifiques.

Nous allons maintenant voir qu'une extension du langage d'interrogation d'Impish, qui puisse permettre de manipuler (rechercher, insérer, supprimer, modifier) des occurrences d'objets à part entière est aussi nécéssaire.

4 Sélection d'objets

En s'appuyant sur les structures que l'on vient de présenter, il va être possible de renvoyer à l'outil des collections d'entités, et non plus des collections de t-uples comme présenté plus haut.

Une requête de sélection d'objets a pour forme un SELECT d'Impish où la "liste de sélection" (la "selection list" de la grammaire ESQL) est la suivante :

<selection_list> ::= OBJECT <type_d'entité>

où <type_d'entité> est le nom d'une classe comme présentée plus haut.

On introduit aussi une nouvelle expression de filtrage (en plus des "where" et "with" clauses), la "matching" expression, de la forme :

MATCHING <id>

où <id> est l'identifiant d'un objet, au sens d'Objective-C, de même type que <type_d'entité>. Les variables d'instances de l'objet <id> qui sont affectées sont prises en compte pour la définition d'une "where" clause :

L'objet "myTaskFilter", dont la variable "attribut2"

vaut "3" et la liste "TaskWorksOnProduct-ProductName", qui contient les identifiants d'objets dont les clés sont "Impw" et "Concerto"(et dont aucun autre "slot" n'est instancié), et considéré comme un filtre dans une "matching" expression, se traduira par la "where" clause suivante :

```
WHERE Task.TaskId = TaskWorksOnProduct.TaskName
    AND TaskWorksOnProduct.ProductName = "Impw"
    AND TaskWorksOnProduct.ProductName = "Concerto"
    AND Task.attribut2 = "3"
```

Cette construction du filtre est récursive et tient compte des objets pointés par les objets pointés par … l'objet passé en paramètre.

On notera le choix qui consiste à n'autoriser que la conjonction dans l'expression d'un filtre par une occurence dans une relation (au sens EAR), et que l'égalité avec des constantes pour la contrainte sur les valeurs d'attributs. Nous ne permettons pas l'expression simple du "ou" dans le requête.

Nous avons eu l'occasion d'implémenter un tel mécanisme dans le cadre du projet Concerto, pour connecter un système Orienté Objet (Ulysse, [CONC86]) et un système EAR (F1, [CAZI84a]), pour la commande "chercher" de Patre [ALLE86a] [ALLE86b]. Cette réalisation sert de base aux développements projetés dans Impish.

En résumé,

```
SELECT OBJECT Task FROM Task, TaskWorksOnProduct
    WITH ancestor_tasks(Task.TaskId,"Task122")
    MATCHING myTaskFilter
```

recueille dans une collection les objets "tache" qui conduisent à la fois à la réalisation des produits Impw et Concerto, dont la valeur de l'attribut2 est 3 (matching) et qui sont en amont (spécification du prédicat "ancestor_tasks") de Task122 (with, cf. la grammaire d'Impish).

L'obtention de ces objets passe par le mécanisme suivant :

- Analyse de l'objet filtre (comme précisée ci-dessus),

- Recomposition de la sélection liste globale composée:

.de la liste des colonnes de l'entité considérée (au sens EAR),

.de la liste des colonnes qui représentent les clés des entités d'autres types avec lesquelles notre objet est en relation (au sens EAR),

- Recomposition de la "where-expression", par l'ajout des conditions d'égalité qui représentent les "joins" existants entre les tables de l'entité et celles de ses mises en relations (cf. les ovales en pointillé dans la figure ci-dessus), et du résultat de l'analyse du filtre,

- Exécution de la commande Impish ainsi reconstituée, qui s'appuie sur le mécanisme rustique déja présenté,

- Création des différents objets du type correspondant à celui indiqué dans l'expression de la sélection, mais aussi de ceux qui sont contenus par les collections correspondants aux "slots" de type

"liste_d'objets_…_rel".

