N° d'ordre : 11

Année 1987

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

Claire SAUREL

CONTRIBUTION AUX SYSTEMES EXPERTS : DEVELOPPEMENT D'UN CAS CONCRET ET ETUDE DU PROBLEME DE LA GENERATION D'EXPLICATIONS NEGATIVES

Soutenue le 15 décembre 1987 devant la Commission d'Examen :

M.	H. FARRENY	Président
Mme	P. DORIO	
MM.	J. NOAILLES	

R E S U M E

La première partie de ce document concerne le problème de l'aide à la réutilisation de logiciels existants. On montre d'abord l'intérêt potentiel de techniques d'intelligence artificielle dans cet objectif. Une étude de système expert est réalisée, suivie de la conception d'une maquette appliquée au domaine du calcul scientifique ; les connaissances sont modélisées en logique des prédicats .

Dans le cadre de l'expérimentation de cette maquette, le problème des explications négatives (pourquoi ne peut-on pas conclure tel fait ?), rarement traité lorsque les connaissances sont exprimées en logique des prédicats, a été soulevé ; pourtant cette fonction est aussi importante que la résolution de problème pour un système expert. Dans la seconde partie est donc proposée une méthode de génération d'explications négatives structurées par niveaux de détails, basée sur une représentation de la résolution par un arbre ET-OU.

MOTS CLES : Système expert
Réutilisation de logiciels
Explications négatives
Arbre ET-OU
Logique des prédicats

ABSTRACT

The first part of this document deals with the problem of aiding in reusing existing software components. The possible interest of artificial intelligence techniques for this aim is first shown. The feasability for an expert system has been studied, and then a prototype in the field of numerical mathematics has been designed ; knowledge is formalized within predicate logic.

During the experimentation of this prototype, the problem of negative explanation generation (why can't be this goal deduced ?), seldom dealt with when knowledge is formalized within predicate logic, has been focussed on ; this function is as important as problem solving for an expert system. So in the second part, a method which generates negative explanations, structured by levels of detail and based upon a representation of the search space by an AND/OR tree, is proposed.

KEE WORDS : Expert system

Reusing software components

Negative explanations

AND/OR tree

Predicate logic

Je remercie vivement chaque membre du jury pour s'être intéressé à mon travail et avoir accepté de le juger :

M. Henri FARRENY, Professeur à l'INP de Toulouse, qui me fait l'honneur de présider ce jury,

M. Patrick SALLE, Professeur à l'INP de Toulouse, qui a accepté courageusement d'être rapporteur de cette thèse,

M. Martial VIVET, Maître de conférences à l'Université du Maine, qui s'est vivement intéressé à ce travail dès ses débuts, l'a encouragé, et a bien voulu en être rapporteur,

M. Guy ZANON, Ingénieur au Département d'Etudes et de Recherches en Informatique du CERT à Toulouse, qui a dirigé ses travaux avec efficacité, rigueur et discrétion,

Mme Paulette DORIO, Chef du Département Mathématiques et Analyse Numérique au Centre National d'Etudes Spatiales à Toulouse, pour avoir fourni le thème initial de ce travail et s'être intéressée à ses suites,

M. Joseph NOAILLES, Professeur à l'INP de Toulouse, qui a spontanément manifesté de l'enthousiasme envers le sujet traité.

Toute ma reconnaissance va à M. François-Régis VALETTE, Chef du département, pour m'avoir permis de mener à bien ce travail dans les meilleures conditions.

Je remercie également tous les membres du DERI qui ont su créer une atmosphère de travail très chaleureuse, par des discussions plus ou moins techniques ... ; en particulier, merci à M. Jack FOISSEAU pour ses encouragements, surtout dans les moments de doute, et merci aux équipes d'Intelligence Artificielle, de Logique et Bases de Données, de Génie Logiciel, et aux autres ....

Je tiens d'autre part à exprimer ma profonde reconnaissance à M. LASALLE-BALIER et Mme SOUBRY, du CNES de Toulouse, pour m'avoir aidée à recenser les besoins auxquels se veut de répondre la première partie de

ce travail. PELAGIE se joint à moi pour remercier aussi M. LAGOUANELLE, de l'Université Paul Sabatier de Toulouse, pour avoir accepté très gentiment de lui faire part de son expertise numéricienne.

Mes remerciements vont aussi à Mesdames Simone CALONNEC et Christiane PAYRAU, pour le soin, la gentillesse et la compétence avec lesquels elles ont assuré la dactylographie de cette thèse.

La réalisation matérielle de cet ouvrage revient au Service d'Edition du CERT ; qu'il trouve ici l'expression de mes sincères remerciements.

Enfin, je remercie le CERT et le CNES de Toulouse pour leur soutien financier.

A mes petits hommes

TABLE DES MATIERES

INTRODUCTION GENERALE..... 23

1ÈRE PARTIE : **CONCEPTION D'UN SYSTEME EXPERT D'AIDE A LA REUTILISATION DE LOGICIELS SCIENTIFIQUES**..... 25

CHAPITRE I - *PRESENTATION INFORMELLE DU PROBLEME*..... 27

I.1 - Situation de l'étude dans un contexte Génie Logiciel..... 27
I.1.1 - Environnements d'assistance à la conception de gros logiciels..... 27
I.1.2 - Situation du problème dans ce processus..... 29
I.2 - Spécifications générales d'un logiciel d'aide au choix et à l'utilisation de modules d'une bibliothèque..... 30
I.2.1 - Fonctionnalités principales..... 30
I.2.2 - Principales caractéristiques de ce logiciel..... 30
I.2.3 - Différences avec un logiciel de recherche documentaire..... 31
I.2.4 - Conclusion : intérêt des techniques d'intelligence artificielle pour ce type de logiciel..... 32
I.3 - Exemple d'application dans le domaine de l'analyse numérique
I.3.1 - Présentation du contexte réel de l'étude..... 35
I.3.1.1 - Description de l'existant..... 35
I.3.1.2 - Problèmes soulevés dans ce contexte..... 36
I.3.2 - Définition informelle des objectifs de l'étude..... 38
I.3.2.1 - Fonctionnalités du système informatique envisagé..... 38
I.3.2.1.1 - Usager..... 38
I.3.2.1.2 - Gestionnaire..... 39
I.3.2.2 - Conclusion : orientation vers une solution de type système expert..... 40

CHAPITRE II - *SYSTEMES EXPERTS ET MATHEMATIQUES*..... 43

II.1 - Généralités sur les systèmes experts..... 43
II.1.1 - Caractérisation des logiciels de type "système expert"..... 43
II.1.2 - Architecture classique des systèmes experts..... 44
II.1.3 - Expression des connaissances..... 47
II.1.3.1 - Les différents types de formalismes..... 47
II.1.3.2 - Description des formalismes..... 47
II.1.3.2.1 - Règles de production..... 48
II.1.3.2.2 - Réseaux sémantiques..... 48
II.1.3.2.3 - Structures ou séquences-types..... 49
II.1.3.2.4 - Combinaison entre différents formalismes..... 50
II.1.3.3 - Notions annexes : incertitude, méta-connaissances..... 51
II.1.3.3.1 - Traitement de l'incertitude. 52
II.1.3.3.2 - Méta-connaissances..... 52
II.1.4 - Stratégies d'inférences..... 53
II.1.4.1 - Principe de fonctionnement d'un moteur d'inférences..... 53
II.1.4.2 - Critères d'efficacité d'un moteur d'inférences..... 53
II.1.4.3 - Stratégies de recherche d'une solution à un problème..... 54
II.1.4.4 - Principales difficultés mal résolues..... 55
II.1.5 - Module d'interface..... 56
II.1.5.1 - Fonctionnalités..... 56
II.1.5.2 - Différents niveaux d'interfaces possibles 57
II.1.6 - Aperçu de l'existant..... 58
II.1.6.1 - Langages de programmation..... 58
II.1.6.2 - Micro-revue de logiciels utilisés de type Système Expert..... 59
II.1.6.3 - Limites actuelles des systèmes experts. 59
II.2 - Littérature concernant notre domaine d'application..... 60
II.2.1 - Caractéristiques des bibliothèques scientifiques... 61
II.2.1.1 - Intérêt de telles bibliothèques..... 61

II.2.1.2 - Difficultés inhérentes à ces bibliothèques..... 61
II.2.2 - Systèmes de résolution de problèmes mathématiques 63
II.2.2.1 - Résolution numérique autour d'une bibliothèque..... 63
II.2.2.1.1 - Un outil général PROTRAN.. 63
II.2.2.1.2 - Un outil spécialisé dans le calcul d'éléments finis : FLUX-EXPERT..... 64
II.2.2.2 - Aide à la résolution en calcul algébrique 65
II.2.2.2.1 - L'approche calcul algébrique 65
II.2.2.2.2 - L'approche raisonnement artificiel..... 66
II.2.2.2.3 - Mixage des deux approches précédentes..... 67
II.2.2.3 - Situation de notre projet par rapport à ces approches..... 67

CHAPITRE III - *CONCEPTION D'UN SYSTEME D'ASSISTANCE A L'UTILISATION DE LOGICIELS SCIENTIFIQUES*..... 69

III.1 - Spécifications détaillées du système..... 69
III.1.1 - Avant propos..... 69
III.1.1.1 - Justification de privilègesà l'usager et à la résolution de problèmes..... 70
III.1.1.2 - Philosophie du système devant les cas d'échec..... 71
III.1.1.3 - Limites du système en matière de résolution..... 72
III.1.2 - Fonction documentaire..... 72
III.1.2.1 - Objectifs..... 73
III.1.2.2 - Présentation des principaux concepts de la base de connaissances..... 73
III.1.2.3 - Représentation externe des objets de la base de connaissances..... 76
III.1.2.3.1 - Principaux champs des structures décrivant les objets mathématiques... 77

III.1.2.3.2 - Principaux champs de la structure module..... 83
III.1.2.4 - Discussion sur la dépendance de la modélisation des objets vis-à-vis du domaine..... 91
III.1.3 - Fonction résolution..... 92
III.1.3.1 - Formalisation du problème de la résolution.. 92
III.1.3.2 - Différentes connaissances utilisées pour résoudre un ou des problème(s)..... 94
III.1.3.3 - Formalisation externe..... 97
III.1.3.4 - Expression de l'incertitude sur les solutions proposées par le système (ou l'expert) 97
III.1.3.4.1 - Méthode d'expression retenue..... 98
III.1.3.4.2 - Limites inhérentes à ce mode d'expression..... 99
III.1.3.5 - Stratégies de résolution du système..... 99
III.1.3.5.1 - Résolution automatique.....100
III.1.3.5.2 - Résolution assistée par l'usager.....107
III.1.3.5.3 - A propos d'apprentissage.....109
III.1.4 - Choix de PROLOG.....109
III.1.4.1 - Choix de la logique du premier ordre pour la représentation interne des connaissances....109
III.1.4.2 - Pourquoi un langage de type PROLOG ?.....110
III.1.4.3 - Représentation des informations de la base de connaissances.....111
III.1.4.3.1 - Les objets.....111
III.1.4.3.2 - Les règles.....111
III.1.4.4 - Remarque : ordre des règles.....111
III.1.5 - Interface utilisateurs/système.....112
III.1.5.1 - Spécifications générales.....112
III.1.5.1.1 - Interface usager.....112
III.1.5.1.2 - Interface gestionnaire.....113
III.1.5.2 - Explications liées au raisonnement du système.....115
III.1.5.2.1 - Généralités.....115
III.1.5.2.2 - Comment ? 116
III.1.5.2.3 - Pourquoi ? 116
III.1.5.2.4 - Pourquoi pas ? 117

III.1.6 - Interactions entre les fonctions documentaire et résolution.....118
III.1.6.1 - Points d'interactions.....118
III.1.6.2 - Conséquence : remarque sur la représentation des connaissances.....119
III.2 - Réalisation d'une maquette : PELAGIE.....119
III.2.1 - Acquisition des connaissances expertes.....120
III.2.1.1 - Généralités.....120
III.2.1.2 - Méthodologie de l'acquisition.....121
III.2.1.2.1 - Méthode de transfert.....121
III.2.1.2.2 - Introduction des coefficients de réussite.....122
III.2.1.2.3 - Remarques.....123
III.2.2 - Implémentation de la fonction résolution.....123
III.2.2.1 - Implémentation des règles (version de base).124
III.2.2.1.1 - Règles inter-objets.....124
III.2.2.1.2 - Règles de comparaison entre objets de même type..... 126
III.2.2.1.3 - Règles de déduction de propriétés.....127
III.2.2.1.4 - Prise en compte des outils de contrôle associés aux outils susceptibles de résoudre un problème.....127
III.2.2.2 - Version simplifiée de l'interpréteur de PELAGIE.....128
III.2.2.3 - Implémentation de la résolution de problème 130
III.2.2.3.1 - Fonction de recherche automatique d'un outil pour résoudre un problème.....131
III.2.2.3.2 - Fonction de vérification qu'un outil O de type T est adapté à la résolution d'un problème donné.....134
III.2.2.3.3 - Stratégie générale de résolution.....135
III.2.2.3.4 - Limites du traitement des échecs.....136
III.2.3 - Implémentation des explications.....137
III.2.3.1 - Comment ? 137
III.2.3.2 - Pourquoi ? 140
III.2.3.3 - Pourquoi pas ? 145
III.2.4 - Utilitaires annexes.....146
III.2.4.1 - Visualisation d'une règle.....146

III.2.4.2 - Modification des données entre deux résolutions successives.....147
III.2.4.3 - Quelques vérifications de cohérence.....147
III.2.5 - Caractéristiques physiques de la maquette PELAGIE148
III.2.5.1 - Matériel utilisé.....148
III.2.5.2 - Organisation physique du logiciel PELAGIE...149
III.2.5.2.1 - Répartition des codes LISP et PROLISP dans PELAGIE.....149
III.2.5.2.2 - Structure dynamique du logiciel.....149
III.2.5.3 - Performances réalisées par PELAGIE.....150
III.2.6 - Utilisation de la maquette PELAGIE.....151
III.2.6.1 - Fonctions documentaires (0,1,2,5,6).....152
III.2.6.2 - Fonction résolution de problème (3).....153
III.2.6.3 - Fonction d'accès aux données d'un problème(4).....154
III.3 - Conclusions en vue de la réalisation d'un système "grandeur nature".....154
III.3.1 - Choix du mode de représentation des connaissances155
III.3.2 - Remarques sur les coûts..... 156
III.3.2.1 - Coûts de développement..... 156
III.3.2.2 - Coûts en performances..... 156
III.3.3 - A propos du traitement des explications.....157
III.3.3.1 - Explications positives.....157
III.3.3.2 - Explications négatives (ou assimilé).....158
III.3.4 - Autres extensions possibles de l'étude.....159

2ÈME PARTIE : EXPLICATIONS NEGATIVES DANS LES SYSTEMES A BASES DE CONNAISSANCES FORMULEES EN LOGIQUE DES PREDICATS.....163

INTRODUCTION.....165

CHAPITRE I - *ETAT DE L'ART EN MATIERE D'EXPLICATIONS DANS LES SYSTEMES A BASE DE CONNAISSANCES*.....167

I.1 - Introduction.....167
I.2 - Techniques générales d'explications.....168
I.2.1 - Manipulations de l'arbre de résolution.....169

I.2.1.1 - Restitution de l'arbre.....169
I.2.1.2 - Simplification de l'arbre.....171
I.2.1.3 - Conclusion.....175
I.2.2 - Utilisation de connaissances autres que celles utilisées en résolution.....175
I.2.2.1 - Distinction entre différents types de connaissances du système.....175
I.2.2.2 - Connaissances dédiées à l'explication.....177
I.2.3 - Conclusion.....178
I.3 - Traitement des explications négatives.....179
I.3.1 - Les problèmes liés à la question pourquoi- pas.....179
I.3.2 - Approche orientée usager.....180
I.3.2.1 - Logique des propositions.....181
I.3.2.2 - Logique des prédicats.....183
I.3.3 - Approche orientée concepteur de la base de connaissances.....185
I.3.3.1 - Comparaison avec le débugging de programme logique.....185
I.3.3.2 - Outils de modification de base de connaissances.....186
I.3.4 - Génération d'hypothèses.....188
I.3.4.1 - Optique génération d'hypothèses.....188
I.3.4.1.1 - Présentation de la méthode de Morgan....188
I.3.4.1.2 - Exemple d'application de la f-résolution en génération d'hypothèses.....191
I.3.4.2 - Optique résolution avec hypothèses.....194
I.3.5 - Théorie des types pour un langage de type PROLOG...196
I.3.5.1 - Introduction.....196
I.3.5.2 - Une théorie des types "automatique" en PROLOG 197
I.3.5.3 - Un schéma de types polymorphes en PROLOG.....199
I.3.5.4 - Conclusion sur l'intérêt des théories des types pour les explications négatives.....201
I.4 - Conclusion.....202

CHAPITRE II - *RESOLUTION DU PROBLEME DES EXPLICATIONS NEGATIVES*205

II.1 - Définition et contexte de résolution de notre problème.....205
II.1.1 - Informations utilisées.....205

II.1.2 - Caractéristiques de la solution cherchée.....206
II.1.3 - Hypothèses de travail.....209
II.2 - Une première approche, à mi-chemin entre la génération d'hypothèses et le contrôle de type..... 209
II.2.1 - Présentation générale de la méthode.....209
II.2.1.1 - Terminologie adoptée.....210
II.2.1.2 - Génération d'un ensemble de faits manquants..211
II.2.1.3 - Réduction de l'ensemble des faits manquants..212
II.2.1.4 - Problèmes stratégiques liés à la méthode.....213
II.2.2 - Critique de la méthode.....214
II.2.2.1 - Exemple.....214
II.2.2.2 - Critiques à partir de l'exemple.....216
II.2.2.3 - Conclusion.....218
II.3 - Une méthode de génération d'explications négatives structurées, basées sur le principe de résolution.....219
II.3.1 - Motivations.....219
II.3.2 - Cas particulier de la logique des propositions....220
II.3.2.1 - Principe général de la méthode.....220
II.3.2.2 - Calcul d'une expression logique permettant d'expliquer $\neg P$ sous les hypothèses décrites en II.3.2.1.....222
II.3.2.3 - Application sur un exemple.....225
II.3.2.4 - Conclusion.....226
II.3.3 - Généralisation du processus à la logique des prédicats restreinte aux clauses de Horn.....227
II.3.3.1 - Le principe dans ses grandes étapes.....227
II.3.3.2 - Normalisation des clauses formant la base de connaissances initiale BdC.....228
II.3.3.2.1 - Motivations.....228
II.3.3.2.2 - Règles de normalisation des clauses....230
II.3.3.3 - Elaboration de l'arbre ET-OU de résolution de $P(x_0,...)$.....232
II.3.3.3.1 - Caractéristiques de l'arbre souhaité...233
II.3.3.3.2 - Constitution de l'arbre de résolution ET- OU de racine P(x,...).....238
II.3.3.4 - Explication de l'échec sur P(x,...) à partir des feuilles en échec de l'arbre.....243
II.3.3.4.1 - Introduction : l'idée intuitive.....243

II.3.3.4.2 - Constitution de l'expression logique associée à une feuille en échec de l'arbre.....247
II.3.3.4.3 - Constitution de l'expression générale exprimant que l'arbre de résolution est en échec.....262
II.3.3.5 - Formation de la formule finale expliquant l'échec.....267
II.3.3.6 - Un exemple complet d'application de la méthode.....268
II.3.3.6.1 - Normalisation des clauses du programme 268
II.3.3.6.2 - Construction de l'arbre ET_OU de résolution du couple (Paul, Marie).....268
II.3.3.6.3 - Expressions logiques caractérisant les feuilles en échec.....270
II.3.3.6.4 - Expression complexe traduisant l'échec sur l'arbre.....271
II.3.3.6.5 - Expression expliquant l'échec sur couple(Paul,Marie).....272
II.3.3.6.6 - Exploitation de cette expression complexe pour générer des explications négatives structurées par niveau, de couple(Paul,Marie).....273
II.3.3.7 - Résultats obtenus avec cette méthode sur l'exemple pris en II.3.3.3.1.....274
II.4 - Conclusion.....277

CONCLUSION GENERALE.....279

BIBLIOGRAPHIE.....281

ANNEXES.....293

ANNEXE 1 - Tableaux récapitulatifs des principales informations contenues dans la base de connaissances de PELAGIE..... 295

ANNEXE 2 - Projet de fichier de mémorisation des sessions usagers 311

ANNEXE 3 - Composition du code de PELAGIE..... 317

ANNEXE 4 - Exemples de sessions avec PELAGIE..... 319

INTRODUCTION GENERALE

Les travaux que nous présentons ici sont nés à partir de besoins très concrets. Dans un premier temps, nous nous sommes intéressés au problème de l'aide à la sélection et à la réutilisation de sous-programmes appartenant à diverses bibliothèques, lors de la conception d'un logiciel scientifique. Après une étude préalable du problème, nous avons essayé de voir dans quelle mesure des techniques utilisées en intelligence artificielle peuvent être intéressantes dans ce contexte. Nous avons conçu ainsi un projet de système expert qui permettrait de faciliter la tâche des programmeurs, dans le domaine de l'analyse numérique, les connaissances étant formalisées en logique des prédicats restreinte aux clauses de Horn. Pour montrer la faisabilité de ce projet, nous avons réalisé une maquette (PELAGIE) spécialisée en algèbre linéaire.

Au cours de cette étude, un de nos objectifs était de générer des explications sur le raisonnement du système. Nous nous sommes alors heurtés au problème des explications négatives (pourquoi le système n'a pas trouvé telle solution à tel problème ?) dans le cadre du formalisme des clauses de Horn. Dans la seconde partie de ce document, nous nous intéresserons donc à ce sujet. Une étude bibliographique nous permettra de constater que relativement peu de travaux ont porté directement là-dessus jusqu'à maintenant. Nous proposerons alors les grandes lignes d'une première méthode qui s'est avérée conduire à une impasse sur un plan pratique ; toutefois, cela nous permettra de justifier l'approche que nous avons finalement adoptée.

Cette dernière méthode nous permet de générer des explications structurées par niveaux de détails, et s'appuie sur une technique de représentation de la résolution d'un problème par un arbre ET-OU. Nous en illustrerons notre présentation par de nombreux exemples d'application. Signalons que dans toute cette seconde partie, les problèmes d'implémentation n'ont pas été abordés.

1ÈRE PARTIE

CONCEPTION D'UN SYSTEME EXPERT D'AIDE

A LA REUTILISATION DE LOGICIELS SCIENTIFIQUES

I. - PRESENTATION INFORMELLE DU PROBLEME

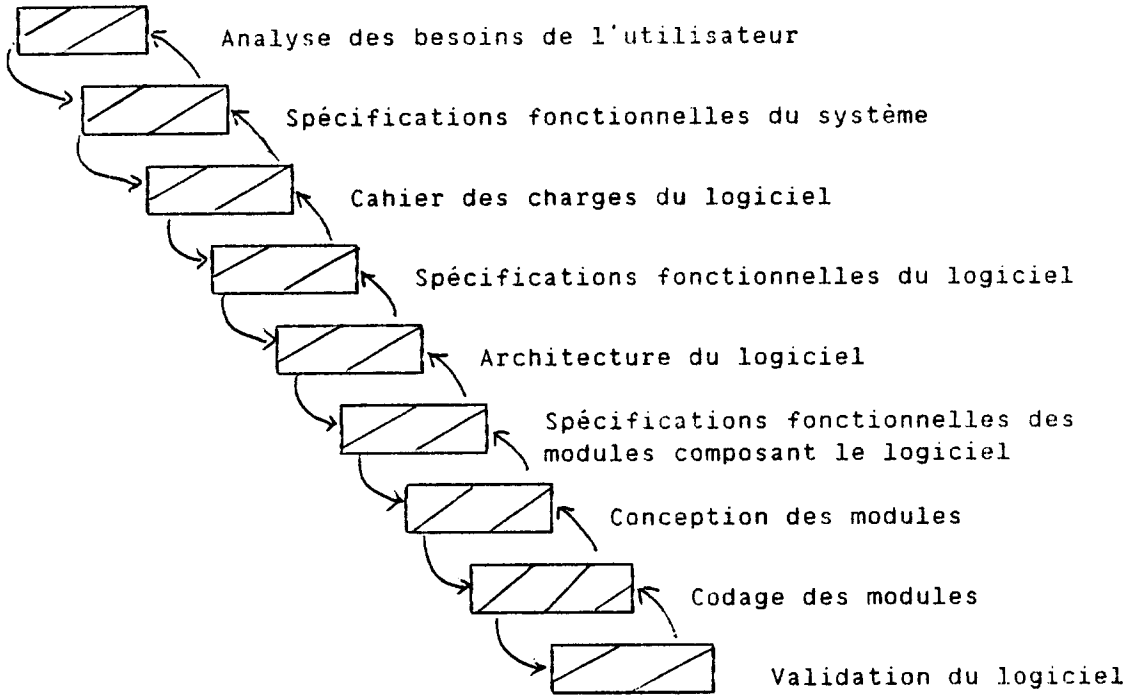
Dans cette première partie, nous commençons par situer puis poser le problème général auquel nous nous intéresserons ; nous montrerons que cette étude peut être considérée aussi bien d'un point de vue génie logiciel qu'intelligence artificielle - les frontières entre ces deux domaines étant d'ailleurs mal définies. Puis nous présentons une application réelle sur laquelle nous travaillerons par la suite.

I.1 - SITUATION DE L'ETUDE DANS UN CONTEXTE GENIE LOGICIEL

I.1.1 - Environnements d'assistance à la conception de gros logiciels

De nombreux travaux visent à permettre la génération de logiciels complexes et volumineux qui répondent effectivement aux spécifications initiales, et qui soient relativement faciles à tester, corriger, puis à maintenir. Une approche envisagée consiste à définir un environnement d'assistance à la conception de logiciels, soulageant l'utilisateur autant que possible, pour accroître la qualité des logiciels ainsi obtenus.

Dans de tels environnements, on distingue généralement différentes étapes (cf. Figure 1). Notons que cette décomposition reflète un processus par raffinements successifs, et qu'à chaque niveau, on doit vérifier que les résultats obtenus correspondent effectivement aux spécifications établies au niveau précédent (/FOI 80/).



Principales étapes dans un processus d'assistance à la production de logiciels.

Figure 1

Un logiciel d'environnement d'assistance à la conception de logiciels est souvent constitué de plusieurs modules coopérant donc ensemble, un module étant associé à une étape. Pour répondre avec efficacité aux besoins des utilisateurs, ces outils doivent vérifier certaines caractéristiques.

Par exemple, il peut être intéressant que l'utilisateur puisse décrire des connaissances sur le domaine d'application, pour définir certaines contraintes de réalisation. Un logiciel d'environnement est également amené à traiter beaucoup de connaissances de toutes sortes : méthodes de décompositions de logiciels, outils fiables déjà utilisés lors de conceptions antérieures de logiciels (évitant la réécriture de code), etc....

D'autre part, l'interface de ces outils doit être assez souple. Il sera en général interactif ; cependant l'utilisateur devrait pouvoir se décharger assez souvent de certains travaux, pouvant s'effectuer de façon automatique, à la demande. De plus, une condition importante de succès est la capacité de tels outils à expliquer leur démarche ou leurs résultats à l'utilisateur.

Une autre caractéristique importante de ces logiciels est celle de l'évolutivité : leurs connaissances doivent facilement pouvoir être mises à jour, et ceci doit être fait à moindre coût.

Cette liste des propriétés est loin d'être exhaustive : seules celles qui nous intéressent pour la suite ont été mentionnées.

I.1.2 - Situation du problème dans ce processus

Notre travail se situe après les spécifications fonctionnelles des modules devant composer un logiciel. Il s'agit de réduire au maximum le travail de conception et réalisation des modules. Pour cela, une solution consiste à trouver des modules existant, fiables, et répondant aux spécifications désirées, dans des bibliothèques de logiciels disponibles. Ces modules pourront alors être réutilisés, en étant intégrés dans le logiciel demandé par l'utilisateur.

Cette démarche doit permettre un gain de temps considérable, en luttant contre la tendance de chaque programmeur à rebâtir sans arrêt son univers de logiciels sans se préoccuper de savoir s'il n'existerait pas déjà des outils répondant à ses besoins. D'autre part, cela permet d'obtenir des logiciels plus fiables et de meilleure qualité, puisque ces modules ont théoriquement déjà fait l'objet de nombreux tests.

Insistons sur le fait qu'il s'agit de choisir des modules lors de la conception du logiciel général : c'est un choix statique. En effet, un autre problème serait celui du choix dynamique de module, en cours d'exécution du logiciel général ; là, le choix pourrait dépendre des résultats obtenus lors de l'exécution jusqu'au point de recherche du module. Les deux problèmes sont bien différents.

Après avoir situé et rapidement posé le problème général, nous allons maintenant le caractériser.

I.2 - SPECIFICATIONS GENERALES D'UN LOGICIEL D'AIDE AU CHOIX ET A L'UTILISATION DE MODULES D'UNE BIBLIOTHEQUE

Nous examinons ici les propriétés générales que devrait vérifier un logiciel d'aide au choix de modules dans une bibliothèque. Nous en tirerons ensuite des conclusions quant au type de techniques à mettre en oeuvre pour réaliser un tel logiciel.

I.2.1 - Fonctionnalités principales

Nous pouvons distinguer deux catégories d'utilisateurs : la première consultera le système pour trouver un ou des logiciels répondant à certaines spécifications ; la seconde se chargera d'alimenter le système de façon à ce qu'il réponde aux besoins de la première catégorie, en mettant à jour ses connaissances au fil du temps. Par la suite, nous désignerons le premier type d'utilisateurs par usager, et le second type par gestionnaire.

I.2.2 - Principales caractéristiques de ce logiciel

Un tel logiciel doit tout d'abord disposer de beaucoup d'informations pour être efficace. Ces informations, ou connaissances, sont également de nature hétérogène ; elles peuvent concerner les modules eux-mêmes, ou des concepts en amont du concept module : éléments de choix d'un module, classe de modules, connaissances sur les domaines d'application des modules, etc ... Ces connaissances doivent pouvoir être mises à jour facilement, et donc être d'accès direct dans le logiciel - d'autant plus que l'ensemble des modules à gérer évoluera vite et sans arrêt.

De plus, on peut être amené à vouloir exprimer des connaissances imprécises ; par exemple, il est possible de décrire les données ou la fonction d'un module, mais il est difficile d'exprimer une notion de qualité des résultats obtenus par l'application de ce module.

D'autre part, comme dans tout logiciel d'assistance à l'utilisateur, il est important que les résultats fournis par ce logiciel soient crédibles auprès de l'usager. La seule façon d'y parvenir est que ce logiciel soit capable d'expliquer et de justifier ses réponses.

Il est également intéressant qu'un tel logiciel ne soit pas trop directif ; l'usager doit pouvoir intervenir à son gré, de façon à orienter le choix du module dans un sens ou dans l'autre. L'idéal pour l'usager est donc de disposer d'un interface évolué, interactif et convivial.

Un autre point important concerne l'efficacité du logiciel. Il ne suffit pas qu'il réponde de façon satisfaisante aux spécifications données, mais encore doit-il le faire plus rapidement que ne le ferait l'homme. Or ce logiciel traitant beaucoup d'informations, nous risquons rapidement de nous trouver face à un problème d'explosion combinatoire. Le logiciel d'assistance doit donc mettre en oeuvre des techniques de recherche relativement astucieuses et efficaces pour éviter de tomber dans ce piège.

Enfin, il serait intéressant qu'un tel logiciel soit indépendant du domaine d'application de l'utilisateur. Toutefois, remarquons que sa convivialité, son efficacité et son succès seront probablement d'autant plus importants que la conception de certaines parties de ce logiciel aura tenu compte de caractéristiques du domaine d'application. Un compromis est donc à trouver sur ce facteur de la dépendance avec le domaine.

Ces caractéristiques auront une influence certaine sur le type de techniques à mettre en oeuvre pour réaliser un tel logiciel.

I.2.3 - Différences avec un logiciel de recherche documentaire

A priori, on peut se demander si une technologie assez classique, telle que celles mises en oeuvre dans des logiciels de recherche documentaire, ne pourrait pas suffire pour un système d'aide à

l'utilisation de bibliothèques.

Cependant, il existe quelques différences notoires entre ces deux types de logiciels. D'une part, dans un système documentaire, l'utilisateur du système spécifie l'objet de sa recherche en donnant des critères syntaxiques (mots clés) très précis, et ces seuls critères permettent d'accéder directement aux objets recherchés. Dans le logiciel que nous voulons réaliser, l'usager ne définit d'abord le problème que dans ses grandes lignes et c'est le système qui, au cours de sa recherche, lui demandera si telle ou telle contrainte est vérifiée par les données du problème (parfois, l'usager ignorera même la réponse) : c'est au système de trouver les critères nécessaires et suffisants qui lui permettront de résoudre le problème de l'usager ; de plus on s'intéresse à des critères plus sémantiques.

D'autre part, d'autres différences sont : les capacités d'auto-explications, de justifications des résultats, et éventuellement la possibilité de traiter des informations imprécises.

Ce sont ces différences qui justifient le recours à des techniques différentes pour la conception de notre logiciel.

I.2.4 - Conclusion : intérêt des techniques d'Intelligence Artificielle pour ce type de logiciel

Nous avons montré la nécessité de disposer de techniques judicieuses de recherche de solution à un problème. Différents algorithmes développés dans le cadre de l'intelligence artificielle pourraient donc répondre à notre besoin ; encore faut-il voir quels seraient les mieux adaptés (/NIL 80/).

D'autre part, nous voulons pouvoir facilement mettre à jour les connaissances utilisées par le logiciel. Si nous utilisons une programmation classique de type procédural, la modification (ajout ou suppression) d'informations implique une correction laborieuse et coûteuse d'une partie du programme, car les parties données et contrôle y sont intimement mêlées. Il serait donc souhaitable d'utiliser une programmation de type déclaratif : les informations et leur mode

d'utilisation sont alors deux entités séparées, et il devient possible de modifier l'une indépendamment de l'autre - donc à moindre coût.

Enfin, rappelons l'utilité d'un logiciel capable d'expliquer son propre raisonnement de manière dynamique à l'utilisateur. Cette capacité est rare avec les programmes de type procédural, à cause de la difficulté à distinguer les connaissances utilisées de la façon de les utiliser.

Ces constatations nous conduisent à penser qu'une technique déclarative de type "système expert", utilisant des heuristiques de recherche adéquates, peut être très intéressante pour ce type de logiciel. Nous vérifierons ceci dans la partie II.

I.3 - EXEMPLE D'APPLICATION DANS LE DOMAINE DE L'ANALYSE NUMERIQUE

Nous présentons ici un exemple concret issu de besoins réels, qui nous servira de base de travail pour la suite.