5 Manipulation d'objets

L'objectif ici est de permettre d'insérer, supprimer ou modifier des faits dans la base de donnée d'Impish en passant en paramètre aux commandes INSERT, DELETE et UPDATE non pas des noms de tables et des restrictions définies par des "where" ou "with" clauses, mais plutôt des objets structurés qui contiendrons la définition de la manipulation à réaliser.

L'exécution de la commande :

```
INSERT OBJECT myTask
```

se traduit par une transaction contenant :

.l'insertion du tuple identifié par les attributs de "myTask",

.l'insertion des différents tuples représentant les relations (au sens EAR) exprimées par les occurrences d'objets dans les slots de "myTask" de type "liste_d'objets_…_rel".

L'exécution de la commande :

```
DELETE OBJECT myTaskFilter
```

supprime toutes les occurences dans la relation (au sens ESQL) qui correspond au type de "myTaskFilter", qui sont filtrées par "myTaskFilter" (comme dans le SELECT OBJECT).

L'exécution de la commande :

UPDATE OBJECT myNewTask myTaskFilter

compose la "set" expression du "UPDATE" Impish à partir des attributs de "myNewTask" (sans tenir compte des relations de cette entité), et sa "where" expression à partir de "myTaskFilter".

6 L'expression des valeurs nulles et des variables "libres"

La sémantique donnée aux valeurs nulles reste, comme dans l'interface avec Prolog, celle qui leur est donnée par le SGBD.

Leur expression, toutefois, est propre à l'environnement orienté objet, et il sera fait usage d'une occurrence de la classe "NullObject" pour décrire le fait qu'un attribut a une valeur nulle dans la base.

Un attribut peut aussi ne pas être instancié (i.e. pointer sur "nil") : c'est ainsi que l'on définira une variable "libre" dans un filtre.

7 L'aspect externe de l'interface

Nos travaux dans Concerto, déja cités, ont permis de définir et d'implémenter graphiquement un certain nombre des commandes que l'on vient de spécifier : nous permettions deconstruire des filtres, en associant des objets (en les mettant en relation), et en permettant l'édition de leurs attributs (cf. 2.6). Cet aspect sera toutefois à retravailler en fonction des moyens utilisés autour d'Impish.