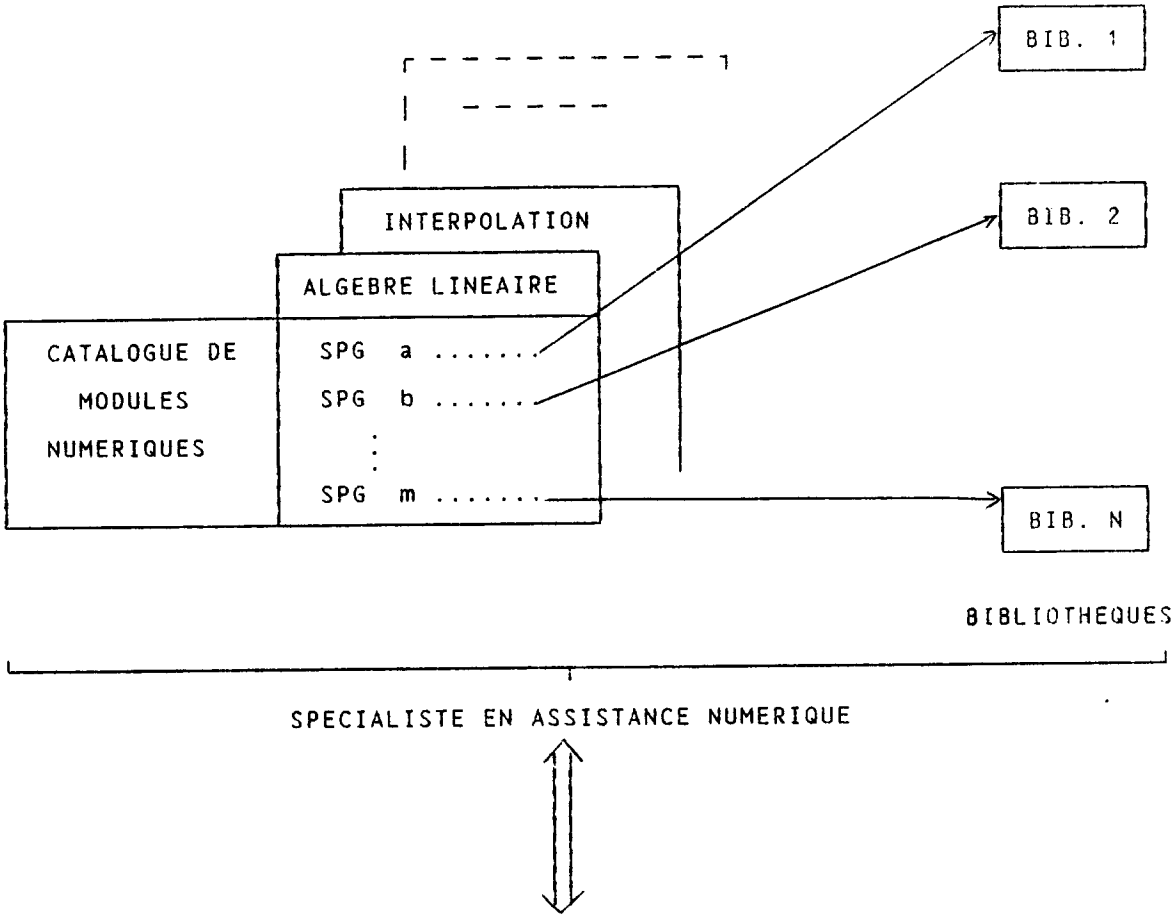
I.3.1 - Présentation du contexte réel de l'étude

I.3.1.1 - Description de l'existant

Plaçons-nous dans le cadre d'un service de physiciens, chimistes ou autres, non spécialistes en analyse numérique. L'un d'eux a un logiciel scientifique à concevoir, dans lequel il doit inverser une matrice. Pour faciliter sa tâche, il a à sa disposition des bibliothèques de logiciels en analyse numérique ; elles contiennent des descriptions complètes et non uniformes des modules. Par commodité, ceux-ci sont recensés dans un catalogue, classé par thèmes puis par ordre alphabétique. Dans ce catalogue, chaque module est décrit très succintement (méthode mise en oeuvre, fonction, principales limites d'utilisation, quelques mots sur les performances d'exécution), et l'on y indique où trouver sa notice d'utilisation détaillée (cf.figure 2).

Pour l'aider à chercher l'outil qui lui conviendra le mieux, notre physicien peut faire appel à un spécialiste en analyse numérique. Celui-ci connait les éléments des bibliothèques numériques, et dispose de connaissances théoriques importantes dans le domaine. Il peut aider également à obtenir de meilleurs résultats en conseillant de reformuler un problème, de mieux conditionner des données, etc...

Ce contexte est décrit de façon plus détaillée dans (/SAU 84.1/).



Description de l'existant

Figure 2

I.3.1.2 - Problèmes soulevés dans ce contexte

Une des premières difficultés liées à cette situation provient de la nature même des bibliothèques. Elles contiennent un nombre considérable de modules (plus de 2000). De plus, elles sont mises à jour fréquemment : correction ou amélioration de certains modules - d'où modification de leurs caractéristiques, suppression ou ajout d'autres modules. A tout moment, le spécialiste en assistance numérique doit parfaitement connaître tous les produits dont il dispose : ceci n'est pas trivial.

En réalité, il arrive alors souvent que le spécialiste se cantonne à conseiller certains modules dont il sait qu'ils donnent généralement des résultats satisfaisants, sans prendre en compte les dernières nouveautés des bibliothèques, qui donneraient peut-être de meilleurs résultats. L'assitance fournie n'est donc pas toujours rationnelle, et risque d'être assez statique.

D'autre part, le spécialiste peut difficilement s'assurer de la validité des conseils qu'il a prodigués. En effet, ses consultants ne se rendent compte qu'un logiciel choisi a posé des difficultés à l'exécution, que lorsqu'une situation fatale a été engendrée, telle qu'un débordement de place mémoire ; dans ces cas seulement, ils reviendront consulter le spécialiste. Le reste du temps, même si les résultats obtenus ne sont pas très bons, il n'y a pas de retour d'information du physicien vers le spécialiste. Tout le monde y perd : les physiciens n'ont pas de bons résultats, et la tâche du spécialiste n'est ni enrichissante ni motivante ; à la longue, sa qualité risque de se détériorer. Cet état de fait entraîne plusieurs conséquences négatives.

Suite au manque de dialogue entre spécialiste et consultants, le premier n'a pas toujours connaissance complète des domaines faisant l'objet d'une forte demande ; de plus, une analyse des résultats obtenus avec des modules souvent appelés serait utile pour que les efforts d'amélioration des bibliothèques soient orientés dans certains secteurs plutôt que dans d'autres.

D'autre part, le spécialiste se lasse de son travail au bout de quelque temps et... va quitter son poste. Se pose alors le problème du transfert des connaissances entre le démissionnaire et son remplaçant ; cela s'accompagne d'une perte de temps, de compétences et d'efficacité.

Une telle situation ne satisfait personne : ni les physiciens ou consultants, ni les spécialistes. On va donc essayer d'y apporter une solution, grâce à un outil informatique adapté.

I.3.2 - Définition informelle des objectifs de l'étude

I.3.2.1 - Fonctionnalités du système informatique envisagé

Comme déjà vu en I.1.2, le système qui pourrait être envisagé comme remède à la situation décrite en I.3.1, a deux types de fonctionnalités : usager et gestionnaire. Nous détaillons donc ici ces fonctionnalités, considérées dans le contexte réel.

I.3.2.1.1 - Usager

Le but du système est de remplacer au mieux le spécialiste en assistance numérique (ou tout au moins, l'assister pour lui laisser plus de temps sur des problèmes délicats).

Il doit donc aider l'usager à choisir méthodes, algorithmes ou modules adaptés à son problème. Notons que, suite à la remarque faite en I.1.2, un problème se définit plutôt qualitativement : contraintes "mathématiques" sur les variables (ex : la fonction à interpoler est dérivable, l'ordre de la matrice à inverser vaut environ 30), et contraintes "de performances" exigées par l'usager (précision, place mémoire et temps d'exécution) ; l'usager ne donnera pas, par exemple, les coefficients numériques d'une matrice.

Lorsque le système ne trouve pas de solution parfaitement adaptée au problème de l'usager, il essaiera de proposer une solution "dégradée", satisfaisant à une plage de contraintes plus large que celle de l'usager ; c'est une tentative de prise en compte des échecs.

Le système doit également pouvoir justifier ses propositions à l'usager. Cette fonction d'explications est très importante, car elle seule peut permettre au système d'être aussi crédible que ne l'était le spécialiste en assistance numérique, puis d'être utilisé ensuite avec succès par les usagers.

Une autre fonction du système consiste à sensibiliser l'usager au problème du contrôle de la qualité numérique des résultats qu'il obtiendra en appliquant un outil (méthode, algorithme ou module) donné ;

en effet, on ne peut jamais être sûr qu'un algorithme à priori parfaitement adapté à un problème conduira à des résultats corrects, à cause principalement des instabilités numériques (erreurs dues à l'algorithme, et à la nature des données). Pour cela, le système peut suggérer à l'usager, si la nature des données le réclame, des conseils, des méthodes ou même des logiciels de contrôle, ces derniers étant associés à des modules particuliers ; un exemple de conseil de contrôle peut être de préconditionner une matrice donnée.

D'autre part, une fonction importante du système sera l'aide à l'utilisation d'une méthode, d'un algorithme ou d'un logiciel. Pour les logiciels, cela consiste à expliquer clairement à l'usager les notices d'utilisation qui existaient dans les bibliothèques : limitations de toutes sortes, variables d'appel, code d'appel (les modules sont codés en FORTRAN), caractéristiques précises sur la place mémoire occupée, etc... Pour faciliter la compréhension du mode d'utilisation d'un module, le système peut fournir à l'usager une batterie de jeux de tests, à titre d'exemple.

Le système pourra également assurer un rôle limité d'enseignement. D'une part, il peut donner des informations sur une méthode ou un algorithme : ceci rejoint le point précédent. D'autre part, il doit pouvoir expliquer à l'usager les termes de base du vocabulaire d'analyse numérique. Il pourrait aussi aider l'usager à reformuler un problème que le système ne sait pas résoudre, en un problème qu'il sait résoudre, lorsque ceci est possible (ex. transformer une équation différentielle d'ordre N en un système de N équations différentielles d'ordre 1). Notons toutefois qu'il ne s'agit pas d'un système d'enseignement assisté par ordinateur.

I.3.2.1.2 - Gestionnaire

Le rôle du gestionnaire du système est de faire en sorte que celui-ci réponde correctement et complètement aux besoins des usagers.

Pour répondre à cet objectif, une des fonctions du système est de permettre un suivi des historiques d'utilisation. Dans un premier temps, le système mémorisera les principales caractéristiques de chaque

session d'usager (problème, solution préconisée, cas d'échec, etc...). Des outils peuvent ensuite permettre d'analyser ces historiques, de façon à mieux cerner les besoins des usagers. La tâche du gestionnaire consiste donc en partie à exploiter les résultats de ces analyses.

Le système doit donc lui offrir tous les outils lui permettant de mettre continuellement à jour les informations nécessaires pour satisfaire l'usager. Il peut s'agir de renforcer les domaines faisant l'objet de fortes demandes des usagers, de faire part de mise à jour de certaines bibliothèques ; ou bien, on peut encore avoir besoin de modifier certaines caractéristiques d'un outil (les performances d'un module dépendant de la machine sur lequel il tourne). Il peut s'agir encore de devoir introduire les résultats de récents travaux sur un problème, car les techniques et les modes évoluent rapidement dans le domaine du numérique, etc...

D'autre part, il serait bon que le système permette facilement au gestionnaire de vérifier la consistance des informations qu'il a, ce qui constitue un vaste programme ...

Ces fonctions pour gestionnaire sont assez ambitieuses ... il s'agit là de "ce qu'il serait intéressant de réaliser".

I.3.2.2 - Conclusion : orientation vers une solution de type système expert

Un but du système est d'assister un spécialiste dans le domaine de l'analyse numérique ; on attend de lui qu'il offre à tout moment un bon niveau d'expertise, constant, prenant toujours en compte les dernières connaissances acquises dans le domaine ; ce système sera donc toujours amené à évoluer.

D'autre part, le nombre des modules étant déjà élevé, le volume des connaissances manipulées par le système sera important, et sans cesse amené à augmenter.

Enfin, outre les deux caractéristiques ci-dessus, notre système présente au moins toutes celles décrites pour un logiciel

général d'aide au choix et à l'utilisation de modules d'une bibliothèque - si ce n'est qu'il s'agit là de plusieurs bibliothèques ne possédant pas toutes les mêmes types d'informations, et qu'il faut donc disposer de techniques informatiques assez souples pour traiter de façon adéquate des informations très diverses et, de plus, s'exprimant difficilement de façon algorithmique.

En faisant référence à la conclusion établie en I.2.4, nous nous orientons donc vers une étude de logiciel utilisant les techniques de programmation mise en oeuvre dans les systèmes experts.

La partie suivante aura justement pour but de confirmer cette orientation.

II - SYSTEMES EXPERTS ET MATHEMATIQUES

L'objet de cette partie est de voir ce que nous propose à l'heure actuelle la technologie informatique pour résoudre notre problème, et comment situer notre approche par rapport à celles existant déjà. Pour cela, nous caractérisons d'abord les domaines de compétence des systèmes experts, ainsi que les différentes techniques qu'ils mettent en oeuvre ; ensuite, nous examinerons différentes approches abordées pour la résolution de problèmes mathématiques.

II.1 - GENERALITES SUR LES SYSTEMES EXPERTS

II.1.1 - Caractérisation des logiciels de type "Système Expert"

Un système expert est un logiciel généralement conçu pour venir en aide au raisonnement humain dans un domaine bien défini, ou encore pour remplacer un ou des expert(s) dans ce domaine. Nous allons donc caractériser le type de problème concernant ce logiciel.

Ce logiciel doit, comme un expert, disposer de connaissances de très haut niveau dans son domaine de compétence ; pour permettre une bonne efficacité, ce domaine doit donc être bien délimité.

D'autre part, les experts ont, en plus de connaissances livresques, beaucoup de connaissances empiriques acquises par l'expérience. Ces connaissances sont généralement difficiles à exprimer de façon algorithmique, non structurées dans la tête de l'expert, et sont emplies de notions d'incertitude ; ceci est caractéristique dans le domaine médical lorsqu'on veut par exemple exprimer que "en présence de tels symptômes, on peut faire l'hypothèse de tel ou tel diagnostic, de façon plus ou moins vraisemblable selon la force des symptômes". Enfin, on doit pouvoir mettre à jour les connaissances du logiciel, comme le fait l'expert qui acquiert de nouvelles connaissances au fur et à mesure du temps (concept d'évolutivité).

Une autre caractéristique de ce type de problème est qu'il ne s'agit pas, comme dans les hypothèses classiques, de trouver la réponse à une question posée par l'utilisateur, mais une des meilleures réponses possibles. De plus, on ne connait pas a priori la façon d'obtenir une réponse acceptable, il s'agit souvent de problèmes non déterministes. Pour obtenir une réponse, l'expert a une multitude de connaissances disponibles, et de façons de les utiliser ; un logiciel classique se heurterait donc souvent au problème de l'explosion combinatoire.

Enfin, ce logiciel doit aider un utilisateur dans un domaine spécialisé ; ceci demande qu'il offre un interface interactif très convivial, permettant de générer des explications concernant les connaissances utilisées pour donner les réponses demandées.

Il semble donc difficile pour un logiciel classique, statique, difficile à modifier une fois conçu, de répondre à ces exigences. Par contre, un logiciel utilisant des techniques d'intelligence artificielle semblerait pouvoir convenir. En effet, ces programmes peuvent se caractériser par une représentation symbolique des connaissances (autre que numérique) permettant de raisonner dessus, tout en établissant une correspondance avec le monde réel extérieur ; ces connaissances sont données au programme de façon déclarative ; ce type de représentation permet au programme de justifier ses réponses à l'utilisateur. D'autre part, ces logiciels utilisent des heuristiques pour résoudre les problèmes lorsque le moyen d'arriver à la solution n'est pas connu à l'avance ; ils sont également capables de raisonner sur des problèmes à données incomplètes, en introduisant la notion de vraisemblance attachée au résultat (cf. /BON 81/,/BON 84-a/).

Un système expert n'est donc qu'un logiciel appliquant certaines techniques de programmation développées en Intelligence Artificielle et devant répondre aux critères décrits ci-dessus. L'architecture de ce logiciel et les techniques qu'il applique sont décrites dans la suite de cette partie.

II.1.2 - Architecture classique des systèmes experts

Plusieurs idées maitresses sont sous-jacentes à cette

architecture.

Une originalité, par rapport aux systèmes classiques, consiste à pouvoir modifier les connaissances utilisées par le système, de façon très libre. L'architecture comprendra donc les composants distincts suivants : un module contenant les connaissances, et un autre chargé d'utiliser correctement ces connaissances pour répondre à une question de l'utilisateur (moteur d'inférences ou interpréteur), qui sont donc faciles à modifier indépendamment l'un de l'autre. Notons qu'en réalité il existe deux types de connaissances : les premières constituent le savoir de l'expert et sont indépendantes de tout contexte d'exécution ; les secondes sont les faits connus sur un problème en cours d'exécution, et dépendent donc des connaissances de l'expert et des données du problème. Les premières constituent la base de connaissances, les secondes sont regroupées dans la base de faits (ou "espace de travail").

Une idée importante, relative à la notion de système expert, est qu'un des buts visés est de pouvoir modifier le domaine d'application, donc la base de connaissances, d'un système sans toucher à son moteur d'inférences. Ceci renforce le choix de séparer autant que possible ces deux composants ; toutefois, un souci d'efficacité peut mettre un frein à ce projet.

D'autre part, l'interface du système joue un rôle primordial et doit, entre autre, pouvoir expliquer le raisonnement du système pour répondre à une question - ie expliquer quelles connaissances ont été utilisées, et comment elles l'ont été. Il doit aussi donner les réponses aux questions de l'usager, et permettre la mise à jour de la base de connaissances. L'interface constitue aussi un module à part.

Tout ceci explique l'architecture présentée dans la figure 3.

Si tous ces composants sont indépendants au point de vue contenu, il n'en est pas de même quant aux techniques mises en oeuvre pour les réaliser. Le premier choix à faire est celui de la technique de représentation des connaissances, car celles pour le contrôle ou l'utilisation des connaissances en dépendent ; de même, les techniques de réalisation de l'interface dépendent de celles mises en oeuvre pour

représenter et utiliser les connaissances (/PIN 81/) (cf. figure 3).

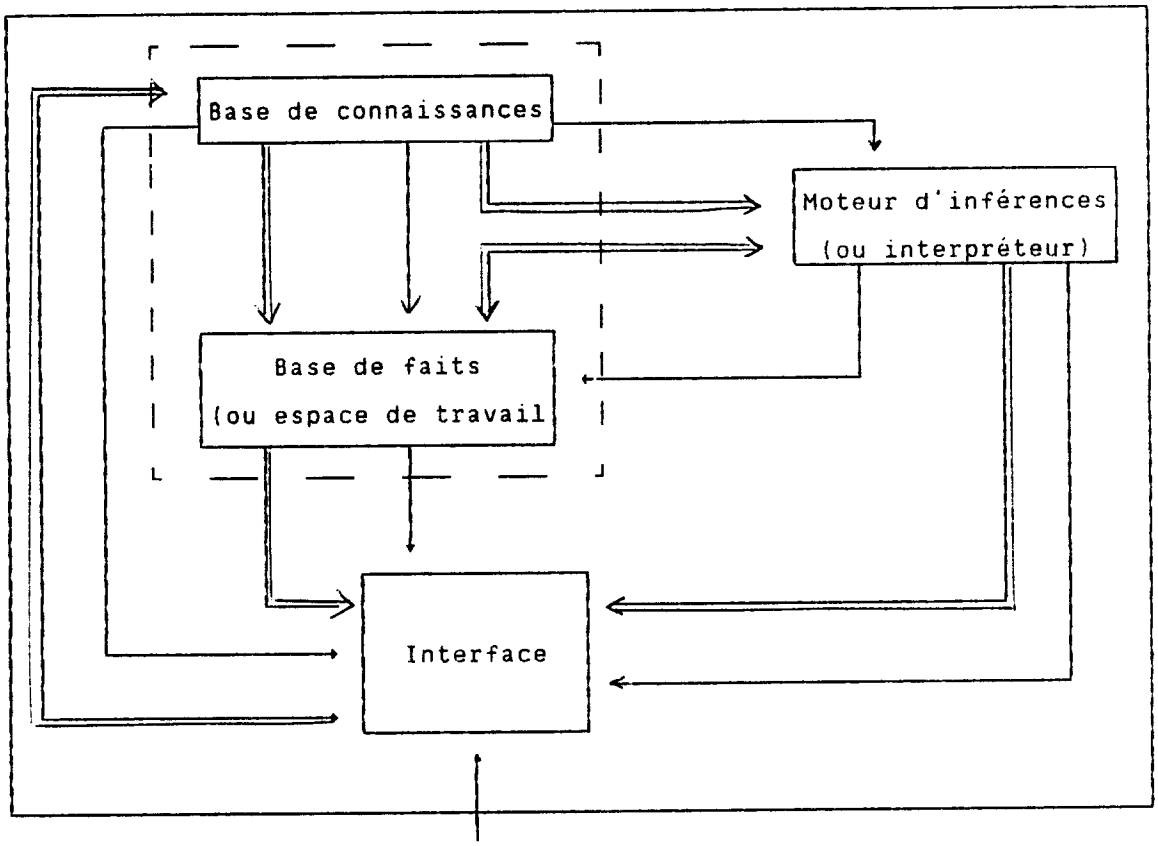
C'est donc dans cet ordre que nous examinerons rapidement les principales techniques de construction des systèmes experts.

Ceci explique aussi que si l'on veut changer le domaine d'application d'un système expert, l'efficacité ne peut être conservée que si les deux domaines ont des connaissances pouvant s'exprimer de façon analogue.

Légende

A ⟶ B = Les techniques de réalisation de B dépendent de celles de A

A ⟹ B = B travaille ou existe grâce à A



Architecture classique de système expert

Figure 3

II.1.3 - Expression des connaissances

Plusieurs techniques de base sont recensées à ce jour pour formaliser les connaissances. Nous avons vu en II.1.2 l'importance d'en choisir une judicieusement, puisqu'elle conditionne les choix à faire pour réaliser l'interpréteur et l'interface. Le choix se fait en fonction du domaine d'application et du type des connaissances à représenter. Nous verrons quels sont ces critères en présentant les diverses techniques.

II.1.3.1 - Les différents types de formalismes

Ils peuvent être classés de plusieurs façons. On distingue les modèles procéduraux (progammes, automates) des modèles déclaratifs (règles de production ou logique d'ordre 0 ou 1, réseaux sémantiques, frames ou scripts) ; nous ne parlerons ici que des déclaratifs. On peut distinguer également les modulaires (règles de production ou logiques d'ordre 0 ou 1), et les modèles en réseau (réseaux sémantiques, frames ou scripts) (/DEM 82/).

Nous avons vu que les représentations déclaratives répondaient mieux aux problèmes résolus par les systèmes experts. D'autre part, la modularité répond mieux au concept d'évolutivité exprimé en II.1.1, et paraît assez adaptée aux experts qui ont tendance à livrer leurs connaissances "en vrac" ; par contre, les représentations en réseau s'adaptent mieux aux domaines où les objets sont structurées, et où l'on doit décrire des relations entre ces objets.

Rappelons encore que, quelque soit le modèle choisi, le fragment le plus petit de la représentation doit correspondre au niveau de détail le plus bas des connaissances à exprimer (/PIN 82/)

II.1.3.2 - Description des formalismes

Nous aborderons les formalismes modulaires, puis ceux en réseaux.

II.1.3.2.1 - Règles de production

Les connaissances s'expriment sous forme de règles "si condition alors action". Pour choisir quelles règles appliquer, les conditions ou prémisses sont testées par rapport aux connaissances établies dans l'espace de travail. C'est l'interpréteur qui décide de l'ordre d'application des règles pour résoudre un problème, ou qui intervient en cas de conflit entre plusieurs règles applicables.

En réalité on associe souvent implicitement le terme "règles de production" aux règles exprimées en logique des propositions (ou d'ordre 0). En fait, la logique des prédicats (ou d'ordre 1) permet aussi d'exprimer des règles de production. On choisit la logique des prédicats dans le cas où une relation (ou prédicat) concerne plusieurs objets, définis implicitement par leur appartenance à un ensemble donné ; dans ces cas, une règle en logique d'ordre 1 correspond à une multitude de propositions en logique d'ordre 0, qui seraient nécessaires pour exprimer explicitement la même connaissance. A l'économie d'écriture résultant en logique d'ordre 1 correspond un interpréteur plus sophistiqué qu'en logique d'ordre 0 (prise en compte des variables, unification ...).

Le formalisme des règles a été le plus utilisé car il permet souvent aux experts d'exprimer leur savoir de façon assez naturelle (/HAY 85/, /LAU 82-1/) ; il est intéressant car il n'y a pas d'interaction entre les règles, il peut s'étendre sans faire apparaître d'effets de bord. Par contre, le système peut devenir lourd s'il possède beaucoup de règles sans pouvoir les compiler (surtout en logique des prédicats) ; d'autre part, il est encore très difficile d'assurer la consistance et la non-redondance logique d'un ensemble de règles, surtout en logique d'ordre 1.

II.1.3.2.2 - Réseaux sémantiques

Aussi appelés "réseaux associatifs", leur appellation courante est liée à leur utilisation fréquente pour traiter du langage naturel. Ce modèle est constitué d'un ensemble de noeuds, et de liens entre ces noeuds. Les liens ont des sens variant selon les applications :

relations d'inclusion, d'exclusion entre ensembles, d'appartenance, de temps (antériorité, postériorité ...), etc... (/FIN 79/). Le résultat d'une question sur la base de connaissances est obtenu en faisant l'intersection des réseaux concernant les divers objets (représentés par les noeuds) apparaissant dans la question.

Ce type de représentation permet facilement d'accéder à des objets donnés par index ; de plus, toutes les connaissances relatives à un objet sont dans son voisinage (dans le réseau) par hypothèse de construction. C'est pourquoi ceci est particulièrement adapté au travail sur le langage naturel, où l'on doit représenter des liens de diverses natures entre les objets du discours.

Il n'est pas facile de modifier une partie du réseau, car une modification peut avoir beaucoup d'incidences sur le reste du réseau, ou tout au moins sur son voisinage. D'autre part, sur une application concrète, un réseau peut vite devenir complexe ; la génération d'une réponse à une question devient alors plus laborieuse, et les explications du raisonnement utilisé deviennent alors très difficiles à obtenir de façon satisfaisante (ne serait-ce que par trace).

En réalité, à part dans le domaine du langage naturel, ce modèle seul est beaucoup moins utilisé que les précédents, car il répond mal à certaines exigences des utilisateurs des logiciels de type système expert.

II.1.3.2.3 - Structures ou séquences-types (frames ou scripts)

Les structures-types ont été introduites par Minsky en 1974 (/MIN 74/) pour décrire des situations stéréotypées, par exemple dans le domaine de la vision ; les séquences-types (Schank et Abelson, 1977), bâties sur le même principe, représentent des évènements.

Une structure représente un concept (ex : animal) ; les attributs de ce concept sont décrits dans les champs de cette structure ; les champs peuvent contenir des informations spécifiques telles que "valeur par défaut : ...", "valeur par exception : ...",

"valeurs possibles : ..." (facettes). Les structures se définissent les unes par rapport aux autres, se rapportant à des concepts de plus en plus particuliers (ex : structure animal, structure chien, structure Milou). Un champ spécial permet donc d'indiquer de quelle classe S1 une classe S2 est-elle la particularisation. Les structures de plus bas niveau doivent correspondre aux données des problèmes.

Ce formalisme permet de répondre rapidement aux questions portant sur les caractéristiques d'un objet donné, par examen des informations contenues dans les champs de la structure pouvant décrire le plus précisément cet objet. En effet, on dispose là de mécanismes d'inférences implicites entre les structures (héritage de propriétés). De plus, on peut raisonner sur des valeurs par défaut pour traiter les problèmes à données incomplètes. Les structures peuvent être enrichies par mémorisation de résultats acquis lors d'expériences antérieures, ce qui confère une certaine efficacité à ce formalisme. Elles permettent donc d'obtenir de bons résultats dans des domaines où les connaissances sont statiques et bien structurées.

Par contre, ce formalisme est assez rigide et la modification d'une structure est difficile (propagation dans la hiérarchie de structures) ; de même, l'acquisition des connaissances est un phénomène assez compliqué. D'autre part, la taille mémoire nécessaire pour une structure est importante, ce qui alourdit la gestion de l'espace mémoire. Enfin, ce formalisme ne facilite pas la génération d'explications des raisonnements mis en oeuvre pour répondre à une question.

Toutefois, il semble promis à un bon avenir, avec le développement des langages-objets ces dernières années (/BON 85/), pour analyser des séquences de langage naturel ou de vision.

II.1.3.2.4 - Combinaison entre différents formalismes

Nous avons vu que le formalisme des règles de production d'une part, des réseaux ou des structures d'autre part, étaient finalement assez complémentaires. Les règles de production autorisent des

modifications très faciles, et de faire des inférences de façon explicite ; de plus, la logique des prédicats permet d'établir une théorie cohérente, même avec des informations non complètement décrites. Par contre, il est difficile d'avoir une vision globale des concepts de la base de connaissances, car les informations y sont écrites de manière éclatée. Enfin, ce mode de représentation permet de générer des explications relativement facilement.

A l'inverse, les réseaux ou les structures sont adaptées à la description d'un univers figé, et supportent mal des modifications dues à l'évolution des connaissances au cours du temps. Le mécanisme d'inférence y est peu explicite, et il y est difficile de générer quelques explications claires d'un raisonnement. Par contre, ces modes de représentation offrent une vision synthétique des objets de l'univers décrit et des relations qui les lient.

Une solution de plus en plus adoptée consiste donc à combiner des types de formalismes différents, pour y gagner sur tous les plans. Par exemple, dans /FIN 79/, nous voyons comment les réseaux peuvent être associés à des petits programmes pour faciliter la modification de la base de connaissances tout en conservant les avantages des réseaux dans certains domaines ; McSkimin et Minker introduisent les réseaux dans la logique des prédicats et évitent de perdre du temps lorsqu'il s'agit de répondre à des questions inconsistantes. Brachman montre aussi comment un compromis entre la logique du premier ordre et les structures permet à la fois d'avoir une terminologie complexe et significative relativement au domaine d'application, et de traiter d'un monde à connaissance incomplète et évolutive (/BRA 82/). On cherche aussi de plus en plus à associer la logique des prédicats aux réseaux ou aux structures (/BEN 86/, /BON 85/, /FIK 85/). Toutefois, il s'agit de voies de travail entreprises assez récemment.

II.1.3.3 - Notions annexes : incertitude, méta-connaissances...

Nous ne dirons que quelques mots sur ces notions, qui font l'objet de travaux importants.

II.1.3.3.1 - Traitement de l'incertitude

Très souvent, les connaissances d'un expert sont empreintes de la notions d'incertitude. Dans le domaine médical, cela est flagrant : il est rare de pouvoir affirmer qu'en présence de tels symptômes, on puisse affirmer de façon absolue l'existence de telle maladie, en général il y a plusieurs possiblités, les unes plus probables que les autres.

Pour rendre compte de cela, la solution généralement adoptée consiste à associer, à chaque fragment représentant des connaissances, un coefficient qui peut avoir plusieurs sens (/PIN 81/) : probabilité, certitude, intérêt pour la résolution du problème, préférence de l'expert, etc... De plus, on peut imaginer que les données d'un problème soient également affectées d'incertitude ou d'imprécision (/TUR 86/). Il s'agit ensuite de combiner les coefficients des différents fragments apparaissant au cours d'un raisonnement, pour éventuellement décider vers quelle voie s'orienter en cours de résolution, ou bien pour connaître la validité de la solution proposée. Là encore, plusieurs méthodes sont utilisées (théorie des probabilités, théorie bayésienne, théorie des possibilités (/TES 84/),etc ...) selon les cas.

II.1.3.3.2 - Méta-connaissances

Ce sont des connaissances portant sur les connaissances d'un expert sur un domaine. Elles sont intéressantes lors de la résolution d'un problème pour accélérer la recherche d'une solution (ex : si plusieurs règles sont applicables, une méta-connaissance peut permettre de choisir quelle règle il faut plutôt appliquer) ; elles peuvent aussi permettre de générer des explications plus synthétiques sur les raisonnements effectués par le système (/BAR 77/), ou encore permettre à un système de faire de l'apprentissage (/DAV 84/) et de s'auto-enrichir. Il peut même y avoir plusieurs niveaux de méta-connaissance. Un des problèmes est de savoir comment exprimer cette méta-connaissance, selon le formalisme de représentation des connaissances adopté : les règles de production semblent encore faciliter la tâche.

II.1.4 - Stratégies d'inférences

Nous rappelons le principe de fonctionnement d'un moteur d'inférences, puis nous introduirons des critères de mesure d'efficacité d'un moteur. Nous verrons ensuite les différentes stratégies de recherche d'une solution à un problème ; nous traiterons essentiellement les systèmes à règles de production. Puis nous évoquerons les difficultés encore mal résolues.

II.1.4.1 - Principe de fonctionnement d'un moteur d'inférences

Le processus de recherche de solution à un problème est initialisé avec le but à atteindre ou problème à résoudre, et les données initiales de ce problème - le tout étant fixé par l'utilisateur, et stocké dans l'espace de travail - Nous parlerons surtout des moteurs utilisant des règles.

Le moteur fonctionne de manière cyclique. Au début d'un cycle, il y a une tâche à exécuter dans l'espace de travail. La première étape du cycle consiste à chercher quelles règles sont applicables pour exécuter la tâche ; la seconde étape est celle de la sélection de la règle qui sera appliquée ; lors de la dernière étape, le moteur ajoute l'effet de l'application de cette règle dans l'espace de travail : il s'agit en général d'une nouvelle tâche à exécuter, et un nouveau cycle commence (/LAU 84/, /COR 84/). Il s'agit donc de savoir dans quel ordre on s'y prend pour résoudre un problème, et comment s'opère la sélection d'une règle parmi un ensemble de règles applicables.

II.1.4.2 - Critères d'efficacité d'un moteur d'inférences

Ces critères font appel à des concepts introduits par Nilsson en 1971 :

- un système est complet si, pour un problème donné auquel il existe une solution, il est capable de trouver cette solution.

- s'il existe une solution à coût minimal de recherche et si le système la trouve, il est admissible.

- la pénétrance d'un moteur est le rapport L/T où L est la longueur du chemin de l'état initial-données au but-solution, et T est le nombre de noeuds générés en cours de recherche de solution.

Dans l'idéal, un moteur d'inférences doit donc être admissible ; il doit bien sûr au moins être complet. De plus, un moteur parfait doit avoir une pénétrance égale à 1, c'est-à-dire qu'il ne doit générer aucun noeud inutile au cours de sa recherche- (/PIN 81/). Pour cela, de bonnes heuristiques ou techniques de recherche devront être mises en oeuvre.

II.1.4.3 - Stratégies de recherche d'une solution à un problème

Deux grandes méthodes de résolution sont utilisées. La première consiste, à partir des données, à générer de nouveaux faits, et ainsi de suite jusqu'à aboutir au but recherché : c'est du chaînage avant ou dirigé par les données. La seconde consiste à décomposer le but en un ensemble de sous-buts, et ainsi de suite, jusqu'à mettre en évidence des sous...sous-buts dont la résolution est triviale avec les données : c'est le chaînage arrière ou dirigé par le but. En outre, certains systèmes utilisent des stratégies mixtes, alliant les deux types de chaînage pour obtenir plus d'efficacité.

Si les connaissances sont exprimées en logique, le chaînage avant est basé sur le Modus Ponens (de $\vdash p$ et $p \vdash q$, on déduit $\vdash q$) ; le chaînage arrière est lui basé sur le Modus Tollens (de $\vdash \neg p$ et $q \vdash p$, on déduit $\neg q$).

Le choix d'utiliser un chaînage avant ou arrière se fait en fonction du domaine, et des types de problèmes que le système devra traiter. Le chaînage avant déduit à tout instant tout ce qu'il peut déduire ; s'il n'est pas doté de quelque heuristique, il ne se focalise pas vers le but à atteindre et le raisonnement risque d'être assez long ; ce type de contrôle s'utilise lorsque l'on veut récolter le plus de renseignements possibles sur un sujet donné : il est souvent utilisé pour faire du diagnostic. Par contre, le chaînage arrière réduit plus

rapidement toutes les possibilités de recherche ; mais il y a risque de bouclage lorsque deux problèmes se réduisent simultanément l'un à l'autre ; on peut par exemple utiliser ce chaînage pour résoudre un problème précis de mathématiques.

De toutes façons, quelle que soit la méthode utilisée,il y a intérêt à éviter les chemins de recherche longs ou caduques. Pour cela, on développera le minimum de noeuds grâce à des heuristiques ou stratégies de recherche développées en intelligence artificielle. On dispose à cet effet de nombreuses méthodes plus ou moins "intelligentes", n'ayant pas toutes les mêmes propriétés quant aux facteurs de complétude et d'admissibilité (recherches en largeur, en profondeur, ordonnées ou non, méthodes opérant des réductions des espaces de recherche, Mini-Max ou $\alpha.\beta$, etc...) (/NIL 80/, /HAND/). Concrètement, on pourra les appliquer avec efficacité lors de la seconde étape d'un cycle du moteur.