Annexe2

Un exemple de modèle de connaissances Impish

Une représentation interne du modèle

```
*********************************

 Command : status


*********************************


urrent-base : "these"

rogram-number(4) ->;

clause1,deterministic(a),erase(a).0.nil> ->;
clause1,TypeOfMeanType("language"),nil> ->;
clause1,TypeOfMeanType("method"),nil> ->;
clause1,TypeOfMeanType("technique"),nil> ->;
clause1,TypeOfMeanType("tool"),nil> ->;

omain("LevelType","CHAR ( 20 )",clause2) ->;
omain("TypeOfMeanType","CHAR ( 20 )",clause1) ->;

clause2,deterministic(a),erase(a).0.nil> ->;
clause2,LevelType("beginner"),nil> ->;
clause2,LevelType("senior"),nil> ->;
clause2,LevelType("expert"),nil> ->;

ttribute("hdatabis","hdId","CHAR ( 10 )",1) ->;
ttribute("hdataone","hdId","CHAR ( 10 )",1) ->;
ttribute("hdataone","hdType","CHAR ( 10 )",2) ->;
ttribute("ResHasCal","ResourceId","CHAR ( 20 )",1) ->;
ttribute("ResHasCal","CalType","CHAR ( 20 )",2) ->;
ttribute("ResHasCal","FromDate","DATE",3) ->;
ttribute("ResHasCal","ToDate","DATE",4) ->;
ttribute("ResKnowMeans","ResourceId","CHAR ( 20 )",1) ->;
ttribute("ResKnowMeans","MeanId","CHAR ( 20 )",2) ->;
ttribute("ResKnowMeans","Experience","LevelType",3) ->;
ttribute("Ressource","ResourceId","CHAR ( 20 )",1) ->;
ttribute("Means","MeanId","CHAR ( 20 )",1) ->;
ttribute("Means","TypeOfMean","TypeOfMeanType",2) ->;

ey("hdatabis","hdId",1) ->;
ey("hdataone","hdId",1) ->;
ey("hdataone","hdType",2) ->;
ey("ResHasCal","ResourceId",1) ->;
y("ResHasCal","CalType",2) ->;
y("ResHasCal","FromDate",3) ->;
```

```
y("ResKnowMeans","ResourceId",1) ->;
y("ResKnowMeans","MeanId",2) ->;
y("Ressource","ResourceId",1) ->;
y("Means","MeanId",1) ->;

nk("Ressource","CV") ->;
nk("Ressource","Photo") ->;
nk("Means","Manual") ->;

ata("hdatabis","hdId") ->;
ata("hdataone","hdId") ->;
ata("hdataone","hdType") ->;
ata("ResHasCal","ResourceId") ->;

ol-doc("plan") ->;

lause3,deterministic(a),erase(a).0.nil> ->;
lause3,earlier(a,b),extract-val(a,c).extract-val(b,d).inferior(c,d).nil> ->;
lause3,extract-val(a,b),extract-mja(a,c,d,e).conc-string(e,c,f).conc-string(f,d,g).string-integer(g,b).nil> ->;
lause3,inferior(a,b),val(inf(a,b),1).nil> ->;
lause3,extract-mja(a,b,c,d),substring(a,1,2,b).substring(a,4,2,c).substring(a,9,2,d).nil> ->;

onstraint("hdatabis","DELETE","AFTER",clause4,"deptest") ->;
onstraint("hdatabis","UPDATE","AFTER",clause4,"deptest") ->;
onstraint("hdataone","UPDATE","AFTER",clause4,"deptest") ->;
onstraint("hdataone","INSERT","AFTER",clause4,"deptest") ->;
onstraint("ResHasCal","INSERT","BEFORE",clause3,"earlier") ->;

arameter(<"hdataone","hdId">,1,clause4,"deptest") ->;
arameter(<"hdataone","hdType">,2,clause4,"deptest") ->;
arameter(<"hdatabis","hdId">,3,clause4,"deptest") ->;
arameter(<"ResHasCal","FromDate">,1,clause3,"earlier") ->;
arameter(<"ResHasCal","ToDate">,2,clause3,"earlier") ->;

eference(<"ResKnowMeans","ResourceId">,<"Ressource","ResourceId">) ->;

clause4,deterministic(a),erase(a).0.nil> ->;
clause4,deptest(a,b,a),erase(hdataone(a,"t1")).erase(hdatabis(a)).nil> ->;

ependence("hdataone",clause4,"deptest") ->;

End world: model

**********
```

```
**********************************

urrent-base : "these"
JTURE           REF_DOC         TEMP            USER
LD              RUB             TOOL_DOC

hese/FUTURE:

hese/OLD:

hese/REF_DOC:
eans            Ressource

hese/REF_DOC/Means:
anual

hese/REF_DOC/Means/Manual:

hese/REF_DOC/Ressource:
V       Photo

hese/REF_DOC/Ressource/CV:

hese/REF_DOC/Ressource/Photo:

hese/RUB:

hese/TEMP:

hese/TOOL_DOC:
lan

hese/TOOL_DOC/plan:

hese/USER:
```

Des contraintes indentifiées, à implémenter

On trouvera ci-après une liste de quelques contraintes obtenues par enquête auprès du CETE, organisme sous tutelle du ministère de l'Equipement, dont l'implémentation doit permettre l'utilisation de l'atelier IMPW dans sa version "clientalisée" :

1. Parmi les ressources matérielles de l'entreprise, si le client est le ministère, seules celles agrées par le ministère sont affectables à une tache.

2. Le commencement d'une phase n'est possible que sous réserve de l'accord du client.

3. Un projet ne peut être commencé que si son finacement est au moins égal (mais pas forcément plus grand) que son coût estimé (ce qui traduit la non obligation de profit).