II.1.4.4 - Principales difficultés mal résolues

Le traitement de l'incertitude sur les connaissances a déjà été mentionné en II.1.3.3.1. Les méthodes citées ci-dessus peuvent être combinées avec l'exploitation des facteurs de certitude attachés aux fragments de la connaissance ; le problème est alors de voir comment faire cette combinaison. Si les connaissances sont exprimées en règles de logique, le moteur peut combiner les coefficients attachés aux différentes règles apparues sur un chemin donné. S'il s'agit de réseaux ou de structures, c'est plus compliqué ; Schubert résoud par exemple ce problème en introduisant dans le réseau des noeuds intermédiaires pour représenter l'information incertaine (/FIN 79/).

D'autre part, les heuristiques appliquées pour l'instant ne sont pas très "intelligentes" ; le choix d'une heuristique judicieuse est lié à la nature du domaine, ce qui va contre le concept d'indépendance entre base de connaissances et moteur : un compromis est donc à trouver. Il s'agit donc de diminuer le rapport coût/efficacité, en évitant d'aller trop loin sur un chemin infructueux et décider à temps quand faire un retour en arrière pour explorer une nouvelle voie. Ceci est un vaste problème.

Beaucoup de travaux concernent également des outils qui permettraient à l'utilisateur d'agir de façon interactive sur le contrôle, de façon à accélérer la recherche d'une solution ou orienter cette recherche dans un sens plutôt que dans un autre. Ceci a également pour but de résoudre les problèmes cités juste précédemment (/FAH 79/, /GAL 79/, /FAR 80/, /DIN 83/...).

Ces problèmes sont d'ailleurs liés à celui de l'apprentissage ; des travaux concernent la faculté du système à mémoriser, voire généraliser les problèmes déjà résolus (y compris les cas d'échecs), de façon à accélérer la résolution de problèmes semblables par la suite (/CAR 82/, /KOL 82/, /UTG 82/, /KOD 85/).

II.1.5 - Module d'interface

Peut être encore plus particulièrement pour les systèmes experts que pour les autres types de logiciels, l'interface entre le système et ses divers utilisateurs doit être particulièrement soigné ; c'est une condition sine-qua-non de succès ou d'échec de ce type de logiciel. Nous rappelons brièvement les fonctionnalités de l'interface, et les différents niveaux de sophistication qu'il peut atteindre.

II.1.5.1 - Fonctionnalités

D'une part, l'interface permet aux usagers (au sens défini en I.2.1) de communiquer avec le système. Les usagers peuvent définir un problème, influencer éventuellement la stratégie de recherche, interroger le système sur ses connaissances.

Le système acquiert les données du problème, donne la solution du problème aux usagers, et surtout peut justifier ses résultats (ie répondre à la question comment ?) ou expliquer une question qu'il pose en cours de recherche de solution (ie pourquoi le système demande-t-il telle information ? que cherche-t-il à montrer ainsi ?). On peut aussi envisager qu'il sache répondre à la question de l'usager : pourquoi pas telle solution à mon problème ?. Cette fonction d'explications est primordiale et doit être très flexible dans un système expert.

(/POL 82/).

D'autre part, l'interface permet au système d'être mis à jour au fur et à mesure du temps et des besoins. Cette mise à jour, correspondant au départ à l'acquisition des connaissances, peut être faite directement par les experts, ou bien par l'intermédaire d'un gestionnaire du système. Or, si le gestionnaire est en général informaticien, les experts n'ont aucune raison de l'être. Donc dans le cas de contact direct avec les experts, l'interface doit être beaucoup plus sophistiqué.

II.1.5.2 - Différents niveaux d'interfaces possibles

Pour l'usager, différents types d'interface sont envisageables, selon les contraintes d'utilisation que l'on décide de lui imposer. On vise un idéal d'interface très souple, ne demandant pas d'effort d'apprentissage de la part de l'usager, communiquant de façon la plus naturelle possible avec lui, en fonction du domaine d'application. A cette fin, on peut utiliser un langage sophistiqué (semi-naturel), des techniques graphiques, et toutes les possibilités offertes par la technologie.

Pour l'autre type d'utilisateurs, deux optiques sont possibles. Pour éviter d'introduire des erreurs lors du transfert des connaissances des experts dans la base de connaissances, l'interface peut être développé de façon à ce que les experts puissent exprimer directement leurs connaissances dans le système ; cela nécessite l'élaboration d'un langage de haut niveau, possédant le vocabulaire du domaine d'application (/FRI 81/), donc lié à celui-ci. En réalité, les experts, a priori non informaticiens, sont amenés, par cette méthode, à se familiariser avec l'informatique ; d'autre part, un tel interface doit être doté d'une capacité de vérification de la cohérence de la base de connaissances obtenues, surtout si plusieurs experts y participent : ce problème est loin d'être résolu. L'autre méthode consiste à ce qu'un gestionnaire soit chargé de mettre les connaissances des experts sous une forme admissible du système ; dans ce cas, l'interface peut être de moins haut niveau, le gestionnaire étant un informaticien, et ayant en fait le rôle d'ingénieur cogniticien (/BON 84-b/, /MAN 84/). Notons en

passant un autre intérêt des systèmes experts : celui d'obliger les experts à formuler correctement leurs connaissances, et en faire le point ; mais il n'existe pas de méthode formelle pour les y aider.

Il n'y a donc pas une ou des techniques de construction d'interface de systèmes experts : chaque application voit son interface particulier, selon la philosophie et les objectifs du système.

II.1.6 - Aperçu de l'existant

Nous mentionnerons d'abord les principaux langages utilisés lors de la réalisation de logiciels type système expert ; nous verrons quelques exemples de systèmes utilisés. Enfin nous conclurons par les limites techniques observées dans ce domaine.

II.1.6.1 - Langages de programmation

A priori tous les langages sont possibles, mais certains seront plus efficaces. En réalité, les préférences tournent autour de LISP et PROLOG ; on utilise aussi des langages orientés objet dans certains cas (voir II.1.3.2.3 et /BON 85/).

Les points communs de LISP et PROLOG tournent autour de la récursivité, du traitement de données symboliques, d'une facilité d'utilisation. LISP est un langage classique, disposant maintenant d'un environnement très riche (possibilité de calcul numérique, compilateur, etc...). PROLOG offre des structures de contrôle plus riches ; il est très déclaratif et permet d'exprimer aisément des règles en logique des prédicats (clauses de Horn), puis de faire des démonstrations les mettant en jeu (/LAU 82-2/, /GEN 85/) ; en fait son interpréteur n'est généralement pas suffisant pour servir directement de moteur à un système, mais il peut alléger considérablement le travail du programmeur du système. Par ailleurs, on commence maintenant à disposer d'environnements de travail assez riches.

II.1.6.2 - Micro-revue de logiciels utilisés de type système expert.

En réalité, très peu de systèmes ont dépassé le stade du prototype pour être utilisés industriellement. Nous mentionnerons ici principalement des logiciels ayant dépassé ce cap.

Les domaines d'application explorés ont été ceux où les connaissances s'expriment bien de façon déclarative : médecine, chimie, biologie, géologie, informatique, mathématique, langage naturel.

- Médecine : MYCIN et ses dérivés (/BUC 84/)
  - MYCIN : diagnostic des maladies bactériennes du sang
    règles de production - chaînage arrière
  - TEIRESIAS : explication du raisonnement de MYCIN, et acquisition des connaissances-de manière intelligente

- Recherche de pétrole : LITHO (industrie - Schlumberger)
  règles de production
  moteur d'inférences de MYCIN - EMYCIN

- chimie : DENDRAL (industrie)
  règles de production (un des premiers SE)
  META-DENDRAL
  apprentissage - généralisation des règles de DENDRAL

- géologie minérale : PROSPECTOR (industrie)
  réseaux sémantiques et règles de production

- mathématiques : MACSYMA (calcul algébrique)
  utilisé par de nombreuses personnes

Cette petite liste est loin d'être exhaustive....

II.1.6.3 - Limites actuelles des systèmes experts

Les systèmes experts ne sont pas la panacée universelle, et ne résoudront des problèmes que si on les a dotés de la connaissance

nécessaire pour le faire.

D'autre part, certains points techniques sont loins d'être réglés.

En voici quelques-uns pris parmi les plus importants :

- compromis entre l'indépendance moteur/base de connaissances, et l'efficacité du système.

- choix d'un formalisme pour représenter les connaissances qui soit adéquat avec le monde réel à décrire.

- acquisition et transfert de l'expertise (aucune méthode reconnue).

- représentation de la notion d'incertitude de façon satisfaisante.

- manipulation et utilisation optimale de la méta-connaissance dans une application donnée.

- génération satisfaisante des explications.

- dégradation brutale des performances du système aux limites du domaine.

- interfaces encore de faible qualité.

- interaction de l'usager sur les stratégies de contrôle du système.

etc...

II.2 - LITTERATURE CONCERNANT NOTRE DOMAINE D'APPLICATION

Nous étudierons d'abord les caractéristiques et les difficultés relatives aux bibliothèques scientifiques et à leur développement. Ensuite, nous situerons notre projet par rapport aux

divers types de logiciels existants d'aide à la résolution de problèmes mathématiques.

II.2.1 - Caractéristiques des bibliothèques scientifiques

II.2.1.1 - Intérêt de telles bibliothèques

Le besoin de disposer de bibliothèques de modules scientifiques s'est rapidement fait sentir dès les années 60, lorsque le calcul scientifique sur ordinateurs a gagné une certaine ampleur ; cela éviterait enfin à chaque programmeur de consacrer un gros effort à concevoir individuellement un logiciel pour résoudre un problème, intéressant en réalité une large communauté de ses collègues. De telles bibliothèques permettraient de mettre les efforts en commun, pour produire des outils de meilleure qualité, écrits une fois pour toutes par des spécialistes de la programmation et du calcul numérique.

Toutefois, la conception de ces bibliothèques n'a pas été sans occasionner des difficultés, dues surtout à celles posées par le calcul numérique sur ordinateur (/BRU 79/).

II.2.1.2 - Difficultés inhérentes à ces bibliothèques

Une des difficultés est celle de la fiabilité des logiciels numériques. Elle a deux origines : des limites dues à la théorie, d'autres dues aux calculateurs (/MOR 79/). En effet, certains problèmes numériques ne sont pas résolus du point de vue théorique, bien que ce domaine soit en évolution permanente ; par exemple, on ne savait pas calculer correctement, à la main, les valeurs propres correspondant à des vecteurs propres très voisins d'une matrice donnée (en 1979) ; l'usage du calculateur n'est d'aucun secours à ce type de difficulté. D'autre part, l'utilisation des calculateurs peut introduire des erreurs de précision machine qui, selon les données, peuvent être à l'origine de résultats catastrophiques (/ARO 78/). Enfin, bien sûr, certains problèmes sont résolus de manière désastreuse lorsqu'ils combinent des difficultés théoriques à celles introduites par la machine (ex : instabilité numérique obtenue avec l'algorithme du Simplexe).

Les difficultés pré-citées sont liées à "l'outil de résolution". Mais il existe aussi le facteur "qualité numérique" du problème à traiter ; nous pensons-là à la matrice 3x3 mal conditionnée, à une grande disproportion de magnitude entre diverses données, etc... En réalité, c'est la combinaison des deux facteurs, l'un fonction du logiciel (ou de la méthode mise en oeuvre), l'autre fonction des données numériques, qui peut conduire à des résultats catastrophiques numériquement, et pas toujours prévisibles.

Un autre souci est celui de la portabilité d'un logiciel scientifique. A cause des différences entre les calculateurs sur des éléments fondamentaux comme : la représentation d'un nombre, la base arithmétique utilisée, ou le traitement de la virgule flottante, il est difficile de produire un logiciel qui s'adapterait sur des machines différentes, en gardant des performances analogues. Les caractéristiques et performances d'un logiciel (place mémoire occupée, nombre d'opérations de base, précision atteinte) dépendent de celles du calculateur d'exécution. Cette difficulté à créer des logiciels portables à moindre coût est liée également au manque de normalisation des langages couramment utilisés pour faire du calcul scientifique (malgré la tentative concrétisée par FORTRAN 77).

Pour que la qualité des logiciels composant une bibliothèque soit toutefois optimale, l'équipe qui conçoit ces logiciels doit travailler de façon très rigoureuse pour combattre les fâcheuses tendances naturelles des programmeurs. Dans ce but, la programmation des modules doit respecter certaines normes de façon à ce que les produits obtenus soient assez facilement maintenables, relativement portables et déchiffrables par une personne autre que leurs concepteurs ; toutefois, il faut trouver un compromis pour ne pas imposer des normes de programmation trop contraignantes qui nuiraient à la qualité du logiciel produit. D'autre part, l'architecture et le mode d'utilisation des logiciels doivent être faciles à comprendre ; les modules doivent donc être parfaitement documentés, de manière claire et si possible uniforme (ceci étant lié à la complexité et au manque de standardisation des langages de programmation utilisés) ; dans le même objectif, devraient figurer dans les notices d'utilisation : les conditions et les limites d'application, les cas d'erreurs, les limites à la portabilité sur

différentes machines, les performances selon la machine d'exécution, etc... Enfin, il doit y avoir une surveillance très rigoureuse de la part des concepteurs des bibliothèques sur la qualité numérique de leurs produits ; cela implique une longue étape de tests pour chaque module, mais aussi la gestion de mises à jour fréquentes et communiquées aux utilisateurs de ces bibliothèques.

Le respect de ces contraintes exige une façon de travailler très organisée et rigoureuse. De telles méthodes de travail ont permis à certaines bibliothèques, comme NAG ou IMSL, d'avoir une certaine réputation, malgré des difficultés encore mal résolues (/FOR 79/).

En conclusion, nous pouvons remarquer la complexité des informations contenues dans ces bibliothèques, principalement due à celles du domaine du calcul scientifique - ie un domaine en constante évolution, et dans lequel on ne peut jamais affirmer à l'avance de façon sûre que les résultats obtenus seront satisfaisants.

II.2.2 - Systèmes de résolution de problèmes mathématiques

Tous les travaux que nous examinerons ont en commun d'opérer à partir d'une base d'outils de résolution de problèmes. Nous verrons comment ces outils sont utilisés, et différentes approches abordées.

II.2.2.1 - Résolution numérique autour d'une bibliothèque

II.2.2.1.1 - Un outil général : PROTRAN

PROTRAN est un logiciel développé en annexe à la bibliothèque IMSL (/RIC 83/). L'utilisateur de PROTRAN définit un problème par ses données numériques et la qualité des résultats espérés. PROTRAN se charge alors de trouver un module de IMSL adapté à ce problème (l'utilisateur suppose que c'est le meilleur!) et de l'exécuter sur les données numériques de l'utilisateur. La réponse fournie par PROTRAN est donc un résultat numérique du problème posé. Précisons que l'utilisateur peut éventuellement spécifier quel type de méthode il veut voir utiliser, via le module de résolution numérique. Toutefois, PROTRAN n'est pas un outil interactif.

C'est un outil intéressant pour l'utilisateur qui veut considérer IMSL comme une boite noire. Mais, sauf exception, il ne permet pas de savoir comment le résultat a été obtenu (critères de choix du module, méthode ou algorithme mis en oeuvre), ni si le résultat obtenu est numériquement correct. De plus, il ne permet de résoudre que des problèmes pas à pas ; la description des données numériques est assez laborieuse s'il s'agit de problèmes de dimensions importantes. Enfin, PROTRAN présente à la fois les avantages et les inconvénients de ne disposer que des modules de la (bonne) bibliothèque IMSL. (/PRO 83/).

II.2.2.1.2 - Un outil spécialisé dans le calcul d'éléments finis : FLUX-EXPERT

FLUX-EXPERT est un logiciel mettant en oeuvre des techniques des systèmes experts permettant de résoudre une équation aux dérivées partielles par la méthode des éléments finis. Le problème à résoudre se décompose en effet aisément en sous-problèmes ; une fois faite cette décomposition, FLUX-EXPERT utilise des algorithmes, adaptés aux caractéristiques des données, pour résoudre numériquement des problèmes élémentaires, ou "opérateurs" (résolution de systèmes linéaires, calcul d'intégrants, calcul de polynômes, etc...) apparaissant dans le graphe de résolution obtenu. (/MAS 84/).

L'approche "système expert" s'explique par le fait que FLUX-EXPERT veut résoudre n'importe quelle équation aux dérivées partielles, pour éviter l'effort classique et laborieux consistant à écrire autant de logiciels que de types différents d'équations. Aussi FLUX-EXPERT possède une base de connaissances ; cette base contient des règles permettant de choisir, pour un opérateur donné, tel algorithme en fonction de telles propriétés sur les données. Il est donc facile d'enrichir cette base de façon à pouvoir traiter un nouveau type d'équation, à moindre coût. FLUX-EXPERT a d'ailleurs été utilisé dans des domaines physiques assez variés.

Cependant, le moteur d'inférences n'est pas indépendant du type général des équations solubles, les grandes lignes du schéma de décomposition du problème initial étant figées. D'autre part, il n'y a

pas beaucoup de souplesse au niveau du choix des algorithmes (ou modules FORTRAN). En effet, pour un opérateur donné, tous les modules doivent, par construction, avoir le même interface (donc ils doivent être écrits spécialement pour FLUX-EXPERT) ; d'autre part, le choix d'un module de résolution d'un problème est tout à fait déterministe et simplifié : on ne peut pas exprimer de contrainte de performances, et il n'y a en général qu'un module possible pour une propriété précise sur les données ... Enfin, les possibilités du système à s'auto-expliquer semblent assez réduites.

Toutefois, cette approche est intéressante et a le mérite de répondre de façon relativement efficace à des besoins concrets. Néanmoins, elle gagnerait à être étendue sur de nombreux points.

II.2.2.2 - Aide à la résolution en calcul algébrique

Nous allons maintenant étudier rapidement les différentes approches utilisées pour résoudre le problème de la résolution algébrique, pour lequel des techniques d'intelligence artificielle sont envisageables. Malgré la dissimilitude entre ce problème et le nôtre, les différents courants rencontrés dans ce domaine sont très intéressants, car ils nous permettront de mieux situer notre philosophie de travail.

II.2.2.2.1 - L'approche calcul algébrique

De nombreux systèmes de calcul algébrique - ou symbolique, ou formel, ont été développés (/VIV 80/). Un des plus performants actuellement est sans doute MACSYMA, disponible en de nombreux sites et assez utilisé. MACSYMA peut simplifier des formules mathématiques, résoudre des équations, dériver ou intégrer des fonctions ... Certaines de ses commandes font appel à des heuristiques, il utilise aussi le mécanisme de filtrage pour décider de la poursuite des calculs. MACSYMA est un système très puissant au niveau calcul, et présente un interface assez évolué, permettant à l'utilisateur de définir ses propres commandes ou fonctions (/HAND/).

Toutefois, aussi évoluée soit-elle, cette approche n'est envisageable que si le système est interactif. En effet, c'est quand même l'utilisateur qui doit piloter le système, en lui indiquant les différentes étapes à réaliser au cours du calcul ; le système lui, ne sait que comment réaliser ces étapes intermédiaires, n'a pas l'intelligence globale du calcul dans son ensemble, et ne peut pas s'auto-expliquer.

II.2.2.2.2 - L'approche raisonnement artificiel

Cette approche n'a pas été développée initialement pour les mathématiques, mais plutôt pour des domaines plus concrets où l'on a besoin de faire des déductions à partir de situations données pour pouvoir aboutir à des décisions concrètes (en robotique par exemple). Tous les travaux de démonstration automatique de théorèmes rentrent dans ce cadre ; cette approche constitue un des axes de l'intelligence artificielle. Combinée à l'exploitation d'heuristiques, c'est elle qui est à l'origine du système AM, capable de découvrir de nouveaux concepts intéressants en mathématiques, à partir d'un ensemble fini de concepts de base (/LEN 77/). Les systèmes capables de réduire des problèmes en sous-problèmes plus simples (au sens de certains critères à définir) rentrent donc aussi dans ce cadre.

C'est donc bien une approche tout à fait complémentaire de la précédente, ne mettant plus en jeu des connaissances de type opératoire, mais de type stratégique : ces connaissances indiquent à quel moment il faut utiliser telle connaissance opératoire.

II.2.2.2.3 - Mixage des deux approches précédentes

Pour pouvoir réellement aider complètement l'utilisateur à résoudre un problème mathématique, l'idéal est donc d'allier les techniques de calcul algébrique (pouvant d'ailleurs aller éventuellement jusqu'au calcul numérique) à celles du raisonnement artificiel.

Le système CAMELIA concrétise bien cette idée (/VIV 84/). Ce logiciel est en fait la superposition d'un module "intelligent" de type système expert, au module de calcul algébrique REDUCE. Les connaissances du module intelligent s'expriment sous forme de "plans de résolution de problème" (savoir-faire de l'expert, connaissances empiriques et à caractère incertain) (/VIV 77/) et de "règles" (connaissances sûres, comme des règles de réécriture d'expressions mathématiques) ; le moteur d'inférences interprète les plans principalement en chaînage arrière, avec possibilité de retour-arrière. Les problèmes élémentaires auxquels aboutissent les plans correspondent à des commandes de REDUCE. CAMELIA a plusieurs modes de fonctionnement, allant du contrôle de chaque décision (ou plan) assisté par l'utilisateur, à une autonomie de résolution totale, auquel cas l'usage de méta-connaissances et d'heuristiques lui facilite la tâche (/VIV 84/).

Une telle approche, mêlant les connaissances de type décisionnel à des connaissances de type opératoire, semble être une voie très prometteuse pour les mathématiques assistées par ordinateur : c'est de loin la plus complète.

II.2.2.3 - Situation de notre projet par rapport à ces approches

La philosophie de notre projet rejoint un peu l'idée de complémentarité entre deux types connaissances exprimée par M.Vivet. Toutefois, le système que nous projetons ne fera pas de la résolution algébrique ou numérique du problème mathématique ; nous ne cherchons qu'à obtenir les meilleurs logiciels susceptibles de résoudre notre problème : les connaissances de type opératoire ne sont pas du même niveau que celles de CAMELIA, les données du problème non plus.

En outre, par rapport aux systèmes décrits en II.2.2.1 fonctionnant autour d'une bibliothèque, l'originalité de notre projet est d'essayer de traiter avec plusieurs bibliothèques, à priori hétérogènes, et constituant déjà en elles-mêmes des objets indépendants. D'autre part, nous désirons sélectionner des outils d'aide à la résolution de problèmes sur des critères qui ne seront pas toujours simples, de façon à mettre en avant la notion de qualité des résultats à obtenir : celle-ci n'est pas facile à exprimer et justifie déjà en partie notre orientation vers des techniques d'intelligence artificielle. Enfin, notre projet est au départ assez ambitieux, puisqu'il ne veut pas se limiter à la résolution d'un seul type de problème.

Signalons enfin un travail intéressant dont l'objectif ressemble au nôtre ; il s'agit de l'application du langage de type objet SHIRKA, pour décrire des liens entre des problèmes et leurs données, et des algorithmes de résolution(/REC 84/). Ce système offre à l'utilisateur deux possibilités : connaître les caractéristiques d'application d'un algorithme donné, et connaître l'algorithme approprié à un problème donné. Les techniques utilisées permettent une bonne représentation des concepts manipulés. Toutefois, certaines difficultés ne sont pas bien résolues : le contrôle du processus d'identification (i.e. éviter de perdre du temps à examiner trop bas dans la hiérarchie des structures dans le cas où il n'existe pas d'algorithme adapté à la résolution d'un problème donné), et la génération d'explications claires des raisonnements du système. Cette approche présente en fait les avantages et inconvénients discutés en II.1.3.2.3. Une différence avec notre projet est aussi que nous ne voulons pas nous limiter à un seul type d'outil de résolution de problème (ici, des algorithmes).

Aussi nous orienterons-nous probablement vers des techniques différentes, étant donné que l'on met l'accent sur les explications des raisonnements générés par le système.

III - CONCEPTION D'UN SYSTEME D'ASSISTANCE A L'UTILISATION DE LOGICIELS SCIENTIFIQUES

Ce chapitre concerne la conception de l'application proprement dite. La première partie contient des spécifications assez détaillées du système qui répondrait aux besoins exprimés en I.3, surtout en ce qui concerne les fonctions destinées à l'usager. Ensuite, nous présentons le travail d'implémentation d'une maquette. Enfin, dans la dernière partie, nous analysons certains résultats obtenus avec cette maquette, pour critiquer quelques choix faits sur les techniques utilisées ; nous en tirons des conclusions quant aux suites possibles de ce travail, en particulier la réalisation d'un système "grandeur nature" et un travail plus approfondi sur la génération des explications dans les systèmes à bases de connaissances.

III.1 - SPECIFICATIONS DETAILLEES DU SYSTEME

Nous rappelons d'abord quelques-unes des idées générales qui ont gouverné cette étude, expliquant entre autres pourquoi nous nous sommes surtout intéressés à l'aspect usager du système, au détriment - dans le cadre de cette thèse uniquement - de l'aspect gestionnaire. Nous présentons donc ensuite les spécifications techniques qui permettront de réaliser les principales fonctions du système pour l'usager : fonction documentaire, fonction résolution et fonction d'interface.

III.1.1 - Avant propos

Ce paragraphe concerne trois remarques préliminaires tournant autour de la fonction résolution de problèmes de l'usager ; nous expliquons pourquoi l'étude de cette fonction, et de celles concernant l'usager, ont été privilégiées ici ; puis nous préciserons d'emblée la philosophie du système dans le cas où il ne connaît pas de solution idéale au problème de l'usager, et ses limites quant au type de problèmes qu'il saura résoudre dans un premier temps.

III.1.1.1 - Justification de privilèges à l'usager, et à la résolution de problèmes

Par rapport aux besoins exprimés en I.3 auxquels doit répondre le système, nous avons choisi de faire porter nos efforts sur les problèmes ayant trait à l'intelligence artificielle ou au génie logiciel. Or pour le gestionnaire, une fois que l'architecture du système a été conçue de façon à assurer facilement l'évolutivité des connaissances, les problèmes à résoudre sont :

- l'élaboration d'un interface lui permettant de mettre à jour les connaissances du système - cette tâche étant par nature très laborieuse.

- la gestion de la cohérence de la base de connaissances.

- la gestion d'historiques d'utilisation du système.

La résolution du premier problème est assez laborieuse ; elle consiste à mettre en place un analyseur grammatical avec un vocabulaire étendu relatif aux mathématiques numériques, puis un interpréteur des fragments exprimant les connaissances en pseudo-français, en items de connaissances codés dans le formalisme de représentation interne ; la complexité de ce travail est donc liée à celle du formalisme.

Le troisième problème est en partie relatif au système d'exploitation sur lequel tournera notre projet, et soulève des problèmes sans lien avec l'intelligence artificielle : mémorisation d'une session usager, restauration d'un contexte de résolution, problèmes de droits d'accès selon les usagers, gestion de l'analyse des sessions enregistrées etc...

Par contre, le problème de la cohérence est, lui, un des axes actuels de recherche concernant les systèmes à base de connaissances, et est loin d'être résolu actuellement. Pour l'instant, nous l'écartons délibérément.

Voilà pourquoi nous laisserons de côté l'aspect gestionnaire du système.

En ce qui concerne l'usager, nous distinguons principalement deux groupes de fonctions : fonction documentaire, de renseignement sur un outil de résolution, et fonction résolution de problèmes. C'est surtout la seconde qui justifie l'appel à des techniques développées en intelligence artificielle, et une programmation de type système expert. C'est pourquoi nous développerons surtout cette seconde fonction, même si la première est d'un grand intérêt également pour l'usager.

Ces choix orienteront la suite de la présentation de cette étude.

III.1.1.2 - Philosophie du système devant les cas d'échec

L'idée est d'éviter que l'usager se retrouve sans aucune proposition du système pour résoudre son problème. Ceci pourrait arriver si le système cherchait systématiquement à fournir une réponse qui aurait toutes les chances de donner de bons résultats, quel que soit le problème posé ; lorsque le produit miracle n'existerait pas (cas des problèmes à mauvaise allure : données mal conditonnées, matrice immense, etc..., ou d'une base de connaissances incomplète), l'usager serait coincé... et choisirait peut-être, au hasard, l'outil le plus mal adapté possible.

Pour éviter cela, on préfère, arbitrairement, éventuellement rebrousser chemin et s'engager sur une voie qui mènera à des outils de résolution moins précis ou moins bons, répondant à des contraintes moins serrées que ne le sont celles de l'usager, pour lui proposer une solution dégradée pouvant au moins fournir quelques résultats. Toutefois, cela n'est envisageable qu'à condition de signaler cette démarche à l'usager, pour qu'il reste sur ses gardes quant aux résultats qu'il obtiendra, et effectue éventuellement des contrôles de qualité.

De façon générale, un expert n'est jamais certain que l'application de tel algorithme à telles données numériques donnera des résultats corrects (cf. II.2.1.2). Le système se doit donc d'indiquer ce

facteur d'incertitude lié à la solution qu'il propose, pour toute résolution de problème. Ensuite, selon le degré de ce facteur d'incertitude, il peut conseiller - ou l'usager devrait décider - d'appliquer des vérifications de qualité numérique. En fait, en calcul numérique, la difficulté est que l'on ne peut jamais assurer à l'avance qu'un outil de résolution aboutira à un succès, l'échec est toujours un risque présent. C'est ce que doit bien faire apparaître le système.

III.1.1.3 - Limites du système en matière de résolution

Les problèmes que notre système devra savoir résoudre sont pour l'instant "élémentaires". Nous entendons par là soit un seul problème simple (ex : trouver le polynôme d'interpolation d'une fonction sur un domaine, résoudre un système linéaire ...), soit un ensemble de problèmes simples portant sur des données communes, et relevant d'une même technique de résolution ; par exemple, le système cherchera un ou des algorithme(s) capable(s) à la fois de résoudre le système linéaire AX=B, et de calculer le déterminant de la matrice A.

Sont pour l'instant exclus les problèmes plus complexes pouvant se décomposer en un enchaînement de problèmes élémentaires. Ceci constituera une étape à placer au-dessus de notre système.

D'autre part, le système connaît les classes de problèmes qui sont de son ressort, ainsi que les problèmes de ces classes qu'il sait résoudre ; il connaît aussi les problèmes de ces classes qu'il ne sait pas résoudre directement, mais qui peuvent se ramener à des problèmes qu'il sait résoudre. Si l'usager lui pose un problème n'appartenant pas à ces catégories, c'est bien sûr un cas simple d'échec irrémédiable ...

III.1.2 - Fonction documentaire

Nous rappelons les différents domaines d'application de la fonction documentaire du système ; puis nous présentons les concepts de la base de connaissances que cette fonction doit manipuler, ainsi que la représentation externe qui en a été choisie.

III.1.2.1 - Objectifs

Cette fonction couvre un bon nombre de petites tâches très diverses. Elle comprend :

- l'explication de vocabulaire mathématique (ex : qu'est-ce qu'une matrice symétrique) quand l'usager en a besoin.

- la génération de renseignements sur des outils de résolution de problèmes. Cela peut être des renseignements théoriques (ex : principe de fonctionnement de la méthode de Lagrange) ou pratiques (ex : comment utiliser un sous-programme donné, des références bibliographiques).

- la génération de renseignements sur des outils de contrôle de qualité numérique ; ici encore, ces renseignements peuvent être théoriques ou techniques et pratiques. Il y a deux catégories d'outils de contrôle : les premiers sont "préventifs" (ex : méthode de préconditionnement d'une matrice), les seconds s'appliquent après l'obtention de résultats numériques, pour contrôler ces derniers (ex : méthode consistant à faire varier numériquement les données d'entrée d'une quantité ε, pour observer la stabilité numérique).

III.1.2.2 - Présentation des principaux concepts de la base de connaissances

Comme cela apparaît dans le paragraphe précédent, il y a plusieurs catégories de concepts (cf. figure 4).

Nous accorderons beaucoup d'importance aux outils de résolution, car ils font l'objet de la fonction résolution, et sont donc des éléments piliers de la base de connaissances. Par contre, les outils de contrôle ou les définitions de vocabulaire sont très importants pour l'usager, mais ne justifient pas vraiment le recours à des techniques de système expert.

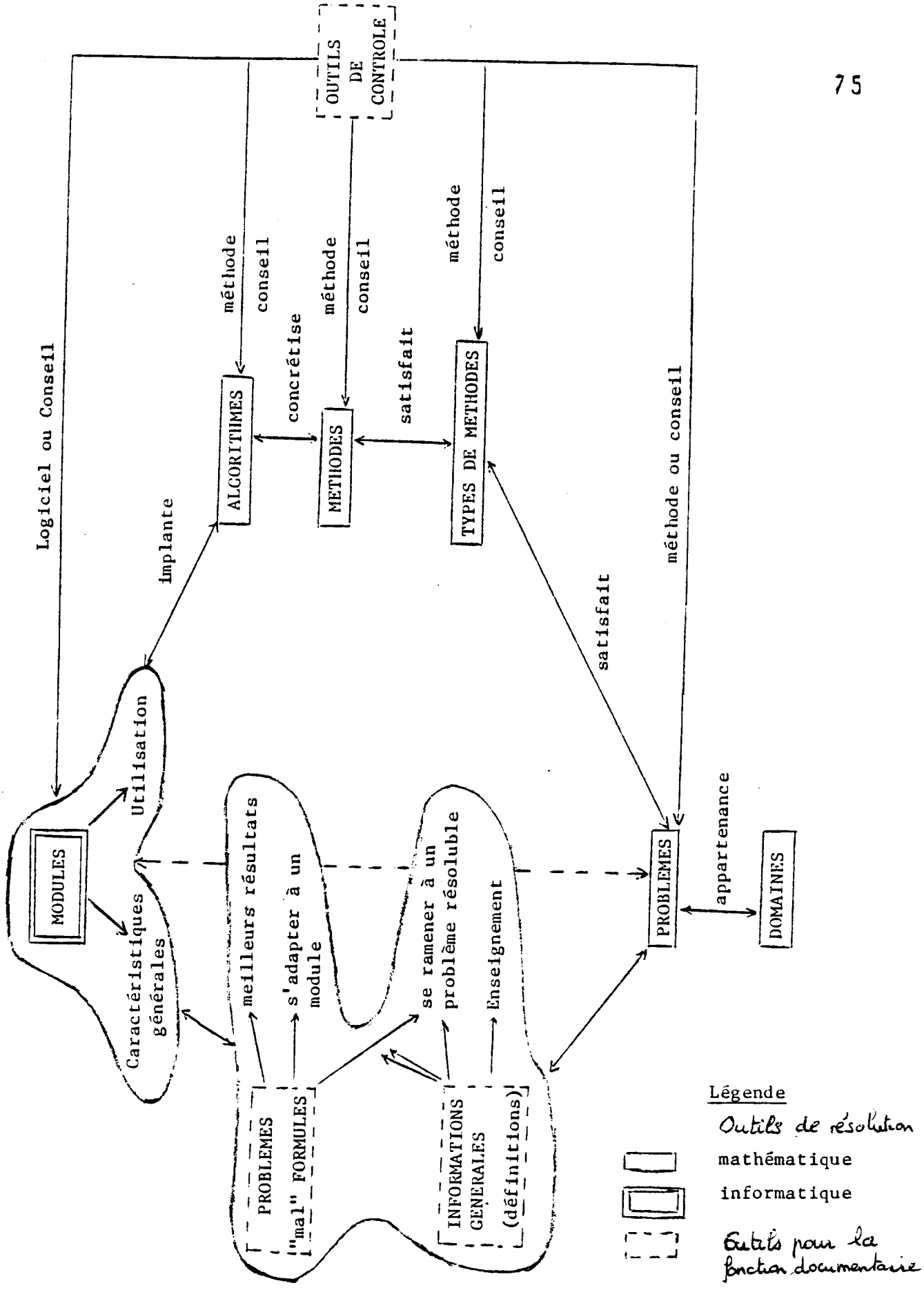
Les outils de résolution sont de plusieurs types, allant du plus abstrait au plus concret. Les concepts de base sont : le domaine mathématique ou encore classe de problèmes (ex : algèbre linéaire), puis problème (ex : inversion de matrice) ; en fait, il y a deux types de

problèmes : ceux que le système sait résoudre, et ceux que l'usager peut ramener à des problèmes du premier type - ce sont surtout les premiers qui nous intéressent. L'outil de résolution le plus abstrait est le type de méthodes (ex : itérative). A un type de méthodes correspond un ensemble de méthodes (ex : méthode du Gradient Conjugué ...) ; une méthode a un niveau d'abstraction encore assez élevé ; c'est un principe général de résolution, qui en indique les grandes étapes. Par contre, dans un algorithme, on a déjà posé très sérieusement la manière de conduire les calculs, selon les directives établies dans la méthode dont il est la mise en oeuvre (ex : LL^T) ; plusieurs algorithmes peuvent donc être issus d'une même méthode, variant sur l'organisation des calculs à mener. Enfin, un module (ou sous-programme) implémente un algorithme ; cette fois la référence à un calculateur intervient ; d'ailleurs, plusieurs modules peuvent implémenter le même algorithme : ils diffèrent en leurs limites numériques d'application, leurs structures de données, leurs performances, etc... Le module est le seul concept où intervient la notion de langage de programmation, contrairement aux concepts précédents où suffit le langage mathématique.

Notons que la terminologie adoptée n'est pas vraiment universelle ; par exemple certains désignent notre concept méthode par le terme algorithme, il n'y a pas de norme dans ce domaine.

Parmi les autres concepts représentés sur la figure 4, apparaît celui "d'outils de contrôle" ; ceux-ci peuvent être conseillés au cours d'une résolution de problèmes lorsque les caractéristiques des données laissent présager des résultats de mauvaise qualité (cf. I.3.2.1.1). Cependant, ils ne peuvent être proposés seuls comme outils de résolution de problèmes, et ne constituent que des annexes aux outils présentés auparavant (ceci étant indiqué sur la figure 4 par des flèches horizontales, associant un ou des types d'outils de contrôle à chaque type d'outil de résolution.

Les flèches reliant les concepts servant à résoudre un problème (de domaine à modules) définissent l'arbre de résolution de ce problème ; cet arbre a un domaine comme racine, et son niveau terminal est défini par l'usager, selon qu'il désire connaître un type de méthode, une méthode,... ou un module pour résoudre son problème.



Les objets de la base de connaissances

Nous emploierons par la suite le terme d'objet de la base de connaissances pour désigner les concepts, et de règle (de la BdC) pour désigner les liens sémantiques entre concepts représentés par des flèches sur la figure 4.

III.1.2.3 - Représentation externe des objets de la base de connaissances

De façon externe - c'est-à-dire déchiffrable par l'homme, nous choisissons de représenter les objets par des structures. Plus précisément nous aurons un modèle de structure par type d'objets, et les informations contenues dans les champs de ces structures seront spécifiques à l'objet décrit. Notons d'ores et déjà que l'usage de ces structures pour représenter un objet n'a rien à voir avec le formalisme des structures-types de Minsky (cf. II.1.3.2.3), puisque nous ne parlerons pas des mécanismes d'héritage entre structures propres à ce dernier ; c'est d'ailleurs par opposition à l'expression "mode de représentation interne des connaissances" que nos structures constituent une représentation "externe" : elles permettent à l'homme une manipulation aisée, sur papier, des concepts de la base de connaissances.

Cette formalisation doit être assez rigoureuse pour que le système puisse interpréter au besoin les informations contenues dans les champs des structures.

La figure 5 montre comment la méthode de Cholesky peut être décrite avec la structure associée au concept méthode.

nom décomposition symétrique (ou Cholesky)
problèmes résolus
pb initial resolsyslin.d(A,B,X,N) caldet.d(A,B,X,N,D)...
pb renommé resolsyslin.d.cho(A,B,X,N) caldet.d.Cho(A,B,X,N,D)...
description de nouvelles variables (* vide ici)
contraintes A symétrique
propriétés (* vide ici)
description-commentaires d'autant meilleure que la diagonale est dominante. Principe : décomposition symétrique et résolution.
domaine Algèbre linéaire
Conseils de contrôle Conschol1 Conschol2
type de méthode directe (* lien vers l'objet "père")
méthode de contrôle (* vide ici)
références WILKINSON Algebraic Eigenvalue Problem - p.375
caractéristiques grande stabilité - bonne précision

Structure représentant la méthode de Cholesky

Figure 5

Dans la suite de ce paragraphe, nous analyserons les principaux champs de ces structures.

III.1.2.3.1 - Principaux champs de structures décrivant les objets mathématiques

Ce paragraphe ne concerne pas les champs spécifiques à la structure décrivant les modules, à cause de la référence à un langage informatique relativement complexe (FORTRAN).

a - *Représentation du lien sémantique entre deux structures*

Il s'agit du lien entre deux objets successifs dans un arbre de résolution, par exemple entre la méthode de Cholesky et le type de méthode directe. Ce lien est représenté dans la structure fille (méthode) par un champ mentionnant la structure mère dont elle concrétise le principe général. Ce lien est fait vers le haut (dans l'arbre) pour faciliter la modification de la base de connaissances. L'idée sous-jacente est que l'on met à jour plus souvent des modules que des types de méthodes ; on pourra donc enlever ou rajouter des structures représentant des modules, sans modifier celles qui représentent les principes qu'ils mettent en oeuvre.

b - *Représentation d'une variable (mathématique)*

Tant que l'on reste dans les objets type de méthode ... algorithme, on parle de "variable mathématique". Une "variable mathématique" (matrice m) peut être représentée par une "variable informatique" (tableau M) dans un sous-programme donné.

Une variable mathématique n'a qu'un rôle, elle est donnée ou (exclusif) résultat. Elle se représente à l'aide de plusieurs sous-champs :

- type mathématique (entier, fonction réelle ...) ; les types peuvent être composés (ensemble-de réels).

- catégorie (soit donnée, soit résultat).

- rôle (fonction à interpoler)

c - *Contraintes et propriétés*

Les contraintes sont vérifiées par les variables données du problème (ex : la matrice à décomposer est symétrique) ; les propriétés sont vérifiées par les variables résultat du problème, après application d'un outil de résolution (ex. une matrice de décomposition est triangulaire).

Dans la structure relative à un objet donné, on ne représente pas, pour alléger la description, les propriétés et contraintes déjà notées dans les structures représentant les antécédants de cet objet dans l'arbre de résolution. Le système peut toutefois les retrouver grâce au champ indiquant l'objet "père" d'une structure (problème d'implémentation).

d - *Relation entre un objet et les fonctions qu'il résoud*

Dans les structures associées aux objets mathématiques, le champ "problèmes résolus" permet d'identifier les problèmes et leurs modes de résolution. Cela permet par exemple au système de faire une

distinction entre une interpolation par Lagrange ou par Hermite. Une différence importante est en effet que, ces deux méthodes étant de type polynômial, celle de Lagrange n'a besoin en donnée que de la connaissance du graphe de la fonction à interpoler en un ensemble de points ; par contre, celle de l'Hermite a besoin en plus du graphe de la dérivée première de cette fonction ; ceci doit être formalisé.

Pour tenir compte de ce type de particularité, le nom de la fonction réalisée par un outil donné symbolise la partie supérieure de son arbre de résolution ; le nom du problème à résoudre évoluera donc au cours de la résolution de façon à tenir compte de nouvelles variables données, etc...

Par exemple, la figure 6 montre comment sont représentées les méthodes de Lagrange et d'Hermite. Ces structures ne sont pas complète ; on aurait un champ commun "type de méthode : polynômiale".

Méthode de Lagrange

problèmes résolus :
pb initial : int.poly(F,X,B,G)
pb renommé : int.poly.L(F,X,G)

Description de nouvelles variables

(* vide ici)

Contraintes
B = BL(base des polynômes de Lagrange)

Méthode d'Hermite

problèmes résolus :
pb initial : int.poly(F,X,B,G)
pb renommé : int.poly.H(F,X,B,G,F1)

Description de nouvelles variables
nom F1
type fonction
catégorie donnée
rôle dérivée première

Contraintes
B = BH(base de polynômes d'Hermite)
F1 = dériv1(F)

description-commentaire :
on utilise la dérivée première de la fonction à interpoler

Remarque : la structure du code symbolisant les problèmes traités reflète l'arbre de résolution (int.poly.H = int(erpolation) de type poly (nomial), par méthode H(ermite)).

Description partielle des méthodes d'interpolation polynômiale de LAGRANGE et de l'HERMITE

Figure 6

e - *Importance relative des informations des différents champs*

Toutes les informations contenues dans une structure n'ont pas la même importance, pour l'usager ou pour le système. La figure 6 montre un champ "description-commentaire" qui renferme du texte très clair pour l'usager, mais qui ne sera pas interprétable par le système. Par contre, il est important que le champ "problèmes résolus" soit correctement rempli, car il pourra servir lors de la résolution de problèmes par le système.

f - *Exemples*

Domaine (ou classe de problèmes)

nom Algèbre Linéaire
commentaire concerne :

- résolution de systèmes linéaires, inversion ou pseudo-inversion de matrices
- calcul des éléments propres d'une matrice carrée, ou des valeurs singulières d'une matrice quelconque
- calcul de certaines quantités liées aux matrices (norme, rayon spectral, trace, déterminant, rang, conditionnement, largeur de bande ...) et recherche de certaines propriétés (matrice symétrique définie positive, symétrique, etc ...)
- calculs matriciels : utilitaires
  opérations matricielles (somme, produit, transposition, mise à l'échelle, normalisation, etc...)
- structuration de matrices creuses, minimisation de largeur de bande d'une matrice.

Remarques :

. Le champ commentaire n'est significatif que pour l'usager : c'est du français.

. Nous distinguons deux types de problèmes associés à un domaine :

- "utilitaires" ex : addition de matrices

A un utilitaire ne correspond pas de méthode et d'algorithme spéciaux.

Problème résoluble

nom résolution de systèmes linéaires carrés (A,B,X,N)
description de variables :

nom	A	B	X	N
type	matrice (N,N)	vecteur (N)	vecteur (N)	entier
catégorie	donnée	donnée	résultat	donnée
rôle	matrice système	2d membre	solution	ordre matrice

relations entre variables A * X = B
description - commentaires
Connaissant A et B, on cherche X tel que AX = B. Le problème de résolution de systèmes linéaires est en relation directe avec celui d'inversion de matrice : en effet, calculer l'inverse de A équivaut à résoudre N systèmes linéaires de matrice A. Le système possède pour tout B une solution et une seule si et seulement si A est inversible ; s'il n'en est pas ainsi, pour certains B l'équation ne possèdera pas de solution, alors que pour d'autres elle en aura une infinité ; néanmoins, dans ce cas les méthodes numériques seront en général impuissantes à mettre en évidence de telles solutions. C'est malheureusement le cas pour des matrices numériquement singulières (et pourtant inversibles).
domaine Algèbre linéaire (* lien vers objet père)
Cas-tests TRSL1, TRSL2
Méthode de contrôle Calrésidu Amelio (* noms des méthodes)
conseil de contrôle Conditmat (* nom du conseil de contrôle de qualité)
propriétés opérateurs (* champ vide)
contraintes det(A) = 0

Type de méthode

nom directe (* résolution de système linéaire)
problèmes résolus
problème initial resolsyslincar (A,B,X,N) invmat (A,B,N) ..
problème renommé resolsyslincar.d (A,B,X,X) invmat.d (A,B,N) ...
description de nouvelles variables (* champ relatif aux variables apparus lors du renommage d'un problème ; ce champ est vide ici)
contraintes (* associées aux variables - vide ici)
propriétés (* associées aux variables - vide ici)
description - commentaires Peu utilisé pour de "grandes" matrices car possibilité de perte de précision, sauf pour Cholesky. Permet de résoudre simultanément des systèmes qui ne diffèrent que par les seconds membres. On parvient mathématiquement à la solution en un nombre fini d'opérations.
domaine Algèbre linéaire
références CIARLET. Introduction à l'analyse numérique mathématique et à l'optimisation
conseil de contrôle contqual (* nom du conseil)
caractéristiques (* pour l'utilisateur, pas pour le Système)
numériquement instable - donne une mauvaise précision.

Méthode

nom décomposition symétrique (Cholesky)
problèmes résolus
  problème initial resolsyslincar.d(A,B,X,N) ...
  problème renommé resolsyslincar.d.Cho(A,B,X,N) ...
description de nouvelles variables (* vide ici)
contraintes A symétrique
propriétés (* vide ici)
description - commentaires D'autant meilleure que la diagonale est dominante. Principe : décomposition symétrique et résolution
domaine Algèbre linéaire
conseils de contrôle
type de méthode directe (* lien vers objet "père")
méthode de contrôle
références WILKINSON Algebraic Eigenvalue Problem
caractéristiques grande stabilité - bonne précision

Méthode

nom Hermite
problèmes résolus
  problèmes initiaux intpol.polyn (F,X,G,B) ...
  problèmes renommés intpol.polyn.He (F,X,G,F1) ...
description nouvelles variables
  nom F1
  type fonction
  catégorie entrée
  rôle dérivée1 de fonction interpolée
Contraintes
  B = base de polynômes d'Hermite
  F continue à ordre élevé
  F1 = deriv1 (F)
Propriétés
Description - Commentaires adaptée à l'interpolation, de polynômes et de fonctions continuement différentiables jusqu'à des ordres élevés.
domaine interpolation
conseils de contrôle
type de méthode polynômial
méthodes de contrôle
références Numérical Interpolation - M. DUPOND
caractéristiques meilleure en précision que LAGRANGE pour des degrés peu élevés. Plus coûteuse en place mémoire que LAGRANGE si interpolation en un point.

Remarque : On retrouve un des problèmes traités par cette méthode via le champ problèmes résolus : c'est le problème "interpoler une fonction" (donner le polynôme interpolant) codé par "intpol".

Algorithme

nom Gauss pivot direct
problèmes résolus
problème initial résolsyslincar.d.eli (A,B,X,N)
problème renommé résolsyslincar.d.eli-gpd (A,B,X,N)
description de nouvelles variables
Contraintes
propriétés
domaine Algèbre linéaire
méthode élimination directe
conseil de contrôle
méthode de contrôle
description - commentaires C'est la méthode directe la plus simple à programmer. Ne doit pas rencontrer de pivot nul ou trop petit, sinon l'algorithme s'interrompt (même si la matrice est inversible). Triangularise la matrice du système en éliminant les inconnues les unes après les autres. La résolution se fait par "remontée" sur le système triangulaire obtenu ...
références Algebra Handbook
caractéristiques plus rapide que Gauss pivot partiel ou total, mais moins précis.

III.1.2.3.2 - Principaux champs de la structure module

La structure qui permet de décrire les modules est assez complexe car elle doit donner de façon claire, et interprétable par le système, tous les renseignements figurant sur la notice d'utilisation d'un sous-programme. Ces informations doivent être structurées dans le but d'accroître les possiblités du système par la suite, de façon à pouvoir un jour traiter les enchaînements de problèmes, donc de modules.

a - *Equation fonctionnelle d'un module*

Ce champ permet d'expliciter formellement les fonctions réalisées par le module, et les transformations subies par les variables du module au cours de son exécution.

Il se présente sous la forme d'une équation. Son membre gauche contient une liste des variables apparaissant dans le module (variables d'appel, variables globales). Le membre droit indique ce que sont respectivement devenues ces variables en fin d'exécution.

assez complexe.

b - *Description des variables (Informatiques)*

Comme nous l'avons vu ci-dessus, une variable peut être entrée et sortie, son rôle et sa valeur se modifiant en cours d'exécution. C'est pourquoi nous avons été amenés à distinguer deux sortes d'informations pour décrire une variable. En effet, le type informatique de la variable est figé, dès la conception du module, contrairement au contenu de la variable. Nous avons donc introduit un sous-champ pour la description syntaxique de la variable, et un sous-champ pour la description sémantique, comme le montre l'exemple ci-dessous.

ex : Module Opérat-matrice
description des variables d'appel
nom B
syntaxe tableau(N) réel
sémantique si X=1 alors { **catégorie sortie**, **rôle** matrice transposée
si X=2 alors { **catégorie** entrée [av(B,operat-matrice)], **rôle** **second membre-produit**
{ **catégorie** sortie [ap(B,operat-matrice)], **rôle** produit-matrices
⋮

Remarque : nous utilisons dans cet exemple une notation particulière pour désigner la valeur de la variable B en entrée du module (av(nom-variable, nom-module)), et la valeur de B en sortie (ap(nom-variable, nom-module)). Cette notation pourrait être utile si l'on veut traiter plus tard les enchaînements de problèmes.

Pour les variables globales (COMMON en FORTRAN), la description fait intervenir éventuellement un autre sous-champ pour indiquer leur valeur initiale (via un BLOCK-DATA).

c - *Contraintes et propriétés*

Les contraintes portent sur les variables en entrée d'un

module, les propriétés portent sur les variables en sortie. Elles font l'objet de champs consacrés, comportant des expressions entre variables et valeurs numériques (ex : M ⩾ N , P = 0 ou P = 1).

d - *Mode d'appel du module*

Un champ de la structure module indique à l'usager la séquence d'appel à écrire dans son propre logiciel pour pouvoir l'utiliser. Ce champ est donc une suite d'instructions FORTRAN, relatives au calculateur sur lequel le module est destiné s'exécuter ; il contient donc des déclarations (variables globales - COMMON -, sous-programmes paramètres du module - EXTERNAL-,...), puis des initialisations de variables (DATA, BLOCK-DATA ...), puis la séquence d'appel du module (CALL..., ou FUNCTION...), et enfin éventuellement une séquence de fermeture de fichiers (/DRE 78/, /FOR 78/).

e - *Sous-modules appelés par le module*

Nous distinguons deux types de sous-modules : ceux qui sont des modules existant déjà dans une bibliothèque, et ceux qui sont fournis par l'usager. Les premiers sont juste mentionnés dans un champ spécial. Les seconds peuvent être soit désignés comme paramètre d'appel - auquel cas l'usager doit les déclarer comme EXTERNAL dans son logiciel principal, soit appelés de façon interne par le module - auquel cas leur nom est une constante.

Un champ permet de décrire ces sous-modules ; ses sous-champs seront grossièrement les mêmes que les champs permettant de décrire un module, mais contiendront en général moins d'informations.

f - *Points d'entrée d'un module*

C'est une notion liée au langage d'implémentation des modules. L'instruction ENTRY permet d'accéder à un sous-programme (subroutine FORTRAN exclusivement sur CDC) par un point d'entrée "secondaire" (par opposition au point d'entrée "principal" habituel) (cf. /DREY 78/).

Nous choisissons de modéliser le code FORTRAN relatif à un

ENTRY comme s'il s'agissait d'un module, dont le nom serait le nom symbolique du point d'entrée, et dont les paramètres d'appel seraient les arguments du point d'entrée. Cela nous conduit à définir une rubrique supplémentaire dans le champ des caractéristiques d'un module : rubrique "module de définition", qui contiendra dans ce cas le nom du module où est défini le point d'entrée. Cette rubrique sera donc le plus souvent vide.

D'autre part, si un module contient des points d'entrée, il n'en sera pas fait mention dans sa description. En fait, cette description correspondra à celle du module sans les points d'entrée.

```
subroutine S (S1, S2, S3)                    structure S (S1,S2,S3)
 {code FORTRAN pour S}

     entry P (p1, p2)                        structure P (p1,p2)
          {code FORTRAN pour P}              module définition S

     entry Q (q1, q2)                        structure Q (q1,q2)
          {code FORTRAN pour Q}

RETURN
END
```

Code FORTRAN — Modélisation

g -*Lien entre un module et le(s) problème(s) mathématique(s) traité(s) correspondant(s).*

Le but est expliquer au Système (toujours en voie d'une extension du système vers le traitement de problèmes plus complexes) ce que représente réellement chaque variable informatique, par rapport au problème mathématique donné par l'utilisateur. Un champ exprime cela formellement, il fera apparaître deux types de fontions : des fonctions faisant correspondre des variables informatiques à des variables mathématiques, et des fonctions établissant des relations entre variables mathématiques. Il subsiste un problème de notation pour différencier les deux types de variables, qui peut être résolu par l'implémentation.

ex : module d'interpolation par la méthode d'HERMITE

Ses variables d'appel comprennent :

- X tableau réel des abscisses où l'on connait la fonction à interpoler,
- Y tableau réel des ordonnées de la fonction en ces abscisses,
- Y1 tableau réel des ordonnées de la dérivée de la fonction en ces abscisses.

:

On peut exprimer la correspondance par :

```
problème-traité : interpol (f,x) = g
        avec    : repr (X) = x
                  repr (Y) = restriction (f,x)
                  repr (Y1) = restriction (deriv1(f),x)
                              .  .
```

h - *Caractéristiques diverses d'un module*

Divers champs doivent contenir les informations permettant à l'usager d'intégrer le module dans son propre logiciel, relatives au langage FORTRAN utilisé : compilation, édition de liens,... D'autres doivent renfermer les caractéristiques du module concernant leurs performances : nombre d'opérations, place mémoire occupée, précision attendue ... Enfin, des champs contiennent encore des références bibliographiques du module, des jeux de tests, des références de logiciels de contrôle de la qualité numérique des résultats obtenus, etc...

i - *Exemples*

Module (seuls les champs les plus intéressants figurent ici)

nom HRMT2
nature subroutine

appel call HRMT2 (N,M,XO,X,Y,Y1,ANS)

fonction module
<N,M,XO,X,Y,Y1,ANS> ← <N,M,X,Y,Y1,intpts-poly-her-a (N,M,XO,X,Y,Y1)>

description variables d'appel

nom N
syntaxe scalaire type entier
sémantique catentrée **rôle** nbpts

nom M
syntaxe scalaire **type** entier
sémantique catentrée **rôle** nbpts où interpoler

nom XO
syntaxe tableau (M) **type** réel
sémantique catentrée **rôle** pt où interpoler

nom X
syntaxe tableau (N) **type** réel
sémantique catentrée **rôle** abscisse-fonction

nom Y
syntaxe tableau (N) **type** réel
sémantiquecatentrée **rôle** ordonnée-fonction

nom Y1
syntaxe tableau (N) **type** réel
sémantique catentrée **rôle** derivfonction

nom ANS
syntaxe tableau (M) **type** réel
sémantique catsortie **rôle** val-fctn

contraintes N < 10

Lien variables-problèmes

pbtraité interpolerfonctionenpts(f,x,x_0,y)
avec X = abscisses (x)
Y = restriction (f,n)
XO = abscisses (x_0)
ANS = ordonnées (y)
Y1 = restriction (deriv1 (f) , x_0)

Module (seuls les champs les plus intéressants figurent ici)

nom VD03A
nature subroutine
appel common / VD03B / SS, SD, EPS, LP, IERR
external function FN
call VD03A (FN, XD, WX, YD, XY, XO, FX, SSERR, MAXIT)

fonction du module (* équation fonctionnelle)
<FM,XD,WX,YD,WY,XO,FX,AX,SSERR,MAXIT,SS,SD,EPS,LP,IERR ; fichier -
-sortie (écran, imprimante) >
← <FN,XD,WX,YD,WY,xminimisantsom (FN,XD,WX,YD,WY), minisom (FN,XD,WX,YD,WY),
PX,SSERR,MAXIT, valsomme (FN,XD,WX,YD,WY),
deriv2 somme (FN,XD,WX,YD,WY), EPS, LP, ferreur (FN,XD,WX,YD,WY) ;
unitésortiemessage (LP) >

avec ferreur = 0 si succès
= 1 si WTX < 0.0
= 2 si WTY < 0.0
= 3 si PX < 0
= 4 si EPS < 0.0

unitésortiemessage = (écran, imprimante) si LP = 0
= (message, imprimante) si LP = 1
= (écran, message) si LP = 2

description des variables d'appel
nom FN
syntaxe fonction **type** réel
sémantique catégorie entrée **rôle** : fonction à minimiser

nom XD
syntaxe scalaire **type** réel
sémantique catenntrée **rôle** abscisse de référence

nom WX
syntaxe scalaire **type** réel
sémantique catentrée **rôle** poids abscisses

nom YD
syntaxe scalaire **type** réel
sémantique catentrée **rôle** ordonnée de référence

nom WY
syntaxe scalaire **type** réel
sémantique catentrée **rôle** poids ordonnées

nom XO
syntaxe scalaire **type** réel
sémantique
catentrée (av XO VD03A) **rôle**vapprox(xmini)
catsortie (ap XO VD03A) **rôle**vexact(xmini)

nom FX
syntaxe scalaire **type** réel
sémantique catsortie **rôle** fonction au minimum

nom PX
syntaxe scalaire **type** réel
sémantique catentrée **rôle** précision demandée

nom SSERR
syntaxe scalaire **type** réel
sémantique catentrée **rôle** indicerreurminisomme
commentaire SSERR=0.0 si la précision machine est atteinte

nom MAXIT
syntaxe scalaire **type** entier
sémantique catentrée **rôle** nbmaxid'itérations

Description des variables globales

nom SS
syntaxe scalaire **type** réel
sémantique catsortie **rôle** minsomme

nom SD
syntaxe scalaire **type** réel
sémantique catsortie **rôle** deviv2minisomme

nom EPS
syntaxe scalaire **type** réel
sémantique catentrée **rôle** +petitnbrereprésentable

val. initiale si machine = IBM 370 alors 10^{-7}
commentaire valeur fonction de la machine

nom LP
syntaxe scalaire **type** entier
sémantique catentrée **rôle** n°ulsortie **val initiale** 6

nom IERR
syntaxe type entier
sémantique catsortie **rôle**indicerreur **val.initiale** 0
commentaire IERR non réinitialisé lors d'appels consécutifs

Description module usager

nom FN
nature subroutine
commentaire le nom est au choix de l'usager
fonction-module
<X FX FD SD> <— <X fonction(X) deriv1(X) deriv2(X) >
commentaire les autres paramètres de la fonction doivent être fournis en COMMON.

Description des variables d'appel
nom X
syntaxe hérité
sémantique catentrée hérité

nom FX
syntaxe scalaire **type** réel
sémantique catsortie **rôle** valeur fonction(x)

nom FD
syntaxe scalaire **type** réel
sémantique catsortie **rôle** valeurdérivéel(x)

nom SD
syntaxe scalaire **type** réel
sémantique catsortie **rôle** valeurdérivée2(x)

Lien variables-problèmes ...

Contraintes PX > 0.0 EPS > 0.0
WX > 0
WY > 0

Caractéristiques générales (* dans ce champ figurent les informations
... recensées en SAU-84.2)

Propriétés Si MAXIT > 0 alors nbiter (VD03A) < MAXIT
(* sinon, non limité)

III.1.2.4 - Discussion sur la dépendance de la modélisation des objets vis-à-vis du domaine d'application

Le schéma de reconnaissance des différents concepts représenté sur la figure 4 semble propre au domaine du calcul numérique, mais dépendant de la classe de problèmes envisagés, à une exception près. Nous n'avons pas testé notre modélisation des objets sur les connaissances relatives aux domaines des statistiques et des calculs de probabilités ; ces domaines sont parfois assez complexes, et demanderaient davantage de temps pour voir si ils rentrent aisément dans le cadre de notre modélisation.

D'autre part, la structure module ne permet pas la représentation de certains modules ne répondant pas aux normes considérées comme standard pour FORTRAN IV, ce langage étant déjà assez complexe. Ceci est un choix lié à notre application.

III.1.3 - Fonction résolution

Le traitement de cette fonction constitue la partie intelligente du système. Nous commencerons par exprimer notre problème concret de façon formelle. Puis nous présenterons les divers types de connaissances qui permettront de résoudre un problème posé par l'usager ; nous en donnerons une représentation externe formelle sous forme de règles. Nous examinerons ensuite une façon de prendre en compte l'incertitude sur l'adéquation parfaite d'un outil de résolution au problème à résoudre. Enfin, nous décrirons complètement les stratégies de résolution de problème, en mode automatique ou assisté par l'usager.

III.1.3.1 - Formalisation du problème de la résolution

Un problème (P_u) défini par l'usager est caractérisé par :

- des données, d_u (ex : matrices d'un système linéaire)
- des résultats attendus, r_u (ex : solution d'un système linéaire)
- un ou des opérateurs, op_u (ex : résolution de système linéaire) tels que l'application de op_u à d_u renvoie r_u.
- une suite $(c_i)_{u\ i=1...n}$ de critères sur les performances attendues, ordonnés par ordre de priorité décroissante.
  (ex: précision, place mémoire,...)
- un type d'outil cherché, t_u, appartenant à
  {type de méthode, méthode, algorithme, module}

Les connaissances du système concernent :

- des outils, permettant de résoudre des problèmes numériques
  (ex: algorithme de résolution de système linéaire LDL^T)
- des relations de comparaison entre outils selon des critères (sur les performances attendues)
  (ex: LDL^T est plus précis que LL^T)

Les outils sont caractérisés par les connaissances suivantes :

- leur type et leur nom

- opérateur(s) qu'ils permettent d'appliquer
  (ex: résolution de système linéaire)

- type et contraintes associés aux données sur lesquelles ils s'appliquent
  (ex: matrice, symétrique)

- type et contraintes associés aux résultats qu'ils permettent d'obtenir
  (ex: matrice, triangulaire - pour une méthode de décomposition triangulaire)

- les outils qu'ils concrétisent (ou "antécédants")
  (ex: pour l'algorithme LDL^T,
  type de méthode : direct
  méthode : décomposition symétrique

- une connaissance approximative, experte, sur la qualité numérique des résultats que l'on peut obtenir, selon celle des données.

Les relations de comparaison entre outils selon un critère peuvent s'écrire comme $\lesssim_c$ avec

$x \lesssim_c y$ si et seulement si y est meilleur que x selon le critère c, x et y étant de même type.

Remarque : $\lesssim_c$ définit une relation d'ordre partiel, et peut être conditionnée par les données de (P_u).

Le système que nous voulons concevoir aura la tâche suivante : Il s'agit de trouver l'ensemble E_n des outils de type t_u, les ensembles E_i pouvant être définis par récurrence par :

- E_0 est un ensemble d'outils de type t_u tels que, pour tout o élément

de E_0,

. les opérateurs op_u peuvent être traités par o.

. les contraintes d'application de o sont vérifiées par données d_u de l'usager.

. les contraintes sur les résultats attendus r_u sont vérifiées par les résultats obtenus par l'application de o.

. la qualité numérique que l'on obtiendrait par application de o est relativement "correcte" (à définir...)

- et, pour i=1...n, E_i est l'ensemble des meilleurs outils de E_{i-1}, selon le critère C_i, et les données d_u de l'usager.

III.1.3.2 - Différentes connaissances utilisées pour résoudre un ou des problème(s)

Ces connaissances reflètent le savoir-faire de l'expert numéricien quant aux façons de choisir une méthode, un algorithme ou un module pour résoudre un problème numérique. Nous parlerons des règles de la base de connaissances pour les désigner ; ce sont également ces règles qui seront déployées à l'usager lors de la justification des résultats du système. Nous avons pu classer ces connaissances en plusieurs groupes.

- Règles inter-objets

Elles permettent d'exprimer le lien entre deux objets adjacents d'un arbre de résolution de problème ; plus précisément, elles servent à indiquer, pour un problème donné, et une fois une direction choisie pour le résoudre, comment concrétiser cette direction pour obtenir une solution réelle.

ex : *pour résoudre un système linéaire donné, si l'on a déjà retenu le type de méthode direct, et si la matrice du système est symétrique, alors retenir la méthode de Cholesky.*

Ces règles expriment donc plus que le lien père-fils : elles font intervenir comme critère des contraintes portant sur les <u>données</u> du problème ; nous les appelons : contraintes "mathématiques".

Remarque : les critères portant sur les résultats attendus du problème sont de deux types :

- propriété des variables résultats ; elles sont prises en compte lors de la définition du problème.
  ex : décomposition d'une matrice faisant intervenir une matrice symétrique.

- critères sur les performances attendues de l'outil de résolution cherché ; ils sont pris en compte par une autre classe de règles (voir ci-dessous).
  ex : précision désirée sur les résultats.

- règles de comparaison entre objets de même nature.

Elles permettent de comparer deux objets retenus à partir d'une même direction de départ, sur un critère de performances spécifié par l'usager (précision, place mémoire, ou temps d'exécution). Notons que des contraintes sur les données peuvent intervenir dans l'expression de ces règles.

ex : *Si les méthodes d'interpolation polynômiale de Lagrange et d'Hermite ont été a priori retenues via les règles inter-objet, et si la dimension du problème est inférieure à 13, alors la méthode d'Hermite sera plus précise que celle de Lagrange.*

Remarque : Chaque règle ne traite que d'un critère. Or l'usager peut vouloir trouver un outil de résolution "aussi précis que rapide". Mais ces critères sont souvent deux à deux contradictoires ; en général, la précision va à l'encontre de la rapidité d'exécution ou de l'occupation minimale en place mémoire ; aussi l'expert a-t-il du mal à résoudre correctement ce problème "à la main". Ce ne sera pas plus facile à résoudre pour un système informatique, une raison essentielle étant qu'il s'agit de traiter un problème de nature plutôt qualitative que quantitative.

Nous excluons donc tout problème de compromis entre critères.

C'est pourquoi on demande à l'usager de classer ses critères sur les performances par ordre de priorité décroissante.

D'autre part, notons que ces relations de comparaison entre objets de même nature n'expriment qu'un ordre partiel et non total entre objets. En fait, elles ne comparent que des objets ayant un même objet-père dans l'arbre de résolution ; par exemple, les méthodes de Lagrange et d'Hermite sont toutes deux de type polynomial. Cela s'explique par le fait qu'il est naturel de vouloir comparer deux méthodes proches par leur principe général ; au contraire,deux méthodes utilisant des principes différents seront difficilement comparables : au besoin, il s'agit plutôt de comparer leurs principes, en fonction des données du problème à résoudre.

- Règles de déduction de propriétés

Ces règles expriment des déductions entre propriétés.

ex : *si une matrice est diagonale, alors elle est bande*

Le système peut utiliser ces connaissances pour chercher des solutions dégradées en cas d'échec de "bonne" solution. Plus couramment, ces règles permettent d'éviter de poser des questions inutiles à l'usager en cours de recherche de solution, lorsque celui-ci a déjà exprimé des contraintes assez "serrées" sur les données de son problème.

Ces trois types de règles sont très importants pour le système de base que nous désirons réaliser. Si l'on veut élargir les possiblités du système vers la résolution de problèmes plus complexes, d'autres types de règles seront à prendre en compte, parmi lesquels :

- Règles liées aux opérateurs

ex : *si A est une matrice symétrique (supposée inversible), alors son inverse est également symétrique.*

Ce type de règle permet de générer des propriétés sur des variables lors d'enchaînements de sous-problèmes.

- Règles d'interprétation de résultats

ex : *si A est une matrice alors*
e(A) < 1 <==> A est à diagonale principale.
(où e(A) = rayon spectral de la matrice A)
(même utilité que les règles liées aux opérateurs)

- Règles de décomposition de problèmes en sous-problèmes

ex : *soit à résoudre un système linéaire AX = B ; une façon de résoudre ce problème en deux étapes consiste à :*
1 - résoudre le système $AX_0 = B$ par une méthode de type direct
2 - résoudre le système AX = B, X_0 étant considérée comme une valeur approchée de X, par une méthode de type itératif.

De telles règles indiquent non seulement comment décomposer un problème en un enchaînement de sous-problèmes, mais éventuellement aussi comment résoudre ces sous-problèmes.

III.1.3.3 - Formalisation externe

De façon externe, ces connaissances s'exprimeront aisément sous forme de règles de production, comme on le voit en III.1.3.2. Une logique d'ordre 1 semble préférable pour faciliter l'expression du fait que ces règles doivent pouvoir être appliquées à n'importe quel problème défini par l'usager.

III.1.3.4 - Expression de l'incertitude sur les solutions proposées par le système (ou l'expert)

Dans la réalité, un expert numéricien ne peut jamais affirmer avec certitude que l'outil qu'il propose pour résoudre un problème numérique donnera des résultats parfaits (cf. III.1.1.2) : des incidents plus ou moins graves peuvent survenir, dûs à l'algorithme, et aux particularités numériques des données. Pour rendre l'usager conscient de ce risque de manque de qualité des résultats, cette notion d'incertitude doit être exprimée dans la base de connaissances, puis prise en compte

par le système en cours de résolution de problème.

III.1.3.4.1 - Méthode d'expression retenue

Plusieurs approches ont été envisagées, plus ou moins sophistiquées, pour exprimer cette notion (cf. II.1.3.3.1). Nous avons choisi ici une méthode très simple.

Nous ne représenterons l'incertitude que dans les règles inter-objets, celles qui ne tiennent compte que de contraintes sur les données du problème. Chacune de ces règles sera en réalité affectée d'un coefficient compris entre 0 et 1, représentant une estimation sur les chances de réussite de l'outil qu'elle propose, en fonction des conditions précisées sur les données ; ce coefficient est donc assez approximatif.