4. Une ressource ne peut être affectée à une tache qui appartient au chemin critique du plan du projet si son expérience est "faible".

On notera qu'il est toutefois difficile de mettre en évidence ces contraintes, et que les techniques d'acquisition des connaissances mériteraient véritablement d'être approfondies.

Annexe3

Détails des mécanismes du couplage PrologII/ESQL

```
----------------------------------------"
xecute an sql command composed of subtrees "
----------------------------------------"

c-ommand, a-nswer, e-rror) ->
     execute-sql-command(c-ommand, a-nswer, e-rror);

cute-sql-command(c-ommand, a-nswer, e-rror) ->
     if-then-else(select-command(c-ommand),
                          execute-sql-select(c-ommand, a-nswer,l-length,t-length, l
                          execute-sql-other(c-ommand, e-rror));


cute-sql-select(c-ommand, a-nswer,l-length,t-length, l-type, e-rror) ->
     send-sql-command(c-ommand)
     Send-sql-select
     Receive-sql-error(e-rror)
     error-if(and(dif(e-rror,+0.0),dif(e-rror,+100.0)),e-rror)
     receive-sql-answer(a-nswer,l-length,t-length,l-type);

cute-sql-other(c-ommand, e-rror) ->
     send-sql-command(c-ommand)
     Send-sql-other
     Receive-sql-error(e-rror)
     error-if(dif(e-rror,+0.0),e-rror);

eive-sql-answer(a-nswer,l-length,t-length,l-type) ->
     Receive-list-length(l-length)
     Receive-tuple-length(t-length)
     receive-sql-type(t-length,l-type)
     receive-sql-answer'(l-length, t-length, a-nswer);

eive-sql-type(0,nil) ->/;
eive-sql-type(t-length,l-type1) ->
     val(sub(t-length,1),t-length1)
     Receive-sql-type(t-length,t-ype)
     receive-sql-type(t-length1,l-type)
     conc(l-type,t-ype.nil,l-type1);

eive-sql-answer'(0, t-length, nil) ->/;
eive-sql-answer'(l-length, t-length, t-uple.l-ist) ->
     receive-sql-tuple(l-length, t-length, t-uple)
     val(sub(l-length, 1) , l-engthl)
     receive-sql-answer'(l-engthl, t-length, l-ist);

eive-sql-tuple(l-length, 0, nil) ->/;
eive-sql-tuple(l-length, t-length, v-alue.l-ist) ->
     Receive-value(l-length, t-length, v-alue)
     val(sub(t-length, 1), t-length1)
     receive-sql-tuple(l-length, t-length1, l-ist);

lect-command(c-ommand) ->
     substring(c-ommand,1,6,"select")/;
lect-command(c-ommand) ->
     substring(c-ommand,1,6,"SELECT")/;

nd-sql-command(s) ->
     scan(s,1,80,l)
     arg(0,l,n)
     Send-sql-command-length(n)
     send-sql-command'(l);

nd-sql-command'(nil) ->;
nd-sql-command'(x.l) ->
     Send-sql-command-part(x)
     send-sql-command'(l);

scan(s,i,j,x.l) ->
     substring(s,i,j,x)
     val(add(i,j),il)
     scan(s,il,j,l);
scan(s,i,j,x.nil) ->
     arg(0,s,l)
     val(add(l,1),ll)
     val(sub(ll,i),jl)
     substring(s,i,jl,x)/;


"------------------------"
"     translator objc    "
"------------------------"

send-ObjC-sql-other-answer(a-nswer) ->
     string-type(s-tring,a-nswer)
     Send-string-on-fifo(s-tring);

send-ObjC-sql-select-answer(a-nswer,l-type) ->
     length(a-nswer,l-length)
     length(l-type,t-length)
     send-list-length(l-length)
     send-tuple-length(t-length)
     send-list-type(t-length,l-type)
     send-ObjC-sql-answer'(l-length, t-length, a-nswer);

send-ObjC-sql-answer'(0, t-length, nil) ->/;
send-ObjC-sql-answer'(l-length, t-length, t-uple.l-ist) ->
     send-ObjC-sql-tuple(l-length, t-length, t-uple)
     val(sub(l-length, 1) , l-length1)