Un exemple complet de règle inter-objet peut donc être :
"si le type de méthode direct a été retenu pour résoudre le système AX=B, si A est symétrique et son ordre inférieur à 50, alors on peut retenir la méthode de Cholesky avec 80% de chances de réussite".

A partir de cet exemple, nous voyons que l'on peut faire varier le coefficient de réussite en fonction des contraintes numériques ; ici, plus l'ordre de la matrice sera élevé, plus la qualité numérique des résultats risque d'être mauvaise, même si Cholesky s'applique toujours en théorie. On peut donc exprimer de façon détaillée cette relation, en faisant varier les facteurs contraintes numériques et coefficients de réussite, pour essayer de coller au mieux aux données des problèmes des usagers - ceci conduisant à une multiplication du nombre de règles inter-objets.

Par contre, la notion d'incertitude n'est pas exprimée dans les règles de comparaison entre objets de même nature sur critères de performances, car les données numériques ont moins d'incidence sur leur validité.

La prise en compte de ces coefficients de réussite lors de la résolution de problèmes sera présentée avec la stratégie de résolution

en III.1.3.5.

III.1.3.4.2 -Limites inhérentes à ce mode d'expression

Les premières remarques sont liées à la façon dont ont été évalués ces coefficients de réussite ; on en déduit des limites sur le type d'exploitation qui pourra être faite de ces coefficients.

Ces coefficients ne seront pas évalués à partir de statistiques effectuées sur des observations de résultats de problèmes numériques déjà résolus ; ils seront affectés à partir d'une donnée très qualitative, l'expérience de l'expert. Dans le même ordre d'idée, ils évoluent de façon discontinue en fonction de variation de plages de contraintes numériques ; on aurait pu penser à un coefficient qui serait une fonction continue de plusieurs variables (ordre d'une matrice, largeur de bande de la matrice, nombre de conditionnements de la matrice, etc...) : cela permettrait de tenir compte des données de l'usager de façon plus précise, mais ces fonctions seraient difficiles à définir, et pourraient poser des problèmes de mise-à-jour (cf./SAU 85/).

D'autre part, l'exemple ci-dessus montre qu'il n'est pas fait mention, dans une règle inter-objets, du coefficient de réussite avec lequel la direction initiale (ie, le type de méthode direct) a été retenue.

En conséquence, il est hors de question d'exploiter ces coefficients de façon absolue, car ce serait leur accorder plus de valeur qu'ils n'en ont en réalité. Ils ne peuvent avoir de sens que relativement les uns par rapport aux autres, et encore à condition que leurs différences soient suffisamment importantes. Ceci sera pris en compte lors de la stratégie de résolution de problèmes par le système.

III.1.3.5 - Stratégies de résolution du système

De façon à en permettre une utilisation très souple par n'importe quel type d'usager, le système doit pouvoir présenter deux modes de résolution : automatique, ou assisté par l'usager - ces deux

modes pouvant être plus ou moins imbriqués. Nous expliquerons donc ces deux types de stratégies ; puis nous envisagerons la façon de traiter les échecs compatible avec la stratégie du système.

En fait nous avons étudié deux classes de résolution possibles : soit chercher un outil de type donné pour résoudre un problème posé par l'usager, soit vérifier l'adéquation, d'un outil donné à un problème donné. Ce second type de résolution entrera en jeu également dans le cadre de résolution assistée par l'usager.

Enfin, notons qu'au cours de toute résolution, si l'usager n'est pas certain de l'existence d'une contrainte donnée sur une variable du problème, le système considère que cette contrainte n'est pas vérifiée. Cette option permet de jouer la carte de la sécurité, car un algorithme d'inversion d'une matrice définie-positive, appliqué à une matrice qui ne le serait pas, risque de donner des résultats catastrophiques. Dans notre cas au pire, les performances de l'outil que proposera finalement le système ne seront pas optimales, mais les résultats numériques obtenus auront plus de chances d'être corrects.

III.1.3.5.1 - Résolution automatique

III.1.3.5.1.1 - *Recherche d'un outil pour résoudre un problème*

III.1.3.5.1.1.1 - Stratégie de recherche

Une fois un problème défini, le type de l'outil désiré spécifié, le système peut procéder par étapes pour chercher ce(s) outil(s). Le principe général consiste à chercher les meilleurs types de méthodes susceptibles de convenir, puis à partir de chacun d'eux, les méthodes qui pourraient a priori donner les meilleurs résultats, et ainsi de suite, jusqu'à obtenir des outils du type désiré.

Nous allons donc détailler un niveau de sélection, c'est-à-dire comment on choisit les "meilleurs" outils de type T
(avec T $\in$ {méthode, algorithme, module}) à partir d'un outil de niveau t

(respectivement avec t ∈ {type de méthode, méthode, algorithme}) déjà retenu par le système au cours de sa recherche.

Cette recherche peut se faire en trois étapes :

e1 - Sélection sur les contraintes portant sur les données du problème

Elle s'opère en appliquant des règles inter-objets ayant en partie condition l'outil de type t déjà retenu. Ces règles permettent d'obtenir un ensemble d'outils de type T, éventuellement applicables avec différents coefficients de réussite (déterminés par les règles qui ont pu s'appliquer). Soit E_0 cet ensemble.

e2 - Prise en compte des coefficients des règles inter-objets

Parmi les éléments de E_0, on décide de ne conserver que les outils affectés des plus forts coefficients de réussite. Concrètement, le seuil d'acceptabilité est défini arbitrairement par 1/3 x max des coefficients affectés aux éléments de E_0 ; ce coefficient de 1/3 permettra d'obtenir les résultats voulus. Cette méthode permet de ne tenir compte des coefficients que de façon relative, comme spécifié en III.1.3.4.2.

Nous obtenons ainsi un sous-ensemble E_1 de E_0.

e3 - Prise en compte des contraintes sur les performances

A partir de E_1, le système applique des règles de comparaison entre objets de même nature, pour éliminer tous les outils qui sont moins bons que d'autres, au sens des critères de performances définis par l'usager.

Il peut y avoir plusieurs critères, ordonnés par priorité décroissante pour l'usager. Dans ce cas, le système applique d'abord la relation d'ordre sur le critère le plus important et génère déjà un sous-ensemble A_1 de E_1 ; puis il applique la relation d'ordre sur le critère placé au second niveau de priorité, et génère A_2, inclus dans A_1, etc... jusqu'à examen de tous les critères.

Ce processus permet de régler simplement le problème de la contradiction entre contraintes de performances. Implicitement, nous

considérons que deux outils de même niveau non comparables via les règles de comparaison d'objets de même nature sur un critère, et concrétisant le même principe, sont équivalents vis-à-vis du critère considéré.

Le système peut éventuellement signaler à l'usager les cas où des critères sont contradictoires. En notant "sel/C_i (E)" la sélection des meilleurs éléments de E selon C_i, ceci correspond au cas où :

$$\begin{cases} \text{sel}/C_1\ (E) = A \\ \text{sel}/C_2\ (E) = B \\ A \cap B = \varnothing \end{cases}$$

A l'issue de ce processus, nous obtenons un sous-ensemble E_2 de E_1, comprenant les "meilleurs" outils de type T vis-à-vis du problème de l'usager, qui concrétisent un outil donné de type t.

Si T n'est pas le type d'outil désiré par l'usager, on réitèrera les trois étapes e_1, e_2, e_3 à partir de chaque élément de E_2.

III.1.3.5.1.1.2 - Présentation des outils résultats

a - *Incertitude liée aux outils résultats*

Une fois que le système a trouvé un ou des outil(s) du type désiré par l'usager, il doit lui indiquer que ces propositions ne donneront pas forcément des résultats miraculeux. Cela peut se faire de deux façons.

De manière systématique, les outils proposés par le système seront affectés d'un coefficient compris entre 0 et 1 et donnant une vague indication sur leurs chances de réussite.

Comment calculer ce coefficient ? Différentes théories peuvent être appliquées (cf. II.1.3.3.1). Nous ne ferons qu'une opération élémentaire, consistant à faire le produit des coefficients

affectés aux règles inter-objet ayant permis d'aboutir à l'outil proposé.

L'exploitation de ces coefficients appelle donc à quelques remarques :

1°) - En cours de recherche, lors d'une étape e_2, le système ne tient pas compte des coefficients rencontrés au cours du chemin l'ayant conduit à cette étape ; ceci est lié au fait que, pour une règle inter-objets du type : "si la direction d a été retenue, en présence des contraintes c sur les données, alors on peut retenir l'outil o avec x chances de réussite", la valeur du coefficient x ne tient pas compte des chances de réussite avec lesquelles la direction d avait été retenue (par simplification). Cela est lié au fait que :

- l'on ne veut pas et ne peut pas utiliser ces coefficients de manière rigoureuse.

- ces coefficients ne serviront qu'à donner au système une idée de comparaison des chances de réussite entre outils "comparables ?, ou concrétisant les mêmes principes généraux.

2°) - Plus le type d'outil recherché par l'usager est concret, plus les valeurs des coefficients affectés aux outils tendront vers 0, par définition de ces coefficients. Ce n'est pas génant dans la mesure où l'usager ne pourra comparer que des coefficients associés à des outils de même type ; d'autre part, ces coefficients n'ont qu'une valeur indicative, pour des raisons liées à la remarque précédente : leur rôle reste essentiellement de rappeler à l'usager que les outils proposés par le système ne donneront pas forcément des résultats corrects.

b - *Outils de contrôle associés aux outils résultats*

Toujours dans le but de rappeler à l'usager les problèmes de qualité des résultats numériques, certaines règles inter-objets précisent, en fonction des contraintes sur les données de leur partie condition, des moyens d'actions à entreprendre pour obtenir de meilleurs

résultats numériques.

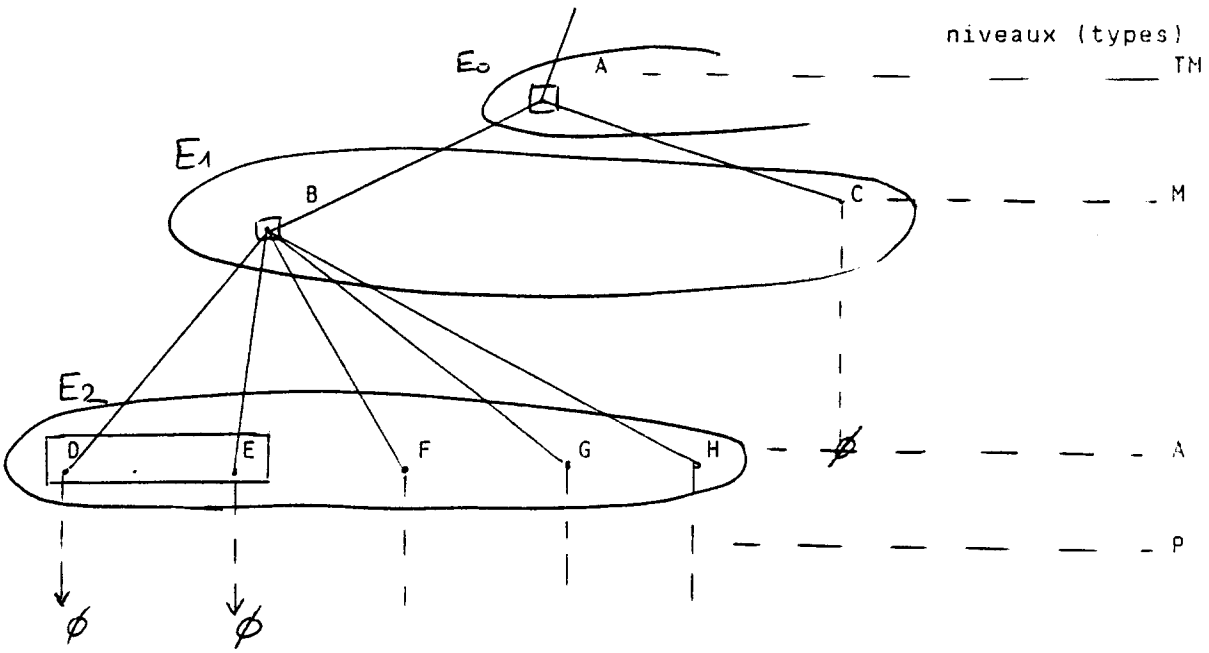
ex : *"si le type de méthode direct a été retenu pour inverser une matrice, si cette matrice est symétrique, d'ordre compris entre 100 et 500, et mal conditionnée, alors on peut appliquer la méthode de Cholesky avec 60% de chances de réussite, à condition de préconditionner la matrice par la méthode M"*

Si le système a dû appliquer des règles inter-objet, mentionnant des outils de contrôle, pour aboutir à la proposition d'un outil donné, l'application de ces outils de contrôle sera conseillée à l'usager lors de la présentation des outils résultats.

III.1.3.5.1.1.3 - Traitement des cas d'échecs

La notion de cas d'échec est ici très relative. Nous parlons de cas d'échec lorsque, pour un problème de l'usager, si le système en cours de recherche préconise une certaine direction de recherche, cette voie n'aboutit pas à l'obtention d'un outil du type désiré par l'usager. Il y a deux raisons possibles à cela. La première est le cas de la base de connaissances incomplète ; supposons par exemple que le système connaisse la méthode du gradient conjugué, mais ne connaisse pas d'algorithme concrétisant cette méthode. La seconde est le cas où la base est complète (au sens ci-dessus), mais les contraintes sur les données du problème sont telles qu'aucune règle inter-objet ne puisse s'appliquer à partir d'un moment donné de la recherche du système. Enfin, rappelons qu'un des objectifs du système est d'éviter que l'usager reste sans aucune proposition.

Supposons le cas de figure suivant :



Ce schéma représente une portion de l'arbre de résolution d'un problème, en cours de recherche. Supposons qu'à un instant donné, le système ait suggéré le type de méthode A comme meilleur (vis-à-vis du problème à résoudre) de tous les types de méthode susceptibles de convenir. Sur le schéma, les ensembles de points entourés par un ovale représentant des outils (ou directions de recherche) obtenus par l'application des règles inter-objets, et les ensembles entourés par un rectangle représentent les meilleurs outils (étapes e_2 et e_3) parmi les précédents. La direction retenue par le système est donc B, puis {D,E} ; le système cherche donc à avancer dans cette voie en cherchant des modules (type "1" pour logiciel) implémentant les algorithmes D et E. Dans le cas où le système ne parvient pas à trouver de tels modules, on peut donc dire que la voie prometteuse que suivait le système a conduit à un échec.

Pour pouvoir proposer néanmoins à l'usager une solution restant très proche dans cette voie, la stratégie du système consiste à revenir sur l'ensemble E_2, à repasser par les étapes e_2 et e_3 (élimination des outils à faibles coefficients, et détermination des meilleurs outils restant vis-à-vis des contraintes sur les performances) à partir de l'ensemble $E_2 \setminus \{D,E\}$, puis continuer sa recherche à partir de là. Supposons qu'à l'issue de e_3 sur $E_2 \setminus \{D,E\}$, les meilleurs outils

soient {F,G} ; si l'on obtient au moins un module concrétisant F ou G, c'est bon ; si l'on retombe sur un cas d'échec, il ne reste qu'à explorer la voie suggérée par H.

Si tous les éléments de E_2 ont conduit à des cas d'échec, la stratégie consiste à faire un retour arrière d'un niveau, et à revenir sur E_1, ensemble des méthodes de type A ayant satisfait aux règles inter-objets. Le processus décrit pour E_2 sera alors le même pour E_1, et le système considérera alors une voie de recherche appliquant ici les idées mises en oeuvre dans la méthode C. Le système ne trouvant pas d'algorithme concrétisant C et adapté au problème de l'usager, A est en échec. Le processus de traitement d'échec consiste alors à remonter encore d'un niveau, et de repartir sur les étapes e_2 et e_3 sur $E_0 \setminus \{A\}$ etc...

Finalement, le système ne tombe sur un cas d'échec réel que si aucune voie indiquée par les règles inter-objets ne peut aboutir à un objet du type désiré par l'usager. Par contre, il peut être amené à fournir des solutions ayant franchement peu de chances de réussir, juxtaposées à des solutions plus prometteuses (issues d'autres voies de recherche ayant abouti normalement), avec ce genre de stratégie.

L'usager doit donc être mis au courant de ces "anomalies". Les coefficients affectés aux solutions proposées peuvent refléter cela ; mais le moyen le plus sûr consiste carrément à expliquer la stratégie du système à l'usager, et lui dire pourquoi et comment certaines solutions dégradées ont été trouvées pour pallier à l'avance au risque d'échec total éventuel.

III.1.3.5.1.2 - *Vérification de l'adéquation d'un outil à un problème*

Cette fois, le rôle du système est de voir si un outil de résolution, que lui propose l'usager, est tel qu'il l'aurait lui-même retenu au cours d'une recherche automatique.

La stratégie utilisée s'inspire de celle décrite précédemment,

mais cette fois le système ne parcourt que la branche de l'arbre de résolution susceptible d'aboutir à cet outil. Par exemple, si l'usager veut savoir si un algorithme A permet bien de résoudre son problème, voici comment procédera le système : A concrétise une méthode M, elle-même étant d'un type TM ; le système vérifiera donc s'il retient TM comme meilleur outil susceptible de résoudre le problème, puis si M fait partie des meilleures méthodes, etc...

Il y a donc plusieurs cas de rejet de l'outil proposé :

- soit un des antécédents de A (TM,M ou même A) n'apparait pas dans un ensemble obtenu à l'issue d'une étape e_1 (règles inter-objets) ; comme il n'est pas compatible avec les contraintes sur les données de l'usager, A ne peut pas l'être non plus.

- soit un des antécédents de A ou A est éliminé des voies retenues par le système à l'issue des étapes e_2, ou e_3. Cette fois, le rejet se justifie par le fait que le système connaissait des meilleures voies pour résoudre le problème de l'usager. Cette raison est donc moins "grave" que la précédente, et ne peut pas justifier un rejet aussi catégorique, mais plutôt un avertissement à l'usager.

Ce mode de fonctionnement peut être envisagé pour l'usager qui a quelques idées sur le genre d'outil qu'il désire obtenir, mais aussi pour l'expert qui veut vérifier le raisonnement du système lorsque le résultat obtenu ne correspond pas à ce qu'il en attendait.

III.1.3.5.2 - Résolution assistée par l'usager

Nous considérons ce mode de résolution comme un mixage des deux fonctions décrites dans le paragraphe précédent.

Par résolution assistée, nous entendons la démarche consistant pour l'usager à poser un problème, en définissant le type d'outil recherché, tout en spécifiant une direction de recherche qu'il désire voir envisager par le système (ex : une méthode donnée).

Dans un premier temps, la tâche du système consiste à vérifier qu'il aurait lui-même retenu la direction indiquée par l'usager. Si sa conclusion est affirmative, le système peut alors effectuer sa recherche en mode automatique, à partir de la direction donnée (cf. paragraphe précédent). Si elle est négative, il doit le signaler à l'usager ; suivant la raison de cette conclusion, l'usager peut alors décider soit d'abandonner son idée initiale, soit de demander au système de continuer tout de même la recherche, à ses risques et périls.

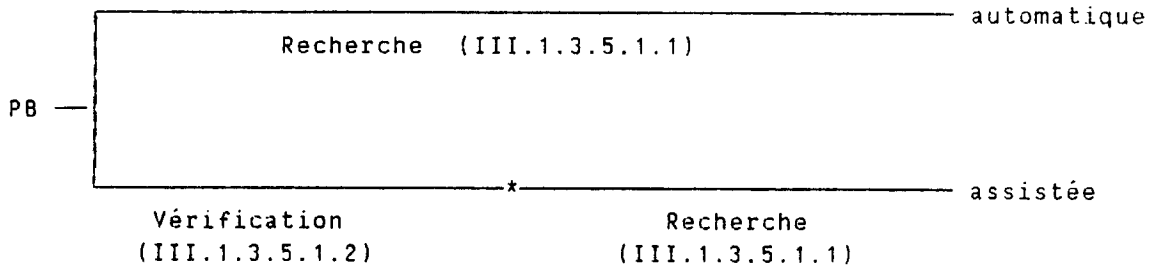
Dans le second cas, il n'y a plus lieu pour le système de faire un travail sur la certitude accordée aux résultats qu'il trouvera ; sa recherche n'ayant plus beaucoup de sens vis-à-vis de ses propres connaissances.

On peut imaginer également que, si la raison du rejet de la direction usager par le système est liée aux critères sur les performances attendues par l'usager, le système indique à l'usager quelles directions il aurait lui-même préférée. Dans ce cas, il doit s'assurer, pour éviter des calculs inutiles, que ces directions peuvent effectivement aboutir à des outils du type recherché par l'usager.

Donc en réalité, ce mode de résolution assisté par l'usager reste relativement contrôlé par le système, toujours pour que l'usager demeure bien conscient de la mauvaise qualité des outils qu'il risque d'obtenir comme solution.

D'autre part, il pourrait y avoir un mode de résolution beaucoup plus assité, où l'usager interviendrait pour prendre des décisions, en cours de résolution du système. Ces idées font appel à des techniques de contrôle plus élaborées et feraient l'objet d'un travail intéressant.

Le schéma suivant résume donc les deux optiques :



III.1.3.5.3 - A propos d'apprentissage

Nous n'avons pas envisagé de fonction d'apprentissage pour notre système. Toutefois, ce serait une extension très utile, en particulier pour le traitement des échecs. Il serait intéressant par exemple que le système sache ce qu'il sait sans avoir à parcourir sa base de connaissances. La nécessité de cette extension deviendra tout à fait apparente dès que la base de connaissances aura atteint un certain seuil de développement.

III.1.4 - Choix de PROLOG

Nous utiliserons PROLOG comme langage d'expression interne des connaissances. Le but de ce paragraphe est de justifier ce choix, et de montrer comment ce langage peut être mis en oeuvre dans notre base de connaissances.

III.1.4.1 - Choix de la logique du premier ordre pour la représentation interne des connaissances

Nous avons vu en III.1.2.3.1 qu'un même outil pouvait résoudre plusieurs problèmes assez proches (ex : résolution de système linéaire, calcul de déterminant, décomposition de matrice, inversion de matrice ...) dans certains cas ; dans les règles de la base de connaissances concernant ces outils, les problèmes à résoudre apparaissent en condition d'application. A l'inverse de la logique des propositions, celle des prédicats nous permet d'exprimer les conditions sur les problèmes de façon intensive, et d'appliquer ainsi une seule règle à des tas de problèmes particuliers de l'usager.

D'autre part, le langage de la logique des prédicats est modulaire et très déclaratif, ce qui facilite la mise à jour de la base de connaissances, et l'expression des données des problèmes à résoudre.

Enfin, nous avons vu en II.1.3.2 qu'il semblait plus simple de générer des explications des raisonnements à partir de connaissances décrites en logique qu'avec un formalisme interne de réseaux ou de structures. De façon générale, une raison du choix de ce formalisme tient au fait que nous disposons de résultats théoriques et d'outils informatiques relativement efficaces pour travailler avec (surtout, pour des clauses de Horn).

Nous allons donc essayer d'exprimer nos connaissances en logique des prédicats.

III.1.4.2 - Pourquoi un langage de type PROLOG ?

En II.1.6.1, nous avons vu les avantages que présentait PROLOG lors de l'écriture d'un système expert utilisant la logique des prédicats pour exprimer ses connaissances - même s'il ne suffit pas en tant que moteur d'inférences.

D'autre part, nous disposons d'un outil de type PROLOG qui n'a pas les faiblesses de PROLOG au niveau environnement de programmation : il s'agit de PROLISP (/ZAN 84/), c'est-à-dire un interpréteur PROLOG écrit et plongé dans l'environnement LISP assez puissant MACLISP dans lequel a été écrit MACSYMA.

Enfin, la restriction de la logique des prédicats aux clauses de Horn nous permet quand même d'exprimer nos connaissances assez facilement. De plus, nous ferons l'hypothèse de monde fermé, ce qui simplifie l'écriture du savoir-faire de l'expert (/GIA 85/).

III.1.4.3 - Représentation des informations de la base de connaissances

III.1.4.3.1 - Les objets

De façon interne, la structure qui permettait de décrire un objet de façon externe est représentée par une assertion. Les différents termes de cette assertion correspondent aux champs d'une structure, et ainsi de suite par décomposition des termes en sous-termes.

III.1.4.3.2 - Les règles

De façon interne, une règle se représente par une clause. Les littéraux négatifs expriment la partie condition de la règle, et le littéral positif exprime la partie conclusion de la règle.

III.1.4.4 - Remarque : ordre des règles

Puisque nous utiliserons PROLOG, notre remarque sera liée à la stratégie de l'interpréteur PROLOG - en dehors de la présence de méta-connaissances déclaratives.

L'ordre des règles est important avec une stratégie de type PROLOG. Si de plus, la programmation de la base de connaissances fait intervenir des prédicats évaluables introduisant du déterminisme (genre /), l'ensemble des solutions que trouvera le système dépendra de l'ordre dans lequel les connaissances seront exprimées dans le système.

Or une propriété importante d'un système expert est de pouvoir être facilement mis à jour. L'idéal est donc de pouvoir insérer une connaissance n'importe où dans la base, sans introduire de conséquence inattendue sur les résultats du système.

C'est dans cet objectif qu'il vaut donc mieux ne pas introduire de déterminisme dans l'expression des connaissances de l'expert, à moins que le gestionnaire du système n'ait lui-même des connaissances dans le langage de représentation interne.

Concrètement, cela signifie que, pour exprimer les connaissances livrées "en vrac" par l'expert, et qui seront servies à l'usager comme explications de résultats du système, on évitera d'introduire des prédicats évaluables amenant du déterminisme dans le contrôle.

Remarque :

Le problème de l'ordre des règles serait à considérer différemment si l'on disposait de méta-connaissances. En effet, on peut supposer que des méta-règles réordonnent explicitement les règles avant la résolution, ou bien ne font opérer la résolution que sur un sous-ensemble de règles, etc ...

III.1.5 - Interface utilisateurs/système

Par utilisateurs nous entendons usagers et gestionnaires. Nous donnerons d'abord des spécifications très générales concernant les fonctions que doit remplir cet interface, et sur l'état d'esprit avec lequel il sera développé. Puis nous développerons le traitement de la fonction de génération d'explications sur le raisonnement du système, fonction aussi importante que celle consistant à résoudre les problèmes des usagers.

III.1.5.1 - Spécifications générales

III.1.5.1.1 - Interface usager

Rappelons qu'il doit principalement permettre à l'usager de :

- de poser des problèmes au système,
- de comprendre les raisonnements du système et les solutions qu'il propose,
- d'accéder à toute forme de documentation sur du vocabulaire, un outil de type : type de méthode, méthode, algorithme, module, ou de contrôle de qualité.

Nous voulons que le système soit très simple à utiliser par des néophytes, contrairement à certains logiciels spécialisés. L'interface doit donc être très convivial et attrayant. En particulier, il est souhaitable de faire en sorte que l'usager puisse aborder le système sans recourir nécessairement à un manuel d'utilisation. On évitera par exemple d'obliger l'usager à apprendre un langage particulier pour dialoguer avec le système (cas de PROTRAN décrit en II.2.2.1.1.).

Mais d'autre part, nous ne consacrerons pas l'essentiel de nos efforts sur l'interface dans un premier temps. En effet, la conception d'un interface bien adapté aux besoins rencontrés peut faire l'objet d'un travail assez long, sur des techniques très sophistiquées ; ceci n'est pas l'objet de notre projet.

Nous nous limiterons donc à des techniques très simples, qui ne doivent pas être trop contraignantes pour l'usager. Toutefois, ces techniques ne suffiront probablement pas pour un système "grandeur nature". Elles doivent surtout suffire, dans un premier temps, à tester le système de façon agréable. Ce sont nos seuls objectifs actuellement.

L'interface usager sera donc constitué de menus hiérarchisés et de mots clés. Il est de toutes façons très lié aux moyens matériels dont on disposera pour implémenter le système. Nous n'en dirons pas plus à ce sujet.

III.1.5.1.2 - Interface gestionnaire

Le gestionnaire est censé bien connaître le système et le manipuler de façon privilégiée.Il est tout à fait normal de lui imposer l'utilisation d'un langage assez sophistiqué lui permettant d'effectuer toutes sortes d'opérations sur le système (en restant dans des techniques d'interface assez simples).

Ce langage peut reprendre les mots clés servant à exprimer les structures pour décrire les objets ou avoir une syntaxe très simple pour décrire les règles.

Un problème crucial pour l'interface gestionnaire est celui de la gestion de la cohérence entre les diverses informations introduites dans la base de connaissances. A un premier niveau, il doit exister des vérifications d'ordre syntaxique de différentes sortes. Les règles doivent mentionner des outils connus (et décrits) du système ; certains champs des structures doivent absolument contenir des informations correctement énoncées (alors que si d'autres champs sont moins remplis, les conséquences sur le fonctionnement du système sont moins importantes) ; si l'on enlève un objet de la base, les règles qui mentionnaient cet objet n'ont pas de raison de subsister dans la base ; si les informations d'un champ de la structure module sont modifiées, cela peut avoir des conséquences sur d'autres champs, qui doivent être respectées, etc... Ces vérifications sont assez simples. A un second niveau, il doit y avoir des vérifications d'ordre logique ; des règles ne doivent par exemple pas se contredire ; la base doit rester consistante même si l'on ajoute ou enlève des règles (ce niveau concerne surtout les règles) ; il faut éviter autant que possible que des règles soient incluses les unes dans les autres par redondance logique, etc... Ce second niveau est déjà plus difficile à réaliser, car il s'agit souvent de problèmes non décidables dans le cadre de la logique du premier ordre. Enfin, un troisième niveau concerne une cohérence sémantique de la base, ce qui est encore plus difficile à vérifier ; un exemple typique de problème peut être le cas d'une base de connaissances générée par plusieurs experts n'utilisant pas le même vocabulaire.

Il est pourtant hors de question de laisser toute la gestion de la cohérence à la charge du gestionnaire. Donc en réalité, la réalisation de l'interface gestionnaire soulève des tas de problèmes, non seulement techniques mais surtout théoriques, qui seront laissés de côté au cours de ce travail - et feraient l'objet de travaux importants.

D'autre part, nous ne dirons rien sur l'aspect gestion des historiques d'utilisation, pour les raisons déjà mentionnées en I.3.2.1.2 (cf./SAU 84-2/).

III.1.5.2 - Explications liées au raisonnement du système

Nous donnons d'abord les idées maîtresses qui guideront notre travail, puis nous examinerons plus précisément les différents types d'explication que devra donner le système.

III.1.5.2.1 -Généralités

A priori, la fonction d'explications passe généralement pour secondaire vis-à-vis de celle de résolution ; toutefois, on se rend souvent compte ensuite que c'est une erreur, et qu'elle conditionne beaucoup le succès du système. Non seulement elle permet aux usagers de comprendre les raisonnements du système - ou d'apprendre comment il raisonne, mais surtout elle permet aussi aux experts de vérifier le contenu de la base de connaissances et éventuellement d'en corriger des erreurs ; en d'autres termes, elle facilite beaucoup la phase de validation du logiciel.

Les explications que nous désirons obtenir, concernant la résolution d'un problème par le système, doivent être complètement liées au problème ; par exemple, elles doivent mentionner les variables par le nom que leur a donné éventuellement l'usager, pour faciliter sa compréhension. Il en résulte que des techniques simples comme le stockage de textes explicatifs associés aux différentes bribes de connaissances du système est exclu, car "hors contexte". La fonction d'explications devra donc opérer de façon plus ou moins liée avec celle de résolution.

D'autre part, le terme "explications" peut être défini de plusieurs manières. Cela peut consister à livrer à l'utilisateur le contenu précis des règles utilisées dans le raisonnement du système ; l'idéal serait toutefois de présenter le raisonnement de manière synthétique et approprié à la vision globale de l'homme, plutôt que les étapes de résolution du système pas à pas, telles que décrites en III.1.3 ; d'autre part, les informations contenues dans la base de connaissances ne correspondent pas à toutes les connaissances de l'expert : certaines de celles-ci, implicites dans la base, mériteraient peut-être d'être mentionnés pour faciliter la compréhension de l'usager.

Nous désirons aborder trois types d'explications : comment ?, pourquoi ?; et pourquoi pas ?. La suite de ce paragraphe a pour but de définir ce que nous entendons par là, et de présenter les façons d'y répondre.

III.1.5.2.2 - Comment ?

Il s'agit pour le système d'expliquer à l'usager comment il a trouvé telle(s) solution(s) à son problème : ce type d'explications n'intervient donc qu'après une solution du système ; il rentre dans la classe des "explications positives" dans la mesure où il concerne un raisonnement effectivement suivi par le système.

Nous donnerons les grandes étapes du raisonnement, accompagnées des règles significatives (pour l'usager) utilisées - ie règles inter-objets, règles de comparaison entre objets de même nature sur les contraintes sur résultats, et règles de déduction de propriétés.

III.1.5.2.3 - Pourquoi ?

Ce type d'explication intervient en cours de raisonnement du système. Pendant la recherche d'une solution à un problème, le système est amené à demander à l'usager s'il peut considérer que telle contrainte sur une donnée du problème est vérifiée ; à ce moment, l'usager peut vouloir savoir pourquoi le système a besoin de cette information, ou encore quelle voie il essaie de suivre ; cette question peut être également posée lors de "debugging" de la base de connaissances par l'expert. Le système doit alors être capable de fournir une réponse compréhensible.

Puisque les connaissances utilisées pour la résolution sont exprimées sous forme de règles, la réponse que donnera notre logiciel sera constituée par :

- la règle que le système essaie d'appliquer,
- les prémisses précédant celle portant sur la contrainte faisant l'objet du pourquoi ?, et déjà montrées vraies par le système.

Par suite, l'usager peut poser plusieurs pourquoi ? successifs jusqu'à remonter le plus haut possible dans les choix qu'à déjà fait le système en cours de résolution.

Exemple de scénario envisageable :

Soient les règles :

R_1 - Une matrice définie-positive est symétrique,

R_2 - Si la matrice est d'ordre inférieur à 50, bien conditionnée, symétrique, etc..., alors on peut appliquer la méthode de Cholesky (80%).

soit, en cours de résolution de problème.

:

S(ystème) - la matrice est-elle définie positive

U(sager) - pourquoi ?

S - je cherche à appliquer R_1 (...) (pas de prémisse)

U - pourquoi ?

S - je cherche à appliquer R_2 (...), sachant déjà que la matrice est d'ordre inférieur à 50 et bien conditionnée.

etc...

Notons enfin que ce type d'explications appartient à la classe des "explications positives" puisqu'il porte sur un raisonnement suivi par le système.

III.1.5.2.4 - Pourquoi pas ?

Il s'agit là d'expliquer à l'utilisateur pourquoi un outil donné ne permet pas de résoudre un certain problème, d'après les connaissances du système. Ce n'est donc qu'une des nombreuses réponses possibles à la question pourquoi pas - (cf. seconde partie). Cela rentre dans le cadre des explications dites "négatives", car elles portent sur un raisonnement non suivi par le système.

Nous n'utiliserons pas ici une technique particulière, mais

nous contournerons artificiellement la difficulté en l'assimilant à un problème de résolution assistée par l'usager (cf. III.1.3.5.2) : il s'agit de savoir si le système aurait lui-même retenu l'outil proposé par l'usager (non, dans ce cas), et d'expliciter les raisons qui ont fait qu'il ne l'a pas retenu : en fait, il faudrait déterminer le point, dans l'arbre de résolution, où le système a abandonné la voie qui aurait pu aboutir à l'outil préconisé par l'usager.

Nous nous ramènerons donc à un problème d'explication de type comment ?, où l'on n'expliquerait que le début du raisonnement suivi par le système pour résoudre le problème donné.

III.1.6 - Interactions entre les fonctions documentaire et résolution

Nous montrerons où, au cours d'une résolution de problème, ces deux fonctions peuvent coopérer de façon à faciliter l'interaction système-usager. Nous en tirerons des conclusions quant à certains choix faits sur le formalisme de représentation des connaissances.

III.1.6.1 - Points d'interactions

Le premier cas concerne l'acquisition des données, ou des contraintes sur les données d'un problème par l'usager. La question posée alors par le système peut ne pas être comprise par l'usager, si elle fait intervenir un terme spécialisé du vocabulaire mathématique.

ex : *"la matrice est-elle tridiagonale ?"*.

Dans ce cas, une option peut permettre à l'usager de faire appel à une fonction documentaire expliquant le vocabulaire spécialisé apparaissant dans la question (ici tridiagonal) - ce mot étant représenté comme un concept (objet) dans la base de connaissances. Cette fois, il ne s'agit plus d'explication au sens résolution, mais au sens définition. On espère que l'usager pourra ensuite répondre correctement à la question posée par le système.

Le second cas concerne le mode de résolution dirigé par

l'usager. Nous avons vu en III.1.3.5.2 que lorsque l'usager préconise un outil O, de type T, le système va vérifier qu'il choisirait effectivement chacun des "ancêtres" de O dans l'arbre de résolution, en allant du type le plus abstrait au plus concret jusqu'à T. Il doit donc retrouver le plus rapidement possible tous ces ancêtres ; pour cela, il dispose des règles inter-objets ou du champ "problèmes résolus" (cf. III.1.2.3.1) de la structure représentant l'objet O. La façon la plus rapide est d'examiner les noms des problèmes décrits dans ce champ pour retrouver les informations nécessaires.

III.1.6.2 - Conséquence : remarque sur la représentation des connaissances

Dans les paragraphes III.1.2 et III.1.3, nous avons parlé de représentation externe des concepts de la base de connaissances sous deux formes, structures et règles. Puis nous avons retenu un même formalisme de représentation interne, celui de la logique des prédicats, par souci de simplification.

Toutefois, étant données les interactions entre fonctions résolution et documentaire, il serait intéressant d'envisager un formalisme de représentation des connaissances qui serait un hybride entre règles de production et frames, ainsi qu'un moteur d'inférences capable de traiter ce formalisme hybride. Nous n'avons pas du tout examiné cette voie, mais elle peut être prometteuse pour un développement ultérieur, si l'on dispose d'un outil adhoc suffisamment efficace.

III.2 - REALISATION D'UNE MAQUETTE : PELAGIE

Nous nous sommes volontairement restreints à un petite partie du domaine envisagé initialement. Nous ne traitons qu'un sous-ensemble de l'algèbre linéaire : les problèmes de résolution de système linéaire, d'inversion de matrice, de calcul de déterminant, et de décomposition de matrice sont abordés ; par contre, ne figurent pas dans la maquette tous les problèmes relatifs aux éléments propres d'une matrice.

D'autre part, comme annoncé en III.1, nous avons privilégié la mise en place de la fonction résolution par rapport à la fonction documentaire ; en outre, la fonction explication, lors de la réalisation de la maquette, n'a été traitée qu'après la fonction résolution.

Nous présenterons d'abord comment s'est déroulée la phase d'acquisition des connaissances expertes. Ensuite, nous aborderons l'implémentation des fonctions résolution et explication, et de quelques utilitaires annexes. Enfin, nous donnerons les caractéristiques physiques et les performances de la maquette, et quelques notes sur son mode d'utilisation.

III.2.1 - Acquisition des connaissances expertes

Nous n'avons pas parlé de cette phase au niveau des spécifications, car c'est l'étape la plus informelle de l'élaboration d'un logiciel de type système-expert. Nous évoquerons quelques idées générales sur notre acquisition des connaissances, puis nous présenterons la méthodologie que nous avons suivie.

III.2.1.1 - Généralités

D'entrée, nous avons fait l'hypothèse d'un intermédiaire entre l'expert et le système expert (appelé parfois ingénieur cogniticien) ; cette hypothèse présente les avantages et inconvénients déjà mentionnés en II.1.5.2, mais répond à notre souci de simplicité de réalisation, puisqu'il s'agit d'une maquette. Cette étape d'acquisition fera donc l'objet d'interactions entre expert et nous.

En fait il n'existe pas de méthodologie reconnue pour acquérir les connaissances ; chaque application donne lieu à une méthode particulière d'acquisition, dépendant du domaine d'expertise et de l'environnement dans lequel est réalisé le logiciel. Néanmoins, cette étape de l'acquisition s'avère généralement très laborieuse, et ceci d'autant plus que le domaine d'expertise sera plus complexe.

Enfin, le nombre et l'attitude des experts participant à l'expérience peut avoir des répercussions sur la difficulté de

l'acquisition des connaissances. Pour notre part, notre tâche a été facilitée par le fait de n'avoir affaire qu'avec un expert, de plus très coopérant et sans réticence psychologique face au projet ; cela nous a évité des problèmes de cohérence entre les diverses informations de la base, et de communication avec "la source d'expertise" : nous n'avons pas eu de barrage à franchir (si ce n'est techniquement) pour accéder à la connaissance qui devait alimenter notre système. Cela n'est pas toujours le cas, et mérite d'être mentionné.

III.2.1.2 - Méthodologie de l'acquisition

Nous explicitons le processus général ; puis nous ferons des remarques sur l'introduction des coefficients de réussite affectés aux règles inter-objet, et sur la qualité des résultats obtenus par cette façon de travailler.

III.2.1.2.1 - Méthode de transfert

Elle procède grossièrement en deux étapes :

- acquisition des connaissances de l'expert de façon assez naturelle, sous une forme libre mais ressemblant déjà à des règles de production.

- apport de compléments aux informations obtenues sous la forme exprimée ci-dessus, de façon à ce qu'elles soient prêtes à être exploitées par le système (au codage en langage interne près).

La première étape a commencé plutôt par un monologue de l'expert qui, passant en revue les principaux outils connus, exprimait naturellement leurs contextes d'application, en essayant d'être le plus clair possible (explication de vocabulaire, des grandes idées à acquérir pour certains types de problèmes, etc...) ; ce premier discours avait une forme entièrement libre. A partir de ces informations - ou plus exactement, de ce que nous en avons compris, nous avons rédigé quelque chose qui ressemblait déjà à des règles de production, en reformulant les connaissances dans des tableaux à trois colonnes (conditions d'application, problèmes concernés, outil adapté) en essayant d'éviter

les redites et en vérifiant que cela reprenait bien tout ce qu'avait dit l'expert. A partir de là a commencé une succession d'aller-retour entre l'expert et nous, l'expert vérifiait que le tableau interprétait correctement ce qu'il avait voulu dire, nuançant ou complétant parfois ses affirmations sous une forme libre ; à nous alors de mettre à jour le tableau de façon à tenir compte des remarques de l'expert, et de le lui renvoyer, et ainsi de suite. Au bout d'un certain nombre d'itérations de ce processus, nous avons considéré d'un commun accord que le produit obtenu pouvait servir de base pour la maquette, quitte à revenir éventuellement dessus par la suite (ce qui ne doit pas poser de difficulté avec une architecture de type système expert).

La seconde étape, beaucoup plus rapide et simple, n'a pas fait intervenir l'expert. Il s'agissait de compléter éventuellement les tableaux en explicitant mieux les relations entre problèmes traités et variables sur lesquelles portaient les conditions (ex : "si la matrice est symétrique" devient "[(problème = inversion de matrice, la matrice à inverser étant symétrique)] ou [(problème = résolution de système linéaire, la matrice du système étant symétrique)] ...") ; cela ne constitue en fait qu'un complément syntaxique. Cela revient à expliciter des informations souvent condensées ou implicites dans la façon naturelle de s'exprimer de l'expert, de façon à ce que l'ensemble des informations soit exploitable par le moteur d'inférences.

III.2.1.2.2 -Introduction des coefficients de réussite

La notion d'incertitude plus ou moins grande sur la qualité des résultats obtenus par l'application d'un outil sur un problème, selon les caractéristiques numériques de données, transparaissait dans tous les propos de l'expert ; mais elle restait très informelle, exprimée seulement par des termes qualitatifs.

Pour quand même la prendre en compte, nous avons demandé à l'expert d'essayer d'affecter ses règles par les coefficients mentionnés dans les spécifications en III.1.3.4, ce qui ne lui était pas du tout aisé. En faisant varier les contraintes d'application, il a alors tenté d'exprimer cette incertitude en affinant les règles, et en leur

affectant des coefficients. Ceux-ci varient de 0 à 100, et ne sont pas les résultats de calculs statistiques sur des cas de problèmes déjà rencontrés. Ces coefficients ont été attribués de façon totalement arbitraire, de façon à pouvoir refléter la réalité tant bien que mal ; l'expert se gardait le droit de les modifier au besoin de façon à ce que le système ne génère que les solutions attendues : il est inutile de livrer à l'usager une méthode qui peut convenir théoriquement, mais qui a toutes les chances de donner de mauvais résultats numériques, si de meilleures méthodes existent a priori.

III.2.1.2.3 - Remarques

A l'issue de cette étape, nous avons obtenu un ensemble de règles comportant souvent des redondances logiques, ou se recoupant parfois les unes les autres. Pour optimiser la tâche du futur moteur d'inférences, ces redondances logiques doivent être évitées. Un certain temps a été consacré à voir quelles règles pourraient être éventuellement simplifiées (voire supprimées) ; cette tâche fastidieuse gagnerait beaucoup à être automatisée, mais ce problème est encore mal résolu.

En conclusion, l'acquisition des connaissances a été relativement longue et laborieuse : pourtant, notre domaine d'expertise est très réduit, et nous avons bénéficié de conditions de travail particulièrement agréables (un expert très disponible et coopérant, et l'aide très efficace d'une personne possédant une formation en mathématiques pour la compréhension des propos de l'expert). Cette étape n'est donc pas négligeable, et le temps à y consacrer peut devenir très important dans le cadre d'une application plus grosse.

III.2.2 - Implémentation de la fonction résolution

Il s'agit en fait de l'interpréteur des règles (règles inter-objets, de comparaison entre objets de même type sur les contraintes sur résultats, de déduction entre propriétés etc...) permettant de faire une résolution. La stratégie de l'interpréteur a été décrite en III.1.3, il reste donc à voir comment elle a été codée en termes de prédicats. Nous expliciterons donc la façon dont les règles (au sens ci-dessus) ont été

codées ; puis nous verrons les graphes d'appel entre prédicats dans le cas des résolutions automatique et assistée ; enfin, nous verrons quelques limites du traitement des cas d'échecs tel qu'il a été implémenté.

III.2.2.1 - Implémentation des règles (version de base)

Nous simplifions ici la présentation de l'implémentation des règles en en donnant le principe de base ; en fait, cette implémentation est un peu plus complexe : la présentation en sera donc complétée lors de celle de l'implémentation de la fonction explications.

III.2.2.1.1 -Règles inter-objets

Les noms du ou des problèmes à résoudre ont une importance capitale pour ces règles ; en effet, ces noms se transforment au cours de la progression dans l'arbre de résolution, et sont symboliques : on peut les interpréter à tout moment pour savoir où l'on en est dans l'arbre de résolution.

Les règles sont codées en deux temps, ce qui se traduit par les deux prédicats CHOIX et CH. Dans tout ce qui suit, les symboles de variables sont précédés d'une étoile et nous utiliserons la syntaxe de PROLISP (/ZAN 84/), et le langage de la logique restreinte aux clauses de Horn (/GIA 85/).

Une règle inter-objets procède en deux temps, elle permet de :

1 - à partir du nom de problème relatif à l'objet A (et de ses pères dans l'arbre de résolution) qui a été retenu dans le chemin d'une solution éventuelle :

. de vérifier si l'objet B, fils de A, pourrait à priori, résoudre le problème de l'usager, sans tenir compte des contraintes sur les données,

. de connaître alors le nouveau nom du problème, tenant compte de B.

ceci étant l'objet du prédicat CHOIX

2 - de vérifier si les contraintes sur les données sont vérifiées pour que B soit applicable.

ceci étant l'objet du prédicat CH

ex :

```
;  CHOIX TYPE_METHODE

((CHOIX TM *lpb1 directe *coef *lpb2)
 (INCLUS *lpb1 (((resolsl)(*a *x *n)(resolsl d)(*a *x *n *b))
                ((invmat)(*a *n *c)(invmat d)(*a *n *c))
                ((caldet)(*a *n *d)(caldet d)(*a *n *d))
                ((decomp)(*a *n *e)(decomp d)(*a *n *e))) nil *lpb2)
 (CH TM directe (*a *n) *coef))

((CH TM directe (*a *n) 90)          ; littéral but
  (BANDE *a )                        ; suite de prémisses (conditions)
  (NON (DEFINIE_POSITIVE *a))
  (NON (ET ((SYMETRIQUE *a)(sup (le taux de remplissage de *a) 5 ))))
  (INF *n 50 )
  (BIENCOND *a )                     ; *a est bien conditionnée
  (INF (le taux de remplissage de *a) 10))
```

En réalité, plusieurs clauses ayant pour prédicat but CH peuvent être appelées par une clause de prédicat but CHOIX, correspondant à autant de règles inter-objets.

Commentaires

Le littéral (CHOIX TM *listeproblèmes1 directe *coef *listeproblèmes2) signifie que, pour résoudre l'ensemble des problèmes définis par *listeproblèmes1, l'outil de type Type-Méthode et de nom directe conviendrait, avec un pourcentage de réussite égal à *coef. La liste des problèmes serait alors renommée en *listeproblèmes2 (ce nom sert à mémoriser, en quelque sorte, le chemin déjà parcouru sur une

branche de l'arbre de résolution).

La clause de tête (CH TM directe (*a *n) 90) exprime que l'outil Type-Méthode peut être choisi avec un pourcentage de réussite égal à 90% si les conditions définies par le corps de la clause sur les variables *a et *n du problème à résoudre sont vérifiées.

III.2.2.1.2 - Règles de comparaison entre objets de même type

Ces règles sont codées selon le même schéma que les précédentes, le but étant de pouvoir exprimer des règles conditionnées par les valeurs de certaines variables (cf /SAU 84-2/, P. 48-49) ; pour cela nous devons expliciter la relation problème-variables. Ces règles n'ont pas de coefficient de validité. Nous utilisons donc deux prédicats pour les exprimer : BETTER, et MEILLEURQUE. C'est ici le prédicat MEILLEURQUE qui traduit les règles telles que les a exprimées l'expert, BETTER servant d'intermédiaire.

ex : Quelques règles de comparaison entre types de méthodes

```
((BETTER TM *cr*tm1*tm2*lpb)(inclus *lpb(((resolsl)(*a *x *n *b))
                                        ((invmat)(*a *n *c))
                                        ((caldet)(*a *n *d))
                                        ((decomp)(*a *n *e))) nil *s)
                            (MEILLEURQUE TM *cr *tm1 *tm2 (*a *n)))

              ;REGLES DE COMPARAISON ENTRE TYPES METHODES(*a *n)))

     ((MEILLEURQUE TM precision iterative_par_points direct (*a *n)))
     ((MEILLEURQUE TM precision composite directe (*a *n))
     ((MEILLEURQUE TM tempsexec directe composite (*a *n))
     ((MEILLEURQUE TM tempsexec composite iterative_par_points (*a *n))
```

Commentaire

La clause de tête (MEILLEURQUE TM précision itérative directe (*a *n)) exprime que le type de méthode itérative est meilleur au point

de vue précision que le type de méthode directe, si les conditions définies éventuellement par le corps de clause (ici vide) sur les variables *a et *n du problème à résoudre sont vérifiées.

Les clauses définissant le prédicat BETTER ne servent qu'à transmettre les noms des variables du problème à résoudre aux clauses définissant MEILLEURQUE.

III.2.2.1.3 - Règles de déduction de propriétés

Elles mettent en oeuvre des symboles de prédicats suffisamment significatifs.

ex : ((symetrique *a)(définie-positive *a))

ie : une matrice est symétrique si elle est définie-positive

III.2.2.1.4 - Prise en compte des outils de contrôle associés aux outils susceptibles de résoudre un problème

Lorsqu'une règle inter-objets s'applique accompagnée d'un conseil d'une opération à effectuer avant ou après l'application de la solution, le Système mémorise cette information, donc en cours de parcours de l'arbre de résolution. Lors de la récapitulation des solutions "finales" (feuilles de l'arbre de résolution), le Système cherche parmi ces informations mémorisées si elles s'appliquent à certaines de ces solutions - ie si certaines solutions ont été obtenues via des règles à conseil -, et dans le cas affirmatif, le Système l'indique automatiquement à l'usager.

```
ex :   ((CH TM directe (*a *n) 15) ; clause but - *n est l'ordre de la
                                     matrice *a
       (BANDE *a)                  ; clauses prémisses
       (NON (SYMETRIQUE *a))
       (SUP *n 500)
       (MALCOND *a)                ; (si *a est mal conditionnée)
       (INF (le taux de remplissage de *a) 2)
       (CHOIXAVANT directe "preconditionner la matrice"))
```

ie : Le type de méthode DIRECTE s'applique dans ces conditions avec un coefficient de validité de 15% à condition d'effectuer auparavant un préconditionnement de la matrice.

Des règles supplémentaires permettent au Système de préciser quelles méthodes de préconditionnement s'appliquent sur une méthode de type direct (implémentées via le prédicat ADAPTAVEC).

Le prédicat CHOIXAVANT concerne les outils de contrôle à appliquer sur les données du problème avant de le résoudre avec l'outil préconisé par la règle. Le prédicat CHOIXAPRES concerne les outils de contrôle à appliquer aux résultats obtenus après application de l'outil proposé par la règle, pour vérifier leur qualité numérique.

III.2.2.2 - Version simplifiée de l'interpréteur de PELAGIE

Les critères sur performances de l'usager sont supposés ordonnés par priorité décroissante ; soit liste-critères la liste $(cr_1 \ldots cr_n)$.

De façon interne, l'interpréteur manipule une information qui évolue au cours de la résolution. Cette information, à un instant t_0, contient le(s) problème(s) à résoudre et une arborescence d'outils de plus en plus concrets (ex: type de méthode direct, méthode de décomposition symétrique) convenables d'après le système pour résoudre le problème général de l'usager. A chaque étape où le système "précise" comment peut être résolu ce problème (ex: algorithme LDL^T), l'information évolue de façon à prendre en compte ce nouveau point. Dans notre jargon, nous disons que nous "rebaptisons ce problème".

Remarque : nous avons justifié cela en III.1.2.3, d.

En ne faisant apparaître que les prédicats correspondant aux principales règles exprimées par l'expert, nous donnons ci-dessous une version très simplifiée de l'interpréteur de PELAGIE exprimée en pseudo-LISP.

Les outils que propose le système pour résoudre un problème de l'usager sont donnés par la fonction "descendants". L'algorithme ci-dessous correspond à un cycle de la recherche, i.e. à un type d'outil précis (ex: algorithme), à partir d'un outil déjà sélectionné de type plus abstrait (ex: méthode).

descendants (0) := meilleurs-descendants(sélection-critères-mathématiques(0))
où
sélection-critères-mathématiques(0) = ensemble des x tels que la clause dont le littéral de tête est (CHOIX pb,...,x,pbx,coef-x) soit vraie.

(notations : pb est l'information d'entrée indiquant le problème en cours de résolution ;
x est un outil concrétisant 0, affecté du coefficient coef-x (indiquant une estimation de réussite selon les contraintes sur les données numériques)
pbx est l'information indiquant le problème "rebaptisé", une fois que l'outil x est retenu par le système)

meilleurs-descendants (F)

C:= meilleurs-coefficients (F, $\max_{g \in F} \frac{\text{coef-g}}{3}$)

M:= meilleurs-critères-performances (C, liste-critères)

S:= $\bigcup_{m \in M}$ descendants (m)

si S = ∅ alors meilleurs-descendants (F):= meilleurs-descendants (F-M)

sinon meilleurs-descendants (F):= S

meilleurs-coefficients (F,coef-limite)

f:= car(F)
si coef-f ≥ coef-limite alors
meilleurs-coefficients (F,coef-limite):=
cons(f,meilleurs-coefficients (cdr(F),coef-limite))

sinon meilleurs-coefficients (F,coef-limite):=
meilleurs-coefficients (cdr(F), coef-limite)

meilleurs-critère-performances (C,liste-critères)

si liste-critères ≠ ∅ alors
| cr_1:= car(liste-critères)
| c_1:= ensemble des x tels que x ∈ C et ∃ y ∈ C
| tel que une règle dont le premier littéral est
| (MEILLEURQUE cr_1,x,y,...) soit vraie
| meilleurs-critères-performances (C,liste-critères)
| := meilleurs-critères-performances (C_1,(cdr liste-critères))
| finsi

sinon meilleurs-critères-performances (C,liste-critères):= C

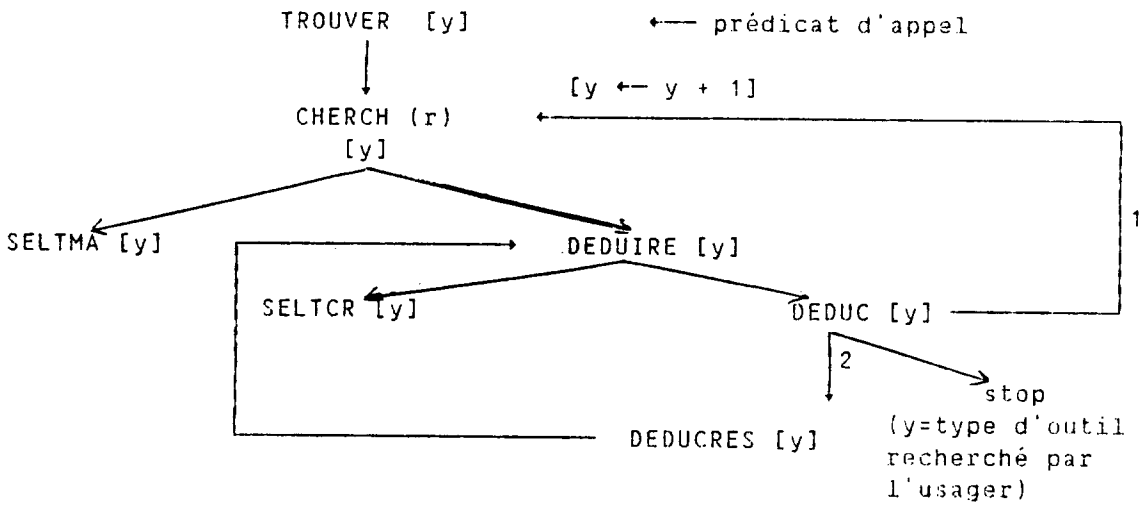
III.2.2.3 - Implémentation de la résolution de problème

III.2.2.3.1 - Fonction de recherche automatique d'un outil pour résoudre un problème

Notons [y] le niveau courant dans l'arbre de résolution où en est la recherche en cours (y = type de méthode, ou méthode, etc...) ; si une méthode vient d'être retenue, y = A(lgorithme) ; c'est en fait le type des outils qui seront sélectionnés lors de la prochaine étape de résolution.

Les noms de prédicats encadrés dans les schémas ci-dessous sont relatifs aux règles de la base de connaissances significatives pour l'usager.

Schéma général

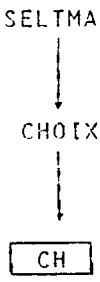


Signification mnémonique des symboles de prédicats

SELTMA	Sélection sur critères mathématiques	(ex: d'une matrice)
SELTCR	Sélection sur critères usagers	(ex: précision...)
DEDUCRES	Déduction à partir des résultats	(obtenus par SELTMA)
DEDUC	Déduction	

Comme exposé en III.1.3.5.1, c'est une recherche par cycle (un cycle correspondant à un ensemble d'outils d'un type donné). C'est ce qu'exprime la flèche 2. Décrivons maintenant un peu chacun des prédicats apparaissant dans ce schéma, ou plutôt les fonctions qu'ils représentent sémantiquement (description en termes procéduraux).

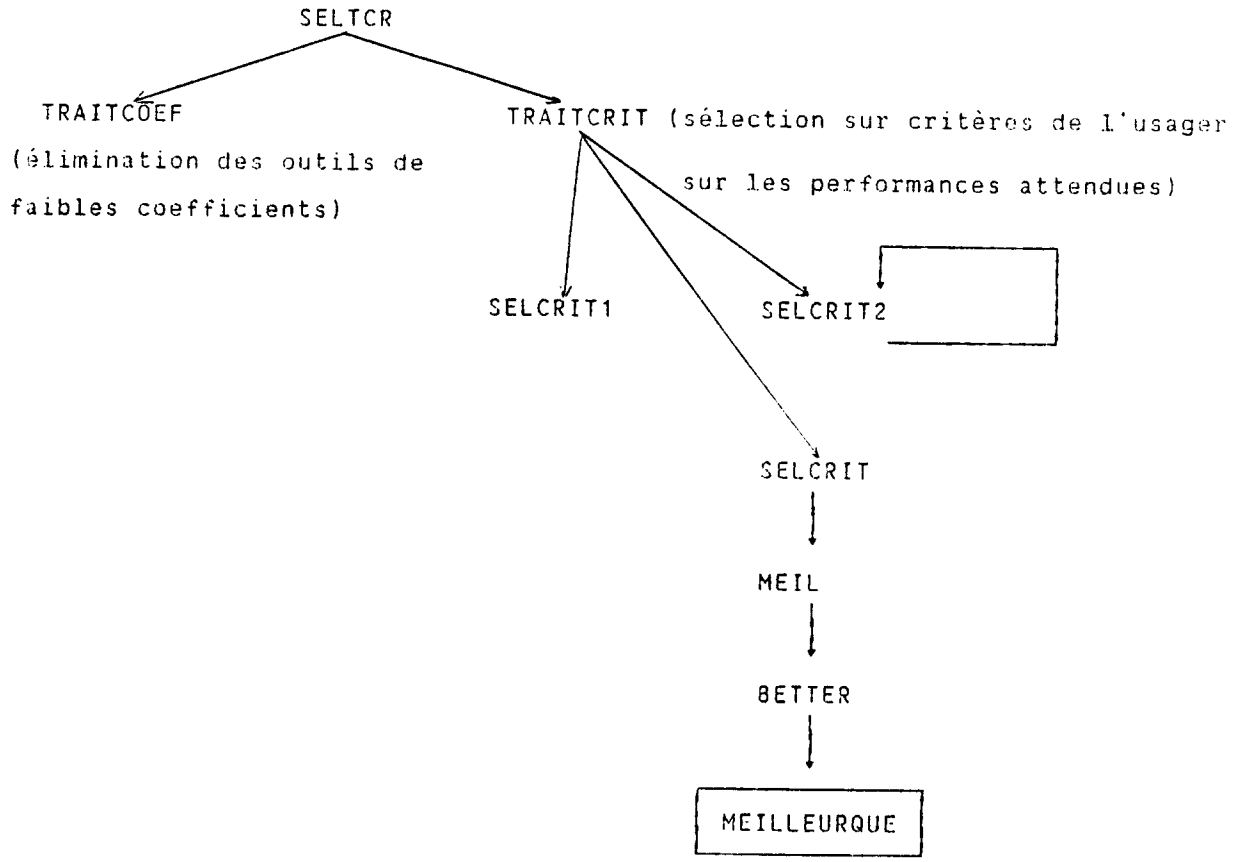
SELTMA opère la sélection des outils de niveau [y] descendant d'un même antécédant retenu au niveau [y-1], d'après les contraintes sur les données du problème. Nous avons :



DEDUIRE déduit, à partir des objets de type [y] retenus en appliquant SELTMA, toutes les solutions du type désiré par l'usager (niveau feuilles de l'arbre de résolution) et répondant à toutes les contraintes de l'usager. L'appel à ce prédicat permet donc,au niveau [y],d'éliminer toutes les propositions retenues par SELTMA avec un trop faible coefficient, et opère une sélection sur les contraintes de performances (SELTCR), puis cherche tous les outils de niveau plus concret auxquels cette sélection peut faire aboutir - donc à partir du niveau [y+1] dans l'arbre de résolution - (DEDUC,1). Dans le cas où cette recherche n'aboutit à rien, le système revient sur la sélection sur coefficients et contraintes de performances au niveau [y] et recherche s'il est possible d'aboutir à des outils du type désiré par l'usager à partir de cette nouvelle sous-sélection... jusqu'à arriver à un ensemble vide après un appel à SELTCR.

SELTCR, quand il est activé, opère à partir d'un ensemble donné d'outils, une première sélection en éliminant les outils affectés d'un coefficient de réussite trop faible (1/3 du max des coefficients - ce rapport étant choisi arbitrairement, de façon à donner les résultats voulus) ; puis il effectue à partir de là une seconde sélection en ne gardant que les outils vérifiant au mieux les contraintes sur performances exprimées par l'usager.

Voici ci-dessous l'arbre des principaux prédicats appelés par SELTCR :



Signification mnémonique des symboles de prédicats

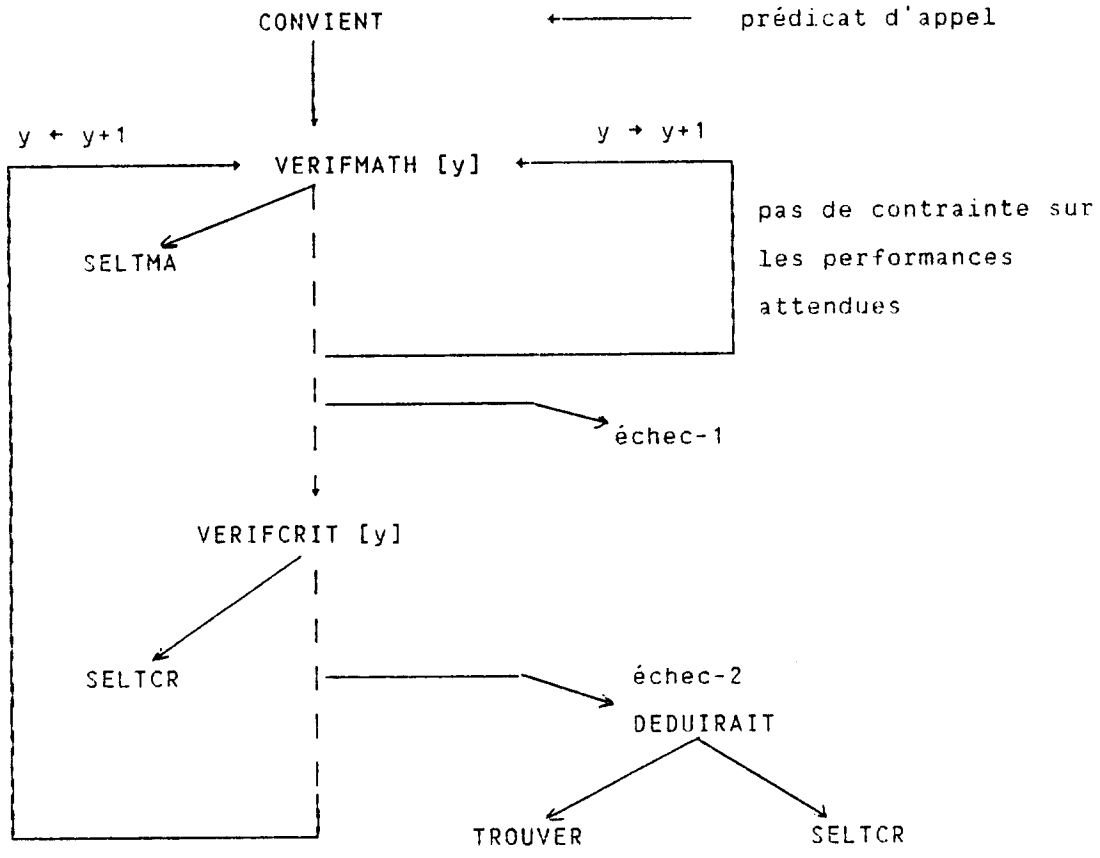
SELTCR	Sélection des meilleurs outils selon les critères de l'usager sur les performances attendues
TRAITCOEF	Traitement sur les coefficients affectés à des outils sélectionnés de même type
TRAITCRIT	Traitement des critères usagers sur les performances attendues
SELCRIT1	Sélection sur le 1er critère usager
SELCRIT2	Sélection sur les critères suivants (dans une liste de critères)
SELCRIT	Sélection sur un critère usager
MEIL	meilleur (selon un critère, argument du prédicat) (comparaison de deux outils de même type)
BETTER	meilleur, en anglais
MEILLEURQUE	meilleur que (prédicat introduisant une règle de l'expert)

III.2.2.3.2 - Fonction de vérification qu'un outil O de type T est adapté à la résolution d'un problème donné

C'est la première fonction assurée à l'occasion d'une résolution assistée par l'usager (cf.III.1.3.5.2). Pour la suite de la résolution, on se ramène au cas précédent.

Nous conservons les notations introduites dans les schémas de III.2.2.3.1.

Version simplifiée du graphe d'appel général



Signification mnémonique des symboles de prédicats

VERIFMATH [y] Vérification qu'un outil o de type y satisfait aux contraintes mathématiques sur les variables, puis aux autres contraintes du problème, de sorte que l'outil

préconisé par l'usager (dont o est un ancêtre) réponde au problème.

VERIFCRIT [y] Vérification qu'un outil o de type y satisfait aux critères sur les performances attendues, de sorte que l'outil préconisé par l'usager (dont o est un ancêtre) réponde au problème.

SELTMA,SELTCR déjà décrits auparavant

DEDUIRAIT essaie de déduire si un objet o de type y, répondant aux contraintes d'un problème, possède un descendant, outil du type préconisé par l'usager, qui répondrait à son problème.

Quelques explications supplémentaires

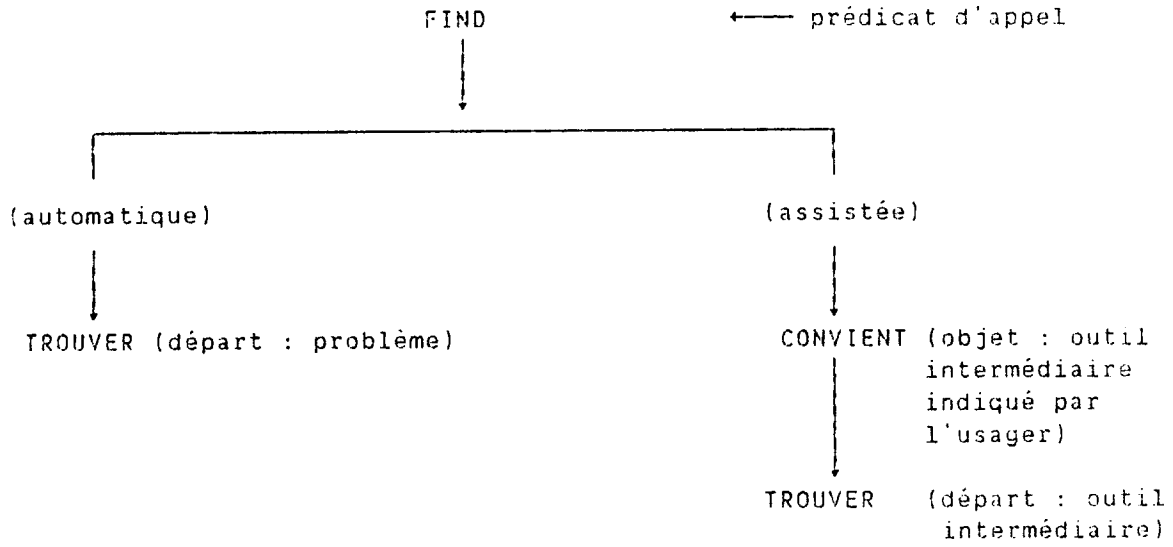
Etant donné un outil O, le système peut retrouver ses ancêtres de chaque type : si l'usager préconise un algorithme A, on suppose donc connus la méthode et le type de méthode associés.

Le schéma précédent représente un cycle de vérification que chaque ancêtre de o de type y, aurait effectivement été sélectionné par le système pour résoudre le problème de l'usager.

Il existe deux raisons possibles d'arrêt (ou d'échec) au cours d'un cycle de vérification. Dans un premier cas, l'outil o de type y ne satisfait pas aux contraintes mathématiques sur les variables du problème (échec-1) ; dans un second cas, cet outil n'est pas le meilleur vis-à-vis des critères sur les performances : temps exécution, précision, ou/et place mémoire occupée (échec-2). Dans ce dernier cas, le système va vérifier s'il connait effectivement des outils de type de celui préconisé initialement par l'usager, issus des meilleurs outils de type y répondant au problème (DEDUIRAIT).

III.2.2.3.3 - Stratégie générale de résolution

Nous avons donc implémenté deux modes de résolution, en utilisant les prédicats selon le schéma ci-dessous :



III.2.2.3.4 - Limites du traitement des échecs

Il est nécessaire de faire au moins deux remarques.