     send-ObjC-sql-answer'(l-length1, t-length, l-ist);

send-ObjC-sql-tuple(l-length, 0, nil) ->/;
send-ObjC-sql-tuple(l-length, t-length, v-alue.l-ist) ->
     send-value(l-length, t-length, v-alue)
     val(sub(t-length, 1), t-length1)
     send-ObjC-sql-tuple(l-length, t-length1, l-ist);

send-list-type(0, nil) ->/;
send-list-type(t-length, t-ype.l-ist) ->
     send-ObjC-sql-type(t-ype)
     val(sub(t-length, 1) , t-length1)
     send-list-type(t-length1, l-ist);

|receive-ObjC-command(c-ommand) ->|
|     recv-string(4,c-ommand);     |

receive-ObjC-command ->
     Init-query
     recv-integer(4,n)
     receive-ObjC-part(n);

receive-ObjC-part(0)->/;
receive-ObjC-part(n)->
     recv-string(4,p-art)
     Send-sql-query-part(p-art)
     outml(p-art)
     minus(n,1,n1)
     receive-ObjC-part(n1);
```

```
nd-ObjC-sql-type(t-ype) ->
      Send-integer-on-fifo(t-ype);

nd-list-length(l-length) ->
      Send-integer-on-fifo(l-length);

nd-tuple-length(t-length) ->
      Send-integer-on-fifo(t-length);

nd-value(l-length, t-length, v-alue) ->
      integer(v-alue)
      Send-integer-on-fifo(v-alue);

nd-value(l-length, t-length, v-alue) ->
      string(v-alue)
      Send-string-on-fifo(v-alue);

nd-value(l-length, t-length, v-alue) ->
      real(v-alue)
      Send-real-on-fifo(v-alue);



--------------------"
  erase enfin !     "
--------------------"

ase(x.p) -> /db-erase(x) erase(p);
ase(x) -> db-erase(x);

-erase(p) ->
      if-then(val(meta-trace,yes),outl(p))
      p;
-erase(p) ->
      split(p, l)
      eq(l,x.t-uple)
      arg(0,l,l-engths)
      good-length(x,l-engths)|
      mrpt(x)
      cond-list(x, c-ond, t-uple, l)
      compose-statement(x, c-ond, s-tatement)
      execute-sql-select(s-tatement, a-nswer,l-lengths,t-lengths, l-types, e-rror)
      contains(a-nswer, t-uple);

od-length(x,n) ->
      string-ident(s,x)
      attribute(s,a,t,n)
      plus(n,l,nl)
      not(attribute(s,a,t,nl))/;

rpt(x) ->
      string-ident(s,x)
      attribute(s,a,t,n) /;

nd-list(t-able, nil, nil, v) ->/;
nd-list(t-able, c-ond, c.l, c-ounter) ->
      free(c)
      val(add(c-ounter, 1), c-ounter1)
      cond-list(t-able, c-ond, l, c-ounter1)/;
d-list(t-able, <c-olumnname, c>.c-ond, c.l, c-ounter) ->
      bound(c)
      string-ident(s,t-able)
      attribute(s, c-olumnname, t-ype, c-ounter)
      val(add(c-ounter, 1), c-ounter1)
      cond-list(t-able, c-ond, l, c-ounter1);

compose-statement(t-able, c-ond, s-tatement) ->
      string-ident(x1, t-able)
      conc-string("select * from ", x1, x2)
      if-then-else(dif(c-ond,nil),
            conc-string(x2, " where ", x3).
            compose-cond-string(c-ond, s).
            conc-string(x3, s, s-tatement),
            eq(s-tatement,x2));

compose-cond-string(<p,v>.nil, s) ->
      compose-one-cond(p,v,s) /;
compose-cond-string(<p,v>.l, s) ->
      compose-cond-string(l, s1)
      compose-one-cond(p,v,s2)
      conc-string(s2, " and ", s3)
      conc-string(s3, s1, s);

compose-one-cond(c-olumnname, v-alue, s) ->
      string-make(v-alue, s-va)
      conc-string(c-olumnname," = ",s1)
      conc-string(s1,s-va,s);

;
```

```
clude sqlca;
clude sqlda;
clude sqltypes;
clude <decimal.h>

fine DSTRSZ 40

tic int tLength, lLength;
tic char *result[1000][20];
tic int types[20];
tic int cLength;
tic char *command;

eive_list_length(err, err_nb)
 *err, *err_nb;