La première concerne la politique générale de la stratégie. Chaque fois que le système retient une voie prometteuse (ex : une méthode), il essaiera de faire aboutir cette voie coûte que coûte, quitte à choisir des outils du type cherché de qualité médiocre due à l'état de la base de connaissances. En fin de raisonnement du système, nous risquons donc, dans des cas extrêmes, de nous voir proposer des outils ne mettant pas en oeuvre les mêmes principes, et de qualités inégales (même si les qualités des principes étaient analogues). Ceci est un peu regrettable ; heureusement, les coefficients de réussite associés à ces outils, lors de leur présentation à l'usager, peuvent l'inciter à la méfiance.

La seconde remarque concerne les contenus des règles, et la façon de les exploiter en cas d'échec. PROLOG étant sous-jacent à notre module de résolution, nous ne voulons pas utiliser de prédicat prédéfini qui rendrait le contrôle trop déterministe, et donnerait de l'importance à l'ordre des règles dans la base de connaissances. Nous précisons donc bien les conditions d'application d'une règle inter-objet, en employant éventuellement des artifices pour qu'elle ne soit appliquée que quand on le désire. Toutefois, cela présente un inconvénient dans le cas suivant :

ex : R_1 *"si la matrice est définie-positive, alors choisir TM_0"*

R_2 *"si la matrice est symétrique et non définie-positive, alors..."*

Supposons que nous ayons à résoudre un problème portant sur une matrice définie-positive. Pour exploiter au mieux cette particularité, le système appliquera les règles de type R_1 et pas celles de type R_2, selon leurs prémisses. Mais cette façon d'écrire les règles est à double tranchant dans le cas d'une base de connaissances incomplète ; supposons que le système ne connaisse pas d'algorithme issu de TM_0 : cette voie, lancée par R_1, tombe en échec. R_2 n'avait pas pu être appliquée, donc le système ne se dirigera pas vers un outil de résolution n'exploitant pas la contrainte "définie-positive" pourtant la matrice du problème est symétrique.

Nous voyons apparaître là l'utilité d'un module d'apprentissage, qui saurait déceler où la base de connaissances est incomplète.

III.2.3 - Implémentation des explications

Les connaissances sont exprimées en logique des prédicats, et, plus particulièrement, le logiciel est réalisé en PROLOG ; nous avons donc implémenté la fonction de génération d'explications selon les idées développées par Clark et MacCabe (/CLA 82/, cf. seconde partie, I.2.1). Nous allons donc voir comment nous avons appliqué cette méthode successivement aux explications de type comment, pourquoi, et (indirectement) pourquoi-pas.

III.2.3.1 - Comment ?

La méthode consiste à rajouter un terme servant de trace uniquement dans les littéraux qui peuvent être assez significatifs pour figurer sous une forme adaptée dans les explications présentées à l'utilisateur. En effet, si l'on traçait systématiquement tous les littéraux effacés lors de la résolution d'un problème, la trace obtenue contiendrait beaucoup de bruit (dû aux nécessités de l'opération de

cours d'une résolution, selon le principe de résolution de PROLOG. Ce

R_4 - ((TOTO *a (toto est vrai sur *a car *j)) (BANDE *a *j)) ;

Le prédicat DON caractérise les faits explicités par l'usager, caractérisant les données du problème. Dans le contexte d'utilisation de ces règles, la variable *a représente une matrice. Les termes soulignés, une fois instanciés lors de l'effacement des règles les contenant, permettent de générer les explications par empilements successifs.

Par exemple, pour effacer R_4, il suffit d'effacer R_3 en effaçant R_1 ou R_2. Supposons que A soit le nom de la matrice du problème considéré. Si le processus d'effacement est (R_1, R_3, R_4), l'explication générée lors de l'effacement de R_4 sera "toto est vrai sur A car A est bande". Si c'est (R_2, R_3, R_4), l'usager a spécifié que A était tridiagonale, et l'explication résultante au niveau de R_4 est : "toto est vrai sur A car A est bande puisque A est tridiagonale".

Les explications obtenues pêchent parfois par la lourdeur de l'expression, mais renseignent assez clairement l'utilisateur sur le raisonnement suivi.

b - *exemple sur une règle inter-objets*

Le codage initial de la règle (cf. III.2.2.1.1) s'alourdit avec l'introduction des termes d'explication-comment (entourés) :

```
((CHOIX *num TM *lpb1 directe *coef *lpb2 *j)
 (INCLUS *lpb1 (((resolsl)(*a *x *n *b)(resolsl d)(*a *x *n *b))
                ((invmat)(*a *n *c)(invmat d)(*a *n *c))
                ((caldet)(*a *n *d)(caldet d)(*a *n *d))
                ((decomp)(*a *n *e)(decomp d)(*a *n *e))) nil *lpb2)
(CH *num TM directe (*a *n) *coef *j))
```

```
((CH   TM directe (*a *n) 90  *jtm)
 (PORTE *a "la matrice")(PORTE *n "l'ordre de la matrice")
 (BANDE *a  *jb)
 (NON (DEFINIE-POSITIVE *a  *jdf))
 (NON (ET ((SYMETRIQUE *a *js)(SUP (le taux de remplissage de *a) 5 *jf))))
 (INF *n 50  *jis)
 (BIENCOND *a  *jcond)
 (INF (le taux de remplissage de *a) 10  *jis1)
 (ID *jtm (*jb et non definie positive et non [symetrique avec son taux
      de remplissage>5] et *jis et jcond et *jis1)))
```

Evidemment, la forme "en français" des termes explicatifs pourrait être améliorée avec un peu de travail de finition supplémentaire.

c - *Exemple sur une règle de comparaison entre objets de même nature sur des contraintes de performances (cf. III.2.2.1.2)*

```
((BETTER TM ... *lpb *j)(INCLUS ....)
                         (MEILLEURQUE TM *cr *tm1 *tm2(*a *n) *j))
((MEILLEURQUE TM precision composite directe (*a *n)
       (composite est meilleure que directe au point de vue précision)))
```

Dans ce cas particulier le terme explicatif est une liste de constantes (il n'y a pas de condition sur les variables *a et *n du problème) ; la clause ((MEILLEURQUE...)) est une assertion.

III.2.3.2 - Pourquoi ?

Le principe est similaire à celui pour les explications de type comment ; mais ici seuls les littéraux dont l'effacement nécessite une intervention de l'usager comporteront un argument trace supplémentaire, dont le contenu est constitué avec le même esprit que précédemment.

Toutefois, la forme de cet argument trace est particulière, de façon à ce que le système puisse répondre aux questions pourquoi ? comme spécifié en III.1.5.2.3. Cet argument est composé de deux blocs. Le premier est formé d'une part d'un identificateur de la clause ou règle

ayant comme prémisse le littéral faisant l'objet du pourquoi ?, d'autre part, des premisses déjà effacés avant ce littéral. Le second est le contexte en cours de résolution, provoquant l'effacement de la règle ayant comme prémisse l'objet du pourquoi ? - ce contexte est lui aussi constitué comme l'argument trace décrit. On appelle le prédicat associé à un littéral qui ne peut être effacé que grâce à un dialogue avec l'usager : "prédicat demandable". L'interface usager (ou acquisition des contraintes sur les données) se fait grâce à un "prédicat d'interface" nommé "demander".

a - *Exemple d'application*

Cet exemple simplifié permet de saisir le principe de fonctionnement de ces arguments d'explication. Ceux-ci sont soulignés dans les notations ci-dessous ; les prédicats supposés demandables sont encadrés.

R_0 : ((APPLICABLE M_0 (*a *n) *cntxo)
(RETENUE TM-directe (*a *n))
(SYMETRIQUE *a *n)
(INTERESSANTE (*a *n) ((R_0.((le type de méthode direct a été retenu)
(*a est symétrique))).*cntxo)))

R_1 : ((INTERESSANTE (*a *n) *cntx1)
(INFERIEUR *n 50)
(BANDE *a *n ((R_1.((*n est inférieur à 50))).*cntx1)))

Pour effacer R_0, il faut effacer la prémisse "INTERESSANTE...", elle-même effaçable via R_1 ; on doit effacer la clause "BANDE...", BANDE étant demandable. Le prédicat d'interface DEMANDER traite cet interface avec l'usager, et a comme argument, outre la question à poser l'usager, le terme explicatif au pourquoi ? apparaissant dans le littéral (BANDE...) ; il permet d'effacer "BANDE...".

Par construction, les réponses possibles de l'usager au prédicat DEMANDER sont :

- oui (ex: la matrice A est bande) ou non, auquel cas le système

insère le fait correspondant dans la base de clauses

- pourquoi ? (ex: pourquoi le système veut-il savoir si la matrice A est bande ?, que cherche-t-il à montrer ainsi ?),

  l'usager pouvant demander plusieurs fois de suite pourquoi ? (pour comprendre toutes les étapes du raisonnement en cours) avant de répondre oui ou non à la question du système.

Au moment où ce prédicat d'interface doit être effacé, si le problème à résoudre porte sur une matrice A d'ordre N, le terme permettant de répondre à la question "pourquoi le système cherche à savoir si A est bande ?" est instancié par :

$((R_1.((N \text{ est inférieur à } 50))).((R_0.((\text{le type de méthode direct à été retenu})(A \text{ est symétrique}))).contexte)))$,

"contexte" désignant le contexte d'appel de R_0.

Si l'usager pose une première question de type pourquoi ? portant sur la spécification de la contrainte "matrice bande", la réponse fait état de la première partie du terme explicatif ; son premier élément désigne la règle que le système essaie d'effacer (R_1), les éléments suivants désignent les prémisses déjà effacées (ici : N est inférieur à 50).

L'usager peut alors redemander pourquoi ? (le système cherche-t-il à effacer R_1) ; on réitère alors le processus de réponse sur la partie restante du terme explicatif : le système cherche donc à effacer R_0, ayant déjà montré qu'il a retenu le type de méthode direct et que la matrice A est symétrique. La partie restante du terme explicatif est liée au contexte, et l'usager peut reposer autant de questions pourquoi ? que nécessaire, jusqu'à connaître tout le raisonnement en cours du système.

b - Conséquences sur l'implémentation des règles

Pour identifier les règles dans les explications de type pourquoi ?, il serait généralement trop lourd d'expliciter le contenu

des règles ; nous avons donc été amenés à les numéroter, ceci permettant d'ailleurs à l'expert de repérer plus rapidement une règle cherchée dans la base de connaissances. Ce numéro est codé comme un argument supplémentaire pour les prédicats CH et MEILLEUROUE ; pour ce dernier, le numéro n'offre qu'une commodité pour la mise-à-jour de la base de connaissances, puisqu'il ne fait pas l'objet de questions de type pourquoi ?. Par contre, quoique les règles de déduction de propriétés puissent faire l'objet de pourquoi ?, nous avons préféré les désigner par leur contenu car elles sont courtes et ne feront pas l'objet de mise-à-jour ; toutefois, on pourrait les numéroter par souci d'homogénéité.

En fin de compte, le code des règles est assez volumineux à cause des deux arguments d'explication (comment et pourquoi), qui deviennent assez importants lorsqu'une règle a beaucoup de prémisses - surtout pour les règles inter-objets.

```
ex :
((CH 1 TM directe (*a *n) 50) *jtm *t)
 (PORTE *a "la matrice")
 (PORTE *n "l'ordre de la matrice")
 (BANDE *a  *jb [((1) *t)] )
 (NON (DEFINIE_POSITIVE *a  *jdf [((1 *jb) *t)] )
 (NON (ET ((SYMETRIQUE *a  *js [((1 *jb (*a est non definie-positive)) *t)] )
          (SUP (le taux de remplissage de *a)          5      [*jf]
               [((1 *jb (*a est non définie_positive) *js) *t)] ))))
 (INF *n          50     *jis
      [((1 *jb
          (*a est non definie_positive)
          (faux (*a soit symetrique)
                ((le taux de remplissage de *a)soit superieur a 5)))
          *t] )
 (BIENCOND *a          *jcond
           [((1 *jb
               (*a est non definie_positive)
               (faux (*a soit symetrique)
                     ((le taux de remplissage de *a) soit superieur a 5))
               *jiq)
           *t)] )
 (INF (le taux de remplissage de *a)          10          *jis1
      [((1 *jb
          (*a est non definie_positive
          (faux (*a soit symetrique)
                ((le taux de remplissage de *a) soit superieur a 5))
          *jis
          *jcond)
      *t)] )
 (IO  *jtn
      (*jb et      non      definie    positive    et      non      [
           symetrique      avec       son         taux    de
           remplissage     >5         ]           et      *jis     et
           *jcond    et    *jis)))
```

Remarque :

les prémisses du prédicat PORTE ont été introduites pour des besoins utilitaires annexes (cf.III.2.4) ; les arguments entourés permettent de répondre au comment ?, les encadrés sont destinés à la réponse au pourquoi ?

Heureusement, le rajout de ces termes explications est aisément automatisable (c'est d'ailleurs ce qui a été fait) !

III.2.3.3 - Pourquoi pas ?

Par construction, les explications de ce type sont tout à fait semblables à celles de type comment : il s'agit de ne présenter que les explications concernant dans l'arbre de résolution, l'endroit où l'orientation du raisonnement effectivement suivi par le système devient contradictoire avec celle qui aurait mené à l'outil préconisé par l'usager.

Dans le cas où le désaccord est dû à de mauvais coefficients de réussite ou aux contraintes sur performances, le système explique pourquoi il retient un certain sous-ensemble d'outils par rapport à un ensemble initial ; un "antécédent" de l'outil préconisé par l'usager appartenant à l'ensemble mais pas au sous-ensemble, l'usager comprend (par déduction) pourquoi son outil n'a pas été retenu par le système.

Par contre, dans le cas où un "antécédent" de l'outil faisant l'objet du pourquoi-pas n'est pas retenu par le système après une sélection sur règles inter-objets, le système ne fait qu'expliquer quelles règles il a pu appliquer pour réaliser cette sélection. Mais il ne dit pas :

- si l'antécédent n'existe pas dans la base de connaissances (auquel cas il n'est mentionné dans aucune règle inter-objets)
- ou s'il existe des règles inter-objets permettant de conclure sur cet outil, et quelles prémisses n'ont pas pu en être effacées.

Les explications générées sont donc souvent incomplètes dans un tiers des cas.

Nous sommes bien conscients que cette approche n'est qu'un essai de résolution partielle du problème des explications négatives, abordé de manière tout à fait détournée. Mais elle donne quand même des indications intéressantes pour celui qui cherche à vérifier l'état de la

base de connaissances du système.

III.2.4 - Utilitaires annexes

Il s'agit, à ce stade de la maquette, de petits utilitaires pour l'usager, mais qui pourraient également servir au gestionnaire de la base de connaissances. Les principales fonctions utilitaires concernent la visualisation d'une règle en français, la possibilité de conserver les caractéristiques de certaines données de problèmes entre deux résolutions successives, et quelques vérifications élémentaires de cohérence de l'expression d'un problème.

III.2.4.1 - Visualisation d'une règle

Cela consiste à donner une représentation d'une règle, désignée par son numéro, en langage pseudo-français, compréhensible par l'usager. Contrairement à la présentation des règles dans les explications de type comment, ici elles sont explicitées hors contexte d'application.

La technique est simple, et est liée au langage d'implémentation. Une clause PROLISP étant une liste LISP, nous considérons cette liste de façon isolée ; son premier élément est la clause but, ses autres éléments sont les prémisses. Nous analysons syntaxiquement les éléments de la liste, en associant à chaque symbole de prédicat la forme correspondante en français. L'interprétation des symboles de variables a été résolue en introduisant dans les règles susceptibles d'être "traduites" (inter-objet, comparaison sur contraintes de performances) une correspondance entre un symbole de variable et une expression indiquant ce qu'elle représente dans un contexte de résolution ; ceci est l'objet du prédicat PORTE (cf.exemple de code d'une règle inter-objet, en III.2.3.2).

Les phrases générées sont compréhensibles, à quelques fautes de français près : la syntaxe LISP est fortement reconnaissable dans les traductions obtenues (surtout avec les prédicats NON et ET).

Cette fonction de visualisation est accessible lorsque le

système interroge l'usager pour savoir si une contrainte sur une donnée de problème est vraie. Elle est nécessaire ici, puisque les explications de type pourquoi font références aux règles par leur numéro. Mais elle pourrait être accesible de façon indépendante.

III.2.4.2 - Modification des données entre deux résolutions successives

Il est pratique pour l'usager de pouvoir résoudre un même problème type plusieurs fois consécutives, en ne faisant varier que quelques contraintes sur les données ; l'idéal est alors d'éviter d'avoir tout à redéfinir à chaque résolution.

Dans ce but, une fonction permet pour l'instant (au niveau maquette) de conserver d'une résolution à l'autre toutes les informations sur certaines données, ou de supprimer toutes les informations sur certaines données ; cela suppose que l'usager garde les mêmes noms pour les données inchangées.

Les faits connus sur une donnée de problème (contraintes vérifiées et non vérifiées) sont inscrits dans la base de faits via les prédicats DON (A est bande) et PASDON (A n'est pas tridiagonale).

En réalité cette fonction demande à être plus souple : elle devrait permettre de modifier seulement certaines des informations sur une donnée, ce qui est facile à réaliser.

III.2.4.3 - Quelques vérifications de cohérence

Ces vérifications sont élémentaires au niveau de la maquette. Elles permettent de donner quelques garde-fous à l'usager lors de la spécification d'un problème.

Lorsque l'usager définit son problème ont lieu les vérifications suivantes :

- deux variables d'un même problème ne peuvent avoir le même nom,

- si l'usager veut résoudre plusieurs problèmes à la fois, ils doivent porter sur des variables communes,

- les types de problèmes à résoudre doivent être connus du système,

- les contraintes sur les performances ne peuvent être que précision, place mémoire et temps d'exécution (le système ne sait pas se débrouiller avec une autre contrainte),

- lorsque l'usager spécifie une voie pour résoudre un problème, le type de l'outil préconisé doit être plus abstrait que le type de l'outil cherché.

Ce ne sont en fait que des vérifications d'ordre syntaxique. Des vérifications d'ordre logique ou sémantique demanderaient beaucoup plus d'efforts mais seraient d'un grand intérêt. Nous avons évoqué cette voie de travail en III.1.5.1.2.

III.2.5 - Caractéristiques physiques de la maquette PELAGIE

Dans ce paragraphe, nous donnerons les caractéristiques du matériel sur lequel la maquette a été implémentée ; puis nous verrons comment est structuré le logiciel, et quelles performances il nous a permis d'atteindre. Enfin, de façon pratique, nous donnerons rapidement quelques indications pour l'utilisation de PELAGIE.

III.2.5.1 - Matériel utilisé

La maquette a été implémentée sur la configuration DPS 68/MULTICS disponible alors au CERT, MULTICS étant un système temps partagé dans l'environnement MACLISP+PROLISP (interpréteur de PROLOG en LISP /ZAN 84/).

III.2.5.2 - Organisation physique du logiciel PELAGIE

III.2.5.2.1 - Répartition des codes LISP et PROLISP dans PELAGIE

Nous disposons de LISP et de PROLISP, et de la possibilité d'atteindre l'un à travers l'autre sans aucune difficulté (fonctions EVAL et PROVE de PROLISP). Toutefois, alors que MACLISP a un compilateur pour le code LISP, PROLISP fonctionne en mode interprété et ne se compile pas. Nous avons donc écrit certaines parties de PELAGIE en PROLISP, et d'autres en LISP pour bénéficier des avantages dûs à la compilation.

Tout ce qui a trait aux bases de connaissances ou de faits, et le moteur d'inférences (ou stratégies d'exploitation des règles), ont été codés en PROLISP, comme décrit jusqu'ici.

Par contre, toutes les fonctions réalisant :

- du calcul sur les coefficients de réussite,
- l'interface système/utilisateurs (définition complète d'un problème et présentation de ses solutions, visualisation des règles, etc...) sont codées en LISP, et sont compilées.

Signalons de plus que au niveau de cette réalisation de maquette, le code n'a pas toujours fait l'objet d'optimisations.

III.2.5.2.2 - Structure dynamique du logiciel

Lorsque la base de connaissances devient assez importante, la tâche du moteur d'inférences devient plus laborieuse vu le nombre de règles qu'il doit manipuler. Or, au cours de la résolution d'un problème, à un instant donné, le système ne s'intéresse qu'aux informations concernant les outils d'un type donné. Une idée consiste donc à typer les règles correspondantes (en leur ajoutant un argument spécifiant ce type, dans la clause positive ou but), et de répartir les règles dans des fichiers différents, par type. Ainsi il suffit, à un

instant donné de la résolution, de ne charger que le fichier correspondant au niveau de résolution courant ; le système travaille alors sur une base de règles réduite, donc de façon plus efficace ; d'autre part, cela évite d'avoir des problèmes de place mémoire. En contrepartie, les temps de chargement des fichiers ne doivent pas devenir trop importants, sinon ils annulent le gain de temps obtenu par la réduction de la masse d'informations contenue en mémoire centrale.

Une phase intermédiaire de l'implémentation a donc consisté à appliquer le chargement dynamique à toutes les sortes d'informations de la base de connaissances ; lorsque le nombre de règles inter-objets et de comparaison sur contraintes sur les performances a atteint environ 350, les temps de chargement des fichiers contenant ces règles est devenu trop important et a fait chuter les performances.

La solution finalement adoptée consiste à garder les règles de la base de connaissances constamment en mémoire centrale ; par contre, les connaissances de type objet sont chargées dynamiquement, selon les besoins du processus de résolution de problème.

Pour que cette gestion de fichiers puisse être gérée par le système de façon intéressante, on peut adopter des noms de fichiers "significatifs" ; dans PELAGIE, ils sont la concaténation de trois codes : l'un désigne le domaine concerné (ex : algèbre linéaire), l'autre la nature des informations contenues (ex : objet), et le dernier le type des outils concernés par le contenu du fichier (ex : algorithme).

III.2.5.3 - Performances réalisées par PELAGIE

PELAGIE est actuellement un logiciel comprenant environ 5000 lignes de code ; le nombre de règles numérotées s'élève aux alentours de 400 (règles inter-objets, règles de comparaisons entre objets de même nature sur contraintes de performances), ce qui permet de se faire déjà une bonne idée des possibilités d'un système réalisé en grandeur nature.

Elle présente certains défauts d'optimisation à différents niveaux. La forme logique des règles n'est sûrement pas encore optimale,

et peut-être reste-t-il quelques redondances logiques. D'autre part, nous n'avons pas consacré beaucoup de temps à essayer d'optimiser le code lui-même (en particulier : code des fonctions LISP, passages de PROLISP à LISP, le prédicat EVAL étant assez coûteux ...). Cela ne peut que nuire aux performances de la maquette.

Un compromis a été adopté quant à la structure plus ou moins dynamique du logiciel, mais il peut être remis en question au fur et à mesure de l'évolution de la base de connaissances.

En réalité le nombre d'inférences faites au cours d'une résolution est très difficile à mesurer ; ce sont elles qui donneraient une mesure exacte des performances de PELAGIE.

Nous ne savons que mesurer les temps de calcul pris par des résolutions de problèmes. Ces temps de calcul ne sont pas les temps de réponse à l'écran, ceux-ci dépendant de l'affluence des utilisateurs de MULTICS (et pouvant donc varier, pour un même problème, de la minute à la demi-heure). La figure 7 permet d'apprécier les performances sur les temps de calcul pour les exemples de résolution figurant en annexe ; ils sont tout à fait acceptables, surtout si l'on pense que la partie écrite en PROLISP n'est pas compilée.

III.2.6 - Utilisation de la maquette PELAGIE

Nous ne donnons ici que quelques indications pour utiliser PELAGIE. En effet, l'interface du système se présente sous la forme de menus arborescents, où toutes les réponses possibles aux diverses questions sont précisées d'avance par le système. Quiconque doit donc pouvoir utiliser PELAGIE sans avoir recours à un manuel complet d'utilisation.

Remarque : Dans le vocabulaire de Pelagie, une même session englobe tous les travaux faits entre un chargement de la maquette et le **(quit)** de l'interpréteur LISP suivant.

Des exemples commentés de sessions figurent en annexe.

Nous ferons ci-dessous quelques commentaires généraux sur les fonctions disponibles dans PELAGIE.

Le menu principal (ou niveau 0) qui apparait à l'écran dès que l'usager a chargé la maquette, ou tapé sous LISP : "(Pelagie)", lui permet d'accéder à diverses fonctions. A chacune de ces fonctions correspond un interface par menu, et ainsi de suite ... (menus arborescents). Lorsque l'usager en a fini avec une fonction, il a accès au menu juste précédent, qu'il peut faire afficher ou non à l'écran.

Remarque

1 - Au niveau de la maquette, tous les dialogues ont lieu à l'écran d'un terminal. L'utilisateur peut très bien faire tracer l'exécution d'une session sur un fichier dont il tirera un listing ensuite il s'agit d'utiliser les fonctions système classiques adhoc. Toutefois, cela n'a d'intérêt que lorsque l'utilisateur dispose d'une imprimante proche de lui : aussi cette option n'a pas été intégrée de façon systématique dans la maquette.

2 - Par défaut, Pelagie travaille sur l'algèbre linéaire. Cependant, il est prévu que l'usager puisse changer (ou choisir) le domaine de travail. En réalité cela n'a pas de sens pour Pelagie puisque elle ne connait pas d'autre domaine.

3 - Dans le menu principal, la fonction (-1) n'est pas pour l'utilisateur (elle sert à modifier les variables globales de Pelagie, dont le domaine).

III.2.6.1 - Fonctions documentaires(0,1,2,5,6)

. Parfois, le Système ne connait pas de réponse à la question que peut poser l'usager : le symbole "nil" (importé de LISP) peut alors apparaître à l'écran.

. Lorsque le Système demande à l'usager de nommer un problème, celui-ci doit répondre en tapant le "nom abrégé" de ce problème : par exemple, resolsl est le nom abrégé du problème "Résolution de système linéaire avec un seul second membre". Si l'usager ne sait pas donner une telle

réponse, le Système l'aidera en lui donnant la liste des problèmes qu'il sait résoudre, avec les noms abrégés correspondant.

. L'utilisation de ces fonctions est simple : pour plus de détails, voir les exemples de session (IV.3).

III.2.6.2 - Fonction résolution de problème (3)

. La remarque de IV.2.1 concernant les noms de problème est valable également ici.

. Les critères usager (précision etc...) intervenant dans le problème doivent être énoncés par ordre de priorité décroissante. Le Système ne sait pas traiter deux critères d'importance égale (sauf s'ils ne sont pas contradictoires ...).

. Si l'usager ne veut résoudre qu'un seul problème à la fois, il n'est pas obligé de baptiser les variables de ce problème.

. Si l'usager veut résoudre un (ou des) problème(s) dont les variables ont les mêmes noms que celles de problème(s) résolu(s) auparavant au cours de la même session (ie. depuis que Pelagie a été chargée), et si l'usager n'a pas annulé les caractéristiques de ces variables par la fonction 4, alors ces caractéristiques sont considérées comme vraies pour la résolution présente : il est donc normal que le système ne pose aucune question concernant ces variables à l'usager.

. Attention donc au cas où on résoud plusieurs fois de suite un problème (pas forcément le même) en ne nommant pas ses variables : dans ce cas, le nom de la variable donné implicitement par le système est son rôle dans le problème (ex : "matrice à inverser"). Si l'usager n'a pas annulé les caractéristiques qu'avait la variable "matrice à inverser" au cours de la résolution précédente (par la fonction 4), ces caractéristiques sont prises en compte pour la résolution courante.

. Il n'est pas possible dans cette maquette, d'interrompre une fonction de recherche, pour modifier les données sur certaines variables et continuer directement l'exécution. La seule façon de s'en sortir est

d'interrompre l'exécution avec **break/g**, puis de retaper (**Pelagie**), et répondre par l'option **3**.

. La fonction Résolution de problème a deux sous-fonctions :

1 - vérification de l'adéquation d'un outil (type de méthode, méthode,...module FORTRAN) à un problème.

2 - recherche des meilleurs outils pour répondre un problème.

. En cas de problème de manipulation de cette fonction 3, se référer aux exemples de session en IV.3.

III.2.6.3 - Fonction d'accès aux données d'un problème (4)

Cette fonction permet de visualiser le dernier problème traité, et les contraintes portant sur ses variables, ainsi que sur toutes les variables de problèmes mémorisées par le système depuis le début de la session courante.

Cette fonction permet aussi d'annuler toutes les contraintes concernant une ou plusieurs variables (en vue d'autres résolutions de problème, par exemple).

Par contre, elle ne permet pas d'annuler directement une seule contrainte portant sur une variable donnée. Cette option ne pose pas de problème d'implémentation et serait intéressante dans le cadre d'une réalisation à plus grande échelle.

III.3 - CONCLUSIONS EN VUE DE LA REALISATION D'UN SYSTEME "GRANDEUR NATURE"

A l'issue de la réalisation de la maquette, nous pouvons tirer quelques conclusions, portant sur le choix définitif du mode de représentation interne des connaissances, et sur les coûts de développement et les performances à attendre d'un système qui traiterait tout le calcul numérique. Enfin, nous avons mis en évidence certains

points qui demanderaient une étude plus approfondie.

III.3.1 - Choix du mode de représentation des connaissances

Dans la maquette, nous avons choisi la logique du premier ordre pour représenter deux types de concepts, objets et règles.

Nous avons déjà mentionné en III.1.6.2 l'intérêt d'une représentation hybride, alliant la logique des prédicats et une représentation par frames, qui représenterait la réalité de façon plus appropriée (en particulier, via un mécanisme d'héritage implicite de propriétés entre types de méthodes, méthodes, ..., modules). L'avantage d'une représentation par structures serait encore plus important si le système prenait également en compte les valeurs numériques des données des problèmes des usagers (ex : représentation explicite des coefficients numériques d'une matrice).

Les difficultés rencontrées actuellement par ceux qui ont travaillé dans cette voie concerne plusieurs points, parmi lesquels (/BEN 86/) :

- difficulté pour obtenir une représentation réellement homogène,
- problèmes de cohérence entre les objets et les règles mentionnant les objets,
- difficultés pour générer des explications, à cause de la présence de deux modes d'inférences différents, accrues si l'on veut éviter qu'il ne s'agisse que d'explications - traces.

Toutefois, cette voie mériterait que l'on s'y investisse, car elle semble très prometteuse. La solution que nous avons adoptée avec la logique des prédicats est convenable actuellement, car c'est un formalisme qui a été relativement bien étudié : nous disposons de résultats ou d'outils assez intéressants pour satisfaire nos besoins. Toutefois, cette solution n'est pas définitive.

III.3.2 - Remarques sur les coûts

III.3.2.1 - Coûts de développement

Notre champ d'application était relativement restreint par rapport à l'envergure du système visé ; pourtant, tout en s'étant réalisée dans des conditions particulièrement favorables, la phase d'acquisition des connaissances a été très longue (environ deux mois) par rapport à la durée du développement de PELAGIE.

Dans la réalité, si l'on garde l'hypothèse selon laquelle les experts n'insèrent pas directement eux-mêmes leur savoir-faire dans la base de connaissances, cette étape d'acquisition des connaissances risque fort d'être la plus longue dans la réalisation du système. Il est difficile d'évaluer sa durée, car elle ne dépend pas que de facteurs techniques ; le nombre d'experts intervenants, ou surtout des conditions psychologiques défavorables, peuvent occasionner des difficultés importantes dans l'élaboration de la base de connaissances (cf. III.2.1.1), retardant des délais de réalisation du système. Il faut donc prévoir d'accorder beaucoup de temps à cette phase, et s'efforcer qu'elle se déroule de façon aussi harmonieuse et soigneuse que possible ; le succès ou l'échec du système dépend beaucoup de cette tâche délicate, et source possible de nombreux facteurs d'erreurs.

III.3.2.2 - Coûts en performances

Dans ce paragraphe, nous faisons l'hypothèse que les connaissances sont exprimées en logique du premier ordre.

Le temps d'exécution d'une résolution, ou démonstration sur la base de connaissances, dépend du nombre d'inférences effectuées pendant cette démonstration. Or, nous travaillons en logique d'ordre 1 ; ce nombre d'inférences peut donc être très variable, selon chaque cas particulier de démonstration. Ce nombre dépend bien sûr de la taille de la base de connaissances, mais il n'y a pas de rapport constant entre les deux quantités : le nombre d'inférences devient rapidement très important dès que l'on augmente le nombre de règles. Si la base de connaissance englobe tous les domaines du calcul numérique, il y aura

donc intérêt à découper la base en secteurs de telle façon à minimiser le temps de parcours des règles et le nombre d'inférences ; il s'agira de trouver un bon compromis avec le temps de gestion de mémoire (cf. III.2.5.2.2).

D'autre part, le temps de calcul d'une démonstration dépend aussi des performances de l'interpréteur des règles. Nous avons utilisé une version relativement rapide de PROLISP, très suffisante pour une réalisation de niveau maquette. Toutefois, PROLISP ne fonctionnant qu'en interprété, il risque de ne plus être assez performant pour une réalisation grandeur nature. D'autre part, on pourra éviter d'éventuels problèmes de place mémoire en utilisant un outil permettant de compiler les règles.

III.3.3 - A propos du traitement des explications

L'objet de ce paragraphe consiste à critiquer ce qui a été fait dans PELAGIE, pour voir sur quels points des améliorations pourront être apportées lors de la réalisation d'un système réel.

De façon générale, dans PELAGIE la fonction explication n'a pas été envisagée dès le départ au même titre que la fonction résolution : nous avons porté notre attention d'abord sur la résolution, et ensuite sur les explications. Ceci n'est pas la meilleure approche : les deux fonctions, d'ailleurs interdépendantes, doivent être étudiées en parallèle, et sont d'égale importance dans un système de résolution de problème (/KEO 85/).

III.3.3.1 - Explications positives

Dans PELAGIE, les explications positives générées sont suffisantes pour aider l'expert à vérifier la validité de sa base de connaissances. Par contre, même si elles aident effectivement l'usager à comprendre les raisonnements du système, elles sont assez lourdes ; ce mixage entre des traces d'exécution et du texte permettant d'expliquer le principe du raisonnement n'offre pas une présentation vraiment globale des étapes d'une résolution. D'autre part, cette présentation ne tient pas compte de ce que sait déjà l'usager ; par exemple, le texte

généré peut comprendre des redites comme "... car la matrice est symétrique, puisque définie-positive" : ce dernier point est facile à corriger grâce à une bonne analyse du texte constituant les explications.

Pour améliorer ces explications, outre une analyse plus fine des explications déjà produites avant d'en générer d'autres, il conviendrait peut-être d'utiliser les arguments trace de façon moins systématique, et de créer peut-être une autre base de connaissances parallèle ; celle-ci contiendrait les principes généraux guidant la résolution de l'expert, et codés de façon éclatée dans la base actuelle de connaissances de PELAGIE ; cette base parallèle ne serait exploitée que pour la génération d'explications.

III.3.3.2 - Explications négatives (ou assimilé)

D'abord, cette fonction n'existe pas en tant que telle dans PELAGIE, mais elle est traitée sous la forme de résolution assistée.

Les explications générées présentent le même défaut que les explications positives (pas assez synthétiques). Mais surtout, elles sont parfois insuffisantes pour vraiment aider l'usager à comprendre pourquoi un outil ne peut pas résoudre un problème, dans le cas de base de connaissances incomplète ou de données ne répondant pas aux contraintes exprimées dans des règles inter-objet. Ceci est un défaut "par construction", puisque ces explications ne peuvent être générées qu'avec effacement des règles au cours d'une résolution. Notre traitement du "pourquoi-pas" est entièrement dépendant du domaine d'application (ie. de la façon de résoudre un problème) et ne répond pas entièrement à la question posée.

C'est pourquoi il faut envisager une autre solution, indépendante du domaine d'application, et répondant réellement à la question ; un processus de traces, tel que celui que nous avons utilisé dans PELAGIE, ne peut pas être satisfaisant.

A l'heure actuelle, le problème du pourquoi-pas est généralement mal résolu dans les systèmes à base de connaissances.

Aussi, nous avons retenu ce thème pour en faire une étude plus approfondie, qui fera l'objet de la seconde partie de ce document : la question du pourquoi-pas est en effet fondamentale dans un système expert, au même titre que la fonction résolution (ces fonctions sont duales).

III.3.4 - Autres extensions possibles de l'étude

Ce paragraphe concerne quelques points délaissés au cours de notre étude, qu'il serait intéressant de développer pour rendre le système plus fonctionnel ou pour étendre ses capacités.

Les premiers points doivent nécessairement être traités pour que le logiciel réponde aux besoins exprimés en I.2.1. Il s'agit d'une part de la possibilité de gestion d'historiques d'utilisation ; ce sujet a été rapidement abordé dans /SAU 84-2/, où figure une ébauche d'organisation possible d'un fichier contenant les informations principales mémorisées au cours des sessions d'utilisation, que l'on pourrait exploiter par des logiciels de statistiques (cf. annexe 2) ; tout reste à faire à ce niveau. D'autre part, un module d'acquisition des connaissances via un gestionnaire doit être réalisé ; ceci ne doit pas soulever trop de difficulté de réalisation dans un premier temps (analyseur d'un langage d'interface, et traducteur des informations exprimées dans ce langage, en langage de représentation interne des connaissances). Néanmoins, à long terme, ce module doit être assez convivial, pour faciliter la tâche laborieuse de la mise à jour de la base de connaissances (insertion de "structures", et des règles inter-objets et de comparaison selon critères associées). Pour atteindre cet objectif, il sera nécessaire de s'investir dans des techniques assez sophistiquées.

Toutefois, il est possible d'aller plus loin dans l'étude du module d'acquisition des connaissances, et d'aborder des problèmes nettement plus complexes : maintien de la cohérence des informations de la base de connaissances lors des mises-à-jour (ceci pouvant être lié à la résolution du problème du pourquoi-pas), apprentissage "intelligent" de la part du système (propositions de la part du système de nouvelles règles pour :

- corriger des erreurs ou des omissions dans la base de connaissances,
- généraliser des résultats obtenus à partir de résolutions de problèmes particulier, de façon à optimiser les prochaines résolutions,
- mieux traiter les cas d'échecs, etc...

D'autre part, on peut chercher à réaliser un interface plus souple que celui de PELAGIE. L'idéal serait de permettre à l'usager d'exprimer son problème en langage mathématique conventionnel (du moins, tant que l'on ne parle pas de sous-programme) ; le système devrait savoir analyser correctement cette formulation, puis résoudre le problème formulé. Ceci constitue un projet ambitieux mais très digne d'intérêt (/MOU 86/).

Il serait également souhaitable d'étendre les possibilités de résolution de PELAGIE à des problèmes plus complexes. Il peut s'agir de problèmes nécessitant plusieurs étapes de résolution pour des raisons numériques (ex : résolution d'un système linéaire mal conditionné) ; mais cela peut aller également jusqu'à des problèmes de plus grande envergure, se décomposant en un enchaînement de problèmes plus simples et éventuellement très divers (ex : problèmes de résolution par éléments finis).

Cette liste est loin d'être exhaustive car beaucoup de thèmes de recherche dans les systèmes experts trouveraient un terrain d'études ici : possibilité plus fine d'interaction de l'usager ou plutôt de l'expert sur la stratégie de résolution du système, gestion de grosses bases de connaissances (ce problème a déjà été mentionné à une petite échelle dans PELAGIE, mais on y sera confronté de façon plus importante lors du développement d'un système grandeur nature), etc...

Enfin, certains problèmes soulevés dans PELAGIE ne sont pas propres au contexte systèmes experts, mais dépendent de son domaine d'application. Par exemple, en ce qui concerne la comparaison d'outils selon divers critères (précision, place mémoire, temps d'exécution), plusieurs questions méritent d'être approfondie. D'une part, nous nous sommes limités à énoncer des relations de comparaison qualitatives et binaires sur les critères, entre les outils ; or il est possible de

caractériser quantitativement un module ou un algorithme selon la précision, le temps d'exécution ou la place mémoire occupée, en fonction des dimensions du problème traité (/ARO 78/), puis de comparer ensuite les formules obtenues à l'aide d'un module de calcul formel algébrique - cela demande du temps, mais donnerait des informations plus précises que celles de PELAGIE. D'autre part, il serait intéressant de pouvoir résoudre le problème de compromis entre critères d'égale importance pour l'usager (cf. remarque en III.1.3.2) - ce problème semblant déjà nettement plus compliqué que le précédent. Enfin, ces travaux sur la prise en compte des critères peuvent être généralisés pour des enchaînements d'algorithmes ou "problèmes plus complexes".

2ÈME PARTIE

EXPLICATIONS NEGATIVES DANS LES SYSTEMES A BASES DE CONNAISSANCES FORMULEES EN LOGIQUE DES PREDICATS

INTRODUCTION

Le problème que nous nous proposons d'étudier dans cette seconde partie s'est posé au cours de la réalisation de la maquette Pélagie. En effet,si le système a proposé un outil O - ou aucun outil - pour résoudre le problème P de l'usager, ce dernier n'a aucun moyen de savoir pourquoi un outil O1 ne convient pas si cela est dû à des caractéristiques de P ; la technique utilisée pour générer des explications dans Pélagie ne permet pas de signaler qu'une règle qui aurait dû être appliquée pour conclure O1 ne l'a pas été à cause de certaines données de l'usager.

Il s'agit donc de savoir pourquoi un fait $P(x_0,...)$ n'est pas démontrable à partir d'une base de connaissances BdC (ce que nous noterons $\sim P(x_0,...)$, ie BdC $\not\vdash P(x_0),...))$, ou même pourquoi $P(x_0,...)$ est logiquement faux sur cette base de connaissances (ie BdC $\vdash \neg P(x_0,...)$ - si l'on fait l'hypothèse de monde fermé)

Jusqu'à présent, peu de travaux ont porté sur ce problème, dit des "explications négatives", lorsque la base de connaissances est formalisée en logique des prédicats. Nous ferons donc d'abord un état de l'art sur tout ce qui a trait à ce sujet. Puis nous définirons notre problème et nos hypothèses de travail. Nous donnerons alors une première approche envisagée pour résoudre notre problème, qui a abouti à une impasse : nous analyserons les raisons de cet échec. Cela nous permettra d'expliquer nos motivations pour la méthode que nous avons finalement conçue, suivant le principe de résolution, et que nous exposerons en détails.

Signalons que les problèmes d'implémentation n'ont pas été abordés.

I - ETAT DE L'ART EN MATIERE D'EXPLICATIONS DANS LES SYSTEMES A BASE DE CONNAISSANCES

I.1 - INTRODUCTION

Historiquement, les systèmes à base de connaissances étaient d'abord conçus pour faire de la résolution de problèmes. Le besoin d'un outil permettant aussi de contrôler comment se faisait cette résolution n'a pas tardé à se faire sentir (/BUCH 84/). De là est née une seconde fonction, la génération d'explications du système, pour l'usager néophyte ou le concepteur de la base.

En réalité, cette fonction est au moins aussi importante que la résolution ; il s'agit d'étudier et présenter la manière de raisonner du système (KAS 86/) de façon à répondre clairement à diverses questions que se pose l'usager sur sa démarche pendant la résolution d'un problème (/SCO 84/). Nous avons déjà vu que les principales questions auxquelles doit au moins répondre une fonction d'explications, sont : (cf. III.5.2.I)

- en explications positives (sur le raisonnement effectivement suivi par le système) : pourquoi ?, comment ?.

- en explications négatives (sur un raisonnement qui n'a pas été effectivement suivi par le système) : pourquoi-pas ?

En réalité, les questions possibles peuvent être très diverses dans chaque cas, concernant les valeurs affectées (ou pas), les paramètres utilisés (ou pas), les règles choisies (ou pas), etc... (/SCO 84/).

Les problèmes soulevés par la génération d'explications sont nombreux et complexes :

- restitution et justification d'un raisonnement ayant abouti à un résultat (du latin, explicare = déplier), de façon la plus naturelle possible

- non redondance des informations figurant dans l'explication

- prise en compte de l'attente de l'utilisateur, et de son niveau de connaissances dans le domaine de la base de connaissances

- formulation des informations dans un langage le plus compréhensible possible à l'utilisateur

- etc...

Dans un premier temps, nous dégagerons, avec exemples à l'appui, les deux grandes catégories de techniques utilisées jusqu'à présent pour générer des explications dans les systèmes à base de connaissances. Puis nous ferons plus particulièrement état des travaux concernant les explications négatives lorsque les connaissances sont exprimées en logique d'ordre 0 ou 1.

I.2 - TECHNIQUES GENERALES D'EXPLICATIONS

Deux types d'approches différents se dégagent. Les techniques les plus fréquemment utilisées reposent sur une manipulation plus ou moins sophistiquée de l'arbre de résolution du problème, générant des explications nécessairement très liées au mode de représentation des connaissances. D'autres systèmes utilisent des connaissances supplémentaires pour générer des explications plus globales, ceci pouvant mener à des modifications majeures dans l'architecture classique des systèmes à bases de connaissances.

I.2.1 - Manipulations de l'arbre de résolution

I.2.1.1 - Restitution de l'arbre de résolution

La technique la plus simple consiste à restituer directement la partie de l'arbre de résolution correspondant à la question posée par l'usager ; on obtient exactement la trace des règles appliquées pour conclure à un but. En logique des propositions, c'est ainsi que fonctionne MYCIN (/HAND/). En logique des prédicats, plus précisément avec PROLOG, on sait récupérer l'arbre de résolution d'une démonstration (/CLA 82/ et cf. 1ère partie, III.2.3).

En effet, Clark et McCabe ont montré que l'on pouvait répondre aux questions comment ? (à la fin d'un démonstration) et pourquoi ? (en cours de démonstration, lors de l'acquisition d'une information de l'usager par le système) en introduisant des arguments de trace dans les littéraux des clauses. Par convention, nous écrirons toujours les arguments de trace en dernière position dans ce qui suit.

a - structure de l'argument de trace COMMENT

Soit la clause $(L\ *x) \vee \neg(L_1\ *y) \vee \neg(L_2\ *z)$

Pour permettre de répondre à la question COMMENT, on peut transformer la clause ci-dessus comme suit :

$$(L\ *x \quad \underbrace{(*l_1 \text{ et } *l_2)}_{*l}) \quad \vee \neg(L_1\ *y\ *l_1 \vee \neg(L_2\ *z\ *l_2)$$

où *l, $*l_1$ et $*l_2$ sont des arguments de trace ayant tous la même structure (littéral effacé, prémisses ayant permis de l'effacer). Lorsque le littéral (L *x *l) est effacé, *l est instancié et l'on a récupéré une trace de la résolution. On a montré (L *x) en montrant $*l_1$ et $*l_2$ (qui sont instanciés, au moment où l'on exploite cette trace) ; de même, on peut ensuite expliquer comment $(L_1\ *y)$ et $(L_2\ *z)$ ont été effacé, en regardant le contenu des arguments $*l_1$ et $*l_2$; et ainsi de suite ...

b - structure de l'argument de trace POURQUOI

Nous supposons que l'on peut désigner chaque clause, par exemple par un numéro de clause. On introduit des arguments de trace dans les clauses comportant des littéraux dont l'effacement nécessite une question à l'usager. La procédure d'acquisition d'une réponse de l'usager admet comme réponses possibles : oui,non (concernant l'objet de la question), et pourquoi (pourquoi le système a-t-il besoin de cette information ? que cherche-t-il à montrer ?), et se fait par un prédicat d'interface approprié).

Soit la clause R_1: (A *x) V ¬(B *y) V ¬(C *z) V ¬(D *w), où l'effacement de (D *w) nécessite une question à l'usager. On peut transformer cette clause comme suit.

(A *x *a) V ¬(B *y) V ¬(C *z)

V ¬(D *w $\underbrace{([R_1.(B\ *y)\ \text{et}\ (C\ *z)].\ *a))}_{*d}$

où *a et *d sont des arguments traces ayant tous la même structure :

- règle en cours d'effacement, prémisses déjà effacées
  contexte dans lequel on cherche à effacer le littéral tête de cette règle.

Lorsque le système cherche à effacer (D *w), il appliquera la règle (D *w *d) V ¬(DEMANDER (D *w) *d), qui enclenchera la procédure d'acquisition. L'utilisateur peut répondre POURQUOI ?, auquel cas la réponse est donnée par la première partie du terme *d ; l'utilisateur peut reposer ensuite la question POURQUOI (la seconde partie de *d ayant une structure analogue à celle de *d) et ainsi de suite, jusqu'à ce que l'argument trace soit réduit à VIDE (ou NIL) - ce qui signifie que l'on a atteint la racine de l'arbre de résolution, et qu'il n'y a donc plus rien à expliquer.

Notons que les introductions de ces traces pourraient être faites automatiquement par un programme qui opérerait la transformation adéquate à partir des clauses initiales ; cependant, il ne faut peut être pas introduire les traces systématiquement dans tous les littéraux,

pour obtenir une information quelque peu compréhensible.

Toutefois, un arbre de résolution complet ne peut constituer une explication au sens véritable de ce terme, car trop lié aux étapes introduites pour des besoins de programmation.

L'idée est alors de simplifier l'arbre de résolution pour obtenir une information plus parlante pour l'usager.

I.2.1.2 - Simplifications de l'arbre de résolution

En logique des propositions, le travail de B.Safar est très représentatif dans ce sens. Elle a développé un module d'explications adapté à des bases de connaissances modélisées de la façon suivante :

- les faits manipulés sont des couples (attribut, valeur)
  ex (dans le domaine du jeu de Bridge): *(distribution,bicolore)*

- les prémisses sont de la forme :
  (attribut opérateur valeur), ou (attribut 1 opérateur attribut 2)
  où opérateur $\in \{=, \neq, <, >, \leq, \geq\}$

- les conclusions sont de la forme :
  (attribut = valeur) (assignation simple)
  ou (attribut 1 = attribut 2) (assignation composée)
  ex si distribution = bicolore et longue1 ≠ longue2
  alors annonce-possible = la plus longue
  (où distribution, longue1, longue2, annonce-possible sont des noms d'attributs, et bicolore est une valeur possible pour distribution)

- l'hypothèse de monde fermé est faite.
  Ces connaissances sont interprétées par le moteur ZERO+.

N8 : par rapport à la logique des propositions, la logique dite d'ordre 0+ intègre des pseudo-variables qui ne peuvent prendre qu'un nombre fini de valeurs.

Pour élaguer l'arbre de résolution d'un but, on décide de ne conserver dans l'arbre que les faits "pertinents" pour l'utilisateur (/SAF 85/). En réalité, ce sont les faits non-pertinents qui sont déterminés, par exemple comme suit (ces termes sont placés entre crochets dans les exemples de règles présentés) :

- on suppose que les faits dont la valeur booléenne a été déduite par l'hypothèse de monde fermé, et placés comme première prémisse d'une règle, ne sont pas pertinents et ne méritent donc pas d'être justifiés (en effet, on suppose que leur rôle est juste d'organiser hiérarchiquement la base de connaissances).

  ex: si[pas exception, et pas soutien-en-majeure], et
  privilégier-la-majeure,
  alors annonce = la majeure

- les feuilles de l'arbre de résolution dont tous les frères sont aussi des feuilles, et ne comportant pas de terme contenant l'opérateur =, seront supprimées ; en effet, on suppose qu'elles correspondent à un niveau de détail trop fin et qu'il suffit de conserver le nom du noeud père.

  ex: si[ouverture/ pique et nb-pique ⩾ 5]
  alors pique-annonçable (=vrai)

- si la déduction d'un fait a nécessité le déclenchement de plusieurs règles, on simplifie la dernière règle déclenchée, celle-ci ne servant généralement qu'à affecter une valeur à un attribut ; ses prémisses déjà rencontrées dans l'arbre ou ne comportant pas l'opérateur = seront jugés non pertinents, et éliminés de l'arbre

  ex: si[privilégier la majeure, et pique-annonçable, et
  nb-pique >nb-coeur]
  alors la majeure = pique

- etc ...

Ces différents filtres, mutuellement indépendants, permettent d'élaguer l'arbre de façon à ne conserver que des informations supposées intéressantes pour l'usager. Appliqués sur un exemple particulier (règles d'annonces au jeu de bridge), ils permettent de fabriquer un bon

squelette d'explications pour l'usager. Leur intérêt est de ne reposer que sur des critères syntaxiques ; toutefois, il reste à voir si ces critères restent toujours intéressants sur d'autres domaines d'application ; d'autre part, ces filtres reposent sur une modélisation bien particulière des connaissances.

En logique des prédicats restreinte aux clauses de Horn, il est également possible de simplifier la présentation de l'arbre de résolution (/ERI 85/), uniquement par des critères syntaxiques, par exemple avec les mesures suivantes :

- élimination des clauses intermédiaires, n'introduisant ni variable supplémentaire, ni décomposition de problème en plusieurs sous-problèmes (au sens de la résolution) sous l'hypothèse de non-récursivité.

- élimination de variable intermédiaire commune à plusieurs littéraux d'une même clause,

  ex: $GP(x,z) \vee \neg P(x,y) \vee \neg P(y,z)$

  "le père du père de x est z"

- exploitation d'une même place particulière d'une variable partagée par plusieurs littéraux d'une même clause

- simplification dans le cas de sous-branches apparaissant plusieurs fois dans l'arbre, etc...

Ces transformations permettent d'obtenir des traces plus dépouillées, donc plus compréhensibles pour l'utilisateur.

Toutefois, si l'on s'arrête là, les explications générées ne sont pas assez structurées, et l'utilisateur est généralement noyé sous un flot d'informations, toutes mises au même niveau, certaines l'intéressant moins que d'autres (trop liées au code).

Certains travaux procèdent donc à des remaniements plus vigoureux de l'arbre, pour aller dans un sens plus "orienté usager". En

logique d'ordre 0+ (logique où les prédicats peuvent avoir un argument valué), le système BLAH (/WEI 80/) est un bon exemple d'application de cette démarche. Par exemple, la base de connaissances peut être divisée en deux blocs : le premier contenant les connaissances de l'usager, le second celles de l'expert ; toutes les parties de l'arbre de résolution reposant sur les connaissances de l'usager ne lui seront pas présentées,, car on suppose qu'il sait tout ce qui peut être réduit à partir de celles-ci. Une autre partition possible de la base est par ensemble de prémisses initiales regroupées. D'autre part, un arbre n'est présenté à l'usager que jusqu'à un niveau de profondeur fixé, arbitrairement égal à 3, ce qui correspond à une mesure du niveau de détail des explications - le processus pouvant être réitéré localement autant de fois que nécessaire. De même, un arbre trop complexe sera découpé en sous-arbres et à chaque sous-arbre sera associé un commentaire résumé - là aussi, le processus pouvant être itéré autant de fois que nécessaire. Certaines de ces mesures sont peut-être appliquées de façon trop arbitraire ; par exemple il serait intéressant de donner une profondeur de détail d'explications qui tienne compte du niveau de l'usager ; d'autre part, l'hypothèse liée à la première mesure est probablement trop brutale et devrait être plus nuancée. Toutefois, les informations ainsi générées sont mieux structurées et donc plus faciles à exploiter pour l'utilisateur.

Par ailleurs, les résultats obtenus par les manipulations d'arbre de résolution sont généralement présentés à l'utilisateur via un module de traduction du code en langage "pseudo-naturel" (ou plutôt, composé de mots clés). Par exemple, dans MYCIN, les règles sont traduites à partir de LISP en une forme compréhensible par l'utilisateur. Les travaux présentés dans /ERI 85/ font référence aussi à un tel module : les symboles de prédicats sont des mots clés du texte présenté à l'utilisateur. Dans BLAH, le module de traduction renforce les articulations logiques du raisonnement en introduisant des conjonctions de coordination entre les différentes composantes de l'arbre. De façon générale, ce module d'interface est absolument nécessaire pour que l'utilisateur puisse exploiter l'arbre de résolution transformé et réduit.

I.2.1.3 - Conclusion

Nous retiendrons que les techniques basées sur la manipulation de l'arbre de résolution peuvent déjà donner des renseignements intéressants à l'utilisateur. Tous les efforts des travaux dans cette voie ont visé à se dégager le plus possible du code sous-jacent. Toutefois, même dans le meilleur des cas, les informations produites ne peuvent être qu'un bon condensé à partir de celles contenues dans la base de connaissances ; elles relèvent plutôt du bon commentaire que de véritables explications (au sens : raisonnement sur la solution fournie), qui devraient également rendre compte des connaissances implicites ou plus générales n'apparaissant pas parmi celles exploitées par le résolveur de problème. Il est difficile d'obtenir ainsi des explications synthétiques, plus naturelles à l'homme, qui permettraient à l'utilisateur de comprendre un raisonnement dans ses toutes grandes lignes d'abord, puis progressivement plus en détail.

C'est pourquoi d'autres travaux font appel à un autre espace de connaissances pour générer des explications de façon plus satisfaisante dans ce sens.

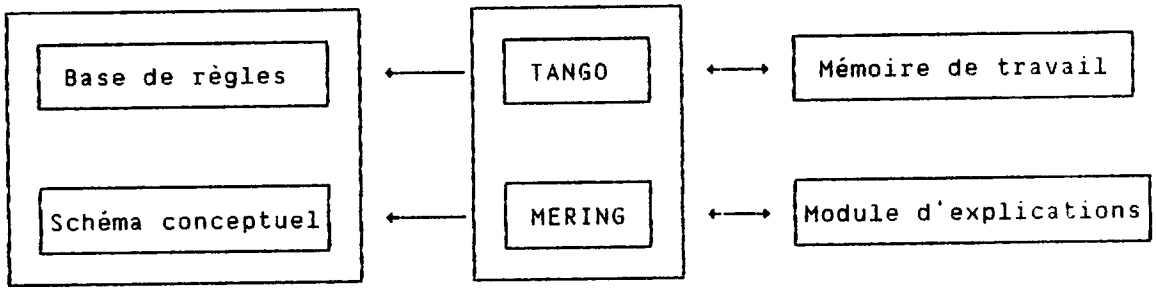
I.2.2 - Utilisation de connaissances autres que celles utilisées en résolution

Le but est donc de pouvoir faire apparaître des connaissances moins opérationnelles, généralement en amont de celles exprimées pour la résolution. Deux grands types d'approches existent : la première consiste à structurer les connaissances de façon adéquate, la seconde consiste à introduire de la (méta-)connaissance ne servant que pour la génération d'explications.

I.2.2.1 - Distinction entre différents types de connaissances du système

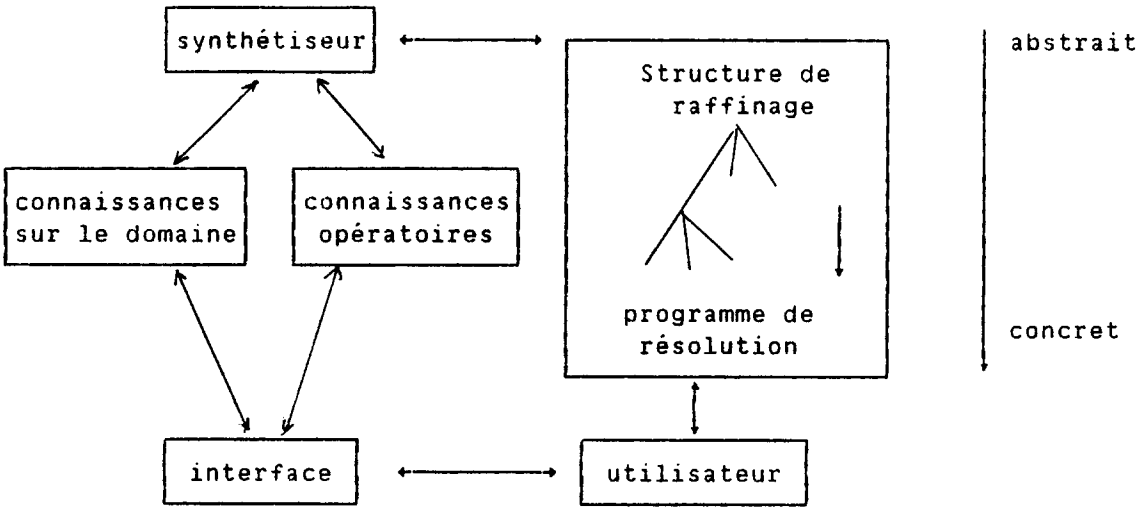
L'idée est généralement de considérer à part les connaissances sur le domaine, de niveau conceptuel (concepts ou objets du domaine, relations entre concepts), de celles utilisées directement en résolution de problème.

En logique des prédicats, G. Kassel (/KAS 86/) utilise d'une part les règles utilisées en résolution, d'autre part le schéma conceptuel. Il se restreint au cas des prédicats à deux arguments, le second représentant la valeur du prédicat (considéré comme fonction) pour le premier ; ce premier argument correspond à un objet du schéma conceptuel. Avec ces hypothèses, il travaille à partir de l'arbre de résolution en essayant d'y distinguer certains blocs concernant des objets divers. Les explications générées expriment alors les relations entre ces objets, tirées du schéma conceptuel. L'architecture du système est donc la suivante :



Une autre démarche, encore plus "révolutionnaire", est celle de Swartout (/SWA 81/, /SWA 83/). Celui-ci dispose d'une part de connaissances conceptuelles, d'autre part de connaissances opératoires permettant d'utiliser le savoir-faire. A partir de là, il propose l'idée d'un synthétiseur du programme de résolution experte. Le produit synthétisé permet de faire de la résolution de problème, alors que les connaissances de départ permettent de générer des explications ou justifications correctes. Le système XPLAIN a été appliqué pour un programme d'aide à l'application de la Digitale en médecine, et a fourni des explications plus synthétiques et abstraites, moins liées aux étapes de calcul. Une démarche voisine a été plus récemment suivie dans d'autres travaux (/OHA 87/).

L'architecture d'un tel système est la suivante :



Dans les deux cas, cela revient à proposer une nouvelle architecture pour les systèmes à base de connaissances. Toutefois, ces idées sont plutôt en cours de validation et ne sont pas très répandues. Certains problèmes sont mal résolus. Par exemple, le synthétiseur de Swartout (programmation automatique) exige que les connaissances soient modélisées de façon "adéquate" ; il faut donc s'assurer de l'indépendance vis-à-vis du domaine d'application. D'autre part, une autre caractéristique des bases de connaissances est l'évolutivité ; reste à vérifier que ces modèles préservent bien ce concept d'évolutivité.

En revanche, les explications générées, dans les deux cas, sont plus satisfaisantes car elles reflètent autre chose qu'une simple trace.

I.2.2.2 - Connaissances dédiées à l'explication

Un travail assez représentatif dans cette optique est celui de Clancey (/HAS 84/), /CLA 83/,). Il constate que les connaissances utilisées par l'expert concepteur d'une base de connaissances ne transparaissent pas toutes dans les règles utilisées par l'interpréteur en résolution de problème, ceci étant vrai à plusieurs niveaux : stratégie générale de résolution d'un problème, justification de l'ordre d'application des règles par l'interpréteur, regroupement des règles par thèmes (identification de problème, déduction,...), justification de

l'ordre des prémisses d'une règle, etc... Or ces connaissances implicites sont nécessaires pour fournir de véritables explications à l'utilisateur.

Il introduit donc d'autres connaissances dans la base de connaissances, qui ne serviront qu'à générer des explications. Le système KS (pour knowledge Source) présente un support hiérarchisé permettant d'exprimer toutes ces connaissances parallèles associées aux règles utilisées en résolution.

D'une part, il contient des méta-connaissances, indépendantes du domaine d'application, sur la stratégie générale de résolution de problème ; ces informations sont organisées en un arbre, dont les noeuds sont des tâches abstraites, reliés par des méta-règles ; les feuilles de l'arbre sont les connaissances manipulées par le moteur d'inférences.

D'autre part, on y trouve aussi le brouillon de la rédaction d'une règle, contenant les différentes versions de cette règle - de la plus simplifiée à la plus élaborée -, les raisons de ces modifications, etc...

KS permet ainsi d'obtenir des explications plus gobales et plus naturelles que celles obtenues seulement à partir des règles de résolution : la sémantique des règles peut être dévoilée, les explications peuvent être plus ou moins détaillées. Par exemple, l'utilisation de KS a entraîné la reconfiguration de Mycin en Néomycin, lequel a pour fonctions l'explication et l'enseignement.

I.2.3 - Conclusion

Nous avons vu que la plupart des travaux se limitent à la manipulation de l'arbre de résolution : cela demande relativement peu de moyens supplémentaires par rapport à ceux mis en oeuvre pour la résolution de problèmes, et cela est à relier au fait que la fonction explication n'a été considérée au départ que comme un "plus" relativement à la fonction résolution.

Toutefois, les gens sont de plus en plus persuadés que seule

l'adjonction d'une base de connaissances explicatives - ou carrément une nouvelle architecture des systèmes à base de connaissances - permettra de fournir des explications synthétiques, que l'utilisateur pourra exploiter directement : plusieurs travaux ont pour objet d'allier les deux techniques (/ROUS 87/, /VIV 87/), pour passer réellement du stade du commentaire de résolution à celui de l'explication.

D'autre part, la prise en compte du niveau de connaissances de l'utilisateur n'est pas encore bien modélisée, ce qui constitue un handicap pour générer de "bonnes" explications. Un autre problème qui ne sera résolu que plus tard est celui du langage utilisé dans ces explications : on est toujours loin d'un langage naturel.

Notons par ailleurs que peu de systèmes traitent des explications négatives, car il est plus difficile de travailler sur un arbre de résolution en échec.... Dans le paragraphe suivant, nous examinerons donc les rares travaux qui se sont intéressés particulièrement à ce problème.

I.3 - TRAITEMENT DES EXPLICATIONS NEGATIVES

I.3.1 - Les problèmes liés à la question pourquoi-pas

Lorsqu'un système n'a pas de réponse à fournir à une question de l'utilisateur (ex. échec d'une démonstration) cela reste frustrant. Il est nécessaire de lui indiquer au moins les sources de cet échec. Ce problème n'est pas propre aux systèmes à bases de connaissances et a déjà été étudié dans un contexte de base de données (/JAN 79/).

Si la solution d'un problème est différente de celle qu'avait prévue l'utilisateur (autre solution, ou pas de solution), on peut suggérer diverses causes pour cette différence, selon le type d'utilisateur. Si celui-ci est un utilisateur néophyte (au sens "usager" de Pélagie), il suppose que les connaissances de la base sont justes et permettent de résoudre son problème ; il cherche plutôt quelles données supplémentaires auraient été nécessaires, pour conclure au but attendu ou pourquoi la nature même de ses données ne permet pas de conclure à ce

but. Bien sûr, il ne suffit pas de répondre en expliquant pourquoi le système a éventuellement trouvé une autre solution (explication positive). En revanche si l'utilisateur est le concepteur de la base, il cherchera généralement plutôt à savoir s'il n'a pas glissé une information fausse dans la base, ou s'il n'a pas oublié d'information de type savoir-faire. Formellement, les problèmes des deux sortes d'utilisateurs sont analogues ; en pratique ils se résolvent différemment. Le premier est à relier à l'acquisition de données, le second à celle des connaissances (au sens : expertes).

Par ailleurs, on retrouve les problèmes déjà évoqués au sujet de la génération d'explications dans le cas général : explications à caractère synthétique et détaché de la programmation, niveau de détail souhaité, prise en compte des connaissances de l'usager.....

Un autre problème lié au pourquoi-pas est celui de l'explosion combinatoire, qui rend encore plus difficile une formulation synthétique de l'explication. En effet, en explications positives il s'agit de relater la voie qui a permis de conclure sur un fait ; en explications négatives, de nombreuses voies peuvent avoir échoué sur le but de l'usager, et c'est de cela qu'il faut informer l'utilisateur.

Dans ce paragraphe, nous étudierons d'abord les travaux orientés spécifiquement vers l'usager ; puis nous verrons l'approche visant plutôt à mettre en cause la base de connaissances du système. Ensuite, nous étudierons d'autres façons possibles d'envisager le problème, dont la génération d'explications négatives ne constitue qu'une des applications possibles : il s'agit d'une part de la génération d'hypothèses, d'autre part du contrôle de type ; ces dernières approches ne font pas de distinction entre les différents types d'utilisateurs, car leur contexte de travail dépasse le cas particulier des systèmes experts, par exemple.

I.3.2 - Approche orientée usager

Seulement deux des travaux mentionnés dans le paragraphe précédent se sont intéressés au problème des explications négatives, en logique d'ordre 0, avec variables : nous verrons comment ils l'ont

abordé. Plus rares sont les études dans le contexte de la logique d'ordre 1.

I.3.2.1 - Logique des propositions

Parmi les systèmes qui traitent la question pourquoi-pas, se retrouvent ceux de B. Safar et de Weiner (BLAH). Leur caractéristique commune est de s'appuyer pour cela sur les informations contenues dans une sorte de descripteur des règles.

B. Safar (/SAF 85/) utilise le moteur d'inférences ZERO+. Les faits manipulés étant représentés par des couples (attribut, valeur) (cf.I.2.1), la représentation interne des connaissances permet de conserver, pour chaque attribut, une liste de propriétés : valeur effectivement associée à l'attribut, ensemble des valeurs rencontrées dans les règles pour l'attribut, ensemble des règles dans lesquelles apparait l'attribut comme condition, liste des couples (règle permettant d'affecter une valeur à l'attribut, valeur), origine de la valeur de l'attribut (déductible, demandé à l'utilisateur, monde fermé ...). A partir de ces informations, il est possible de répondre à des questions négatives simples, selon le type de l'attribut (booléen, valué) et l'origine de sa valeur.

De façon générale, le principe est le suivant :

- explorer toutes les règles qui auraient pu aboutir à la valeur attendue de l'attribut

- extraire de chacune de ces règles les prémisses en échec, d'après les données enregistrées au cours de la résolution faite auparavant

- éventuellement comparer ces prémisses fausses à celles qui ont permis de conclure à une autre réponse que celle attendue par l'usager

- réitérer sur chacune de ces prémisses ... jusqu'à arriver sur certaines conditions d'arrêt.

Les deux premières étapes du cycle sont faciles à mettre en oeuvre grâce à l'organisation des connaissances avec le moteur ZERO+. Par contre, le but du jeu consiste, comme pour les explications positives, à ne générer que des informations "pertinentes" pour l'usager, donc à élaguer l'ensemble des réponses possibles.

Pour cela, à chaque cycle, on réduit l'ensemble des prémisses en échec, en ne s'intéressant qu'au sous-ensemble des prémisses pertinentes. Dans le cas d'un fait attendu de type Attribut=valeur, tel que la valeur ne pouvait affectée que par une assignation simple, le sous-ensemble s'obtient par une recherche des prémisses communes en échec sur toutes les règles obtenues dans la première étape. S'il y a un sous-ensemble de prémisses communes,on ne réitère le processus que sur chacun des éléments de ce sous-ensemble ; sinon, on va choisir un sous-ensemble de prémisses communes au plus grand nombre possible de règles obtenues à l'issue de l'étape 1, et on va essayer d'éliminer les autres règles de l'explication finale ; pour cela, de nouveaux critères sont nécessaires. La réduction peut s'opérer alors grâce à la déclaration (par le concepteur de la base) de certains attributs comme étant "liés au contexte" ; on choisit alors de ne pas s'intéresser aux règles ayant des prémisses portant sur des attributs non liés au contexte (/ROUS 87/). D'autres critères peuvent encore être proposés.

La procédure s'arrête sur les prémisses en échec qui ne pouvaient pas être réduits par l'application d'une règle (ex : feuilles de l'arbre dont la valeur a été fournie par l'utilisateur, prémisses contenant des opérateurs comme $\neq, <, \ldots$, prémisses dont la valeur est d'origine monde-fermé).

Ces heuristiques permettent ainsi d'expliquer de façon satisfaisante les questions portant sur une valeur d'attribut booléenne d'origine monde-fermé, sur un attribut dont la valeur attendue pouvait être déduite directement par une ou des règles (dont la conclusion était : attribut=valeur), et sur les faits dont la déduction nécessite l'application simultanée de plusieurs règles (ex. attribut1=attribut2).

Le processus d'élimination des informations non pertinentes à l'usager doit être encore élargi pour être réellement efficace. Ce

dernier problème est une des composantes de la difficulté à répondre à la question pourquoi-pas, qui sera encore amplifiée en logique d'ordre 1. D'autre part, comme dans le cas des explications positives, il reste à vérifier que les critères syntaxiques, sur lesquels reposent les heuristiques