     put_integer(1, lLength, err);



eive_tuple_length(err, err_nb)
 *err, *err_nb;

     put_integer(1, tLength, err);



eive_sql_error(err, err_nb)
 *err, *err_nb;

     float fl;
     fl = sqlca.sqlcode;
     put_real(1, fl, err);


ceive_sql_error_message(err, err_nb)
 *err, *err_nb;

     char mess[72];
     strcpy(mess , sqlca.sqlerrm);
     printf("message  %s\n",sqlca.sqlerrm);
     put_string(1, mess, err);


ceive_sql_type(err, err_nb)
t *err, *err_nb;

t i;
     get_integer(1, &i, err);
     put_integer(2, types[i], err);


ceive_value(err, err_nb)
t *err, *err_nb;

     int lineNumber, columnNumber;

     get_integer(1, &lineNumber, err);
     get_integer(2, &columnNumber,err);

     lineNumber   = lLength - lineNumber + 1;
     columnNumber = tLength - columnNumber + 1;
     put_value(3, lineNumber, columnNumber, err);

}


put_value(parameterNumber, lineNumber, columnNumber, err)
int parameterNumber, lineNumber, columnNumber, *err;
{
     int *i,i1;
     long *li,li1;
     double  *d,d1;
     float fl,*f,f1;
     dec_t *np,np1;


/*   printf("line, column %d    %d",lineNumber, columnNumber);
     printf("result(l,c)= %s",result[lineNumber][columnNumber]);
     printf("type= %d\n",types[columnNumber]);*/

     i = &i1;
     li = &li1;
     d = &d1;
     np = &np1;
     f = &f1;

     switch (types[columnNumber])
     {
     case SQLDATE:
     case SQLCHAR:
          put_string(parameterNumber, result[lineNumber][columnNumber], err);
          break;
     case SQLINT:
          rstol(result[lineNumber][columnNumber], li);
          fl = *li;
          put_real(parameterNumber, fl, err);
          break;
     case SQLSMINT:
          rstoi(result[lineNumber][columnNumber], i);
          fl = *i;
          put_real(parameterNumber, fl, err);
          break;
     case SQLFLOAT:
          d = result[lineNumber][columnNumber]; /* genere un warning */
          fl = *d;
          put_real(parameterNumber, fl, err);
          break;
     case SQLSMFLOAT:
          f = result[lineNumber][columnNumber];/* genere un warning */
          fl = *f;
          put_real(parameterNumber, fl, err);
          break;
     case SQLMONEY:
     case SQLDECIMAL:
          rstod(result[lineNumber][columnNumber],d);
          fl = *d;
          put_real(parameterNumber, fl, err);
          break;
     case SQLSERIAL:
          rstol(result[lineNumber][columnNumber], li);
          put_integer(parameterNumber, *li, err);
          break;
     }
```

```
d_sql_select(err, err_nb)
 *err, *err_nb;

    struct sqlda *udesc;
    struct sqlvar_struct *col;
    int pos;
    char *buffer;
    char decstr[DSTRSZ];
    int i, len, j;
    register char *cp;
    char *malloc();
    $ char chaine[400];

    strcpy(chaine , command);

    init_sql_var();

    $ prepare usqlobj from $chaine;

    if (sqlca.sqlcode != 0) return;

    $ describe usqlobj into udesc;

    if (sqlca.sqlcode != 0) return;

    tLength = udesc->sqld;
    pos = 0;
    for (col = udesc->sqlvar, i = 1; i <= udesc->sqld; col++, i++)
        {
            types[i] = col->sqltype;
            switch(col->sqltype)
            {
            case SQLSMFLOAT:
                pos = rtypalign(pos,CFLOATTYPE);
                col->sqltype = CFLOATTYPE;
                pos += rtypmsize(CFLOATTYPE);
                col->sqllen = rtypmsize(CFLOATTYPE);
                break;

            case SQLFLOAT:
                pos = rtypalign(pos,CDOUBLETYPE);
                col->sqltype = CDOUBLETYPE;
                pos += rtypmsize(CDOUBLETYPE);
                col->sqllen = rtypmsize(CDOUBLETYPE);
                break;

            case SQLMONEY:
            case SQLDECIMAL:
                pos = rtypalign(pos,CDECIMALTYPE);
                col->sqltype = CDECIMALTYPE;
                pos += rtypmsize(CDECIMALTYPE);
                col->sqllen = rtypmsize(CDECIMALTYPE);
                break;

            default:
                pos = rtypalign(pos,CFIXCHARTYPE);
                col->sqllen = rtypwidth(col->sqltype,col->sqllen)+1;
                pos += col->sqllen + 2;
                col->sqltype = CFIXCHARTYPE;
            }
        }
            buffer = malloc(pos);
            pos = 0;
            for (col = udesc->sqlvar, i = 0; i < udesc->sqld; col++, i++)
                {
                pos = rtypalign(pos,col->sqltype);
                col->sqldata = buffer + pos;
                pos += col->sqllen;
                if (col->sqltype != CDECIMALTYPE)
                pos++;
                }


        $ declare usqlcurs cursor for usqlobj;

        if (sqlca.sqlcode != 0) return;

        $ open usqlcurs;

        if (sqlca.sqlcode != 0) return;

        $ fetch usqlcurs using descriptor udesc;
        lLength = 1;
        while (sqlca.sqlcode == 0)
                {
                pos = 0;
                for (col = udesc->sqlvar, i = 1; i <= tLength; col++, i++)
                    {
                                {
                                if (col->sqltype == CDECIMALTYPE)
                                        {
                                        dectoasc(col->sqldata,decstr,DSTRSZ,-1);
                                        ldchar(decstr,DSTRSZ,decstr);
                                        result[lLength][i] = malloc(DSTRSZ+1);
                                        strcpy(result[lLength][i], decstr);
                                        }
                                else
                                        {
                                        len = col->sqllen;
                                        cp = col->sqldata + len - 1;
                                        while (len > 1 && *cp == ' ') len--, cp--;
                                        *(++cp) = 0;
                                        result[lLength][i] = malloc(len+1);
                                        cp = result[lLength][i];
                                        strcpy(result[lLength][i], col->sqldata);
                                        }
                                }
                        }
                        lLength++;
                $ fetch usqlcurs using descriptor udesc;
                }
        lLength--;
}


send_sql_other(err, err_nb)
int *err, *err_nb;
{
        $ char chaine[400];

        strcpy(chaine , command);

        init_sql_var();

        $ prepare query from $chaine;
```

```
    $ execute query;


d_sql_command_length(err,err_nb)
 *err, *err_nb;

    char *malloc();

    get_integer(1, &cLength, err);
    command = malloc(400);
    strcpy(command, "");



d_sql_command_part(err,err_nb)
 *err, *err_nb;

    char chaine[132]; /* superieur a 81 */
    get_string(1, chaine, err);
    strcat(command, chaine);



t_sql_var()

    tLength = lLength = 0;
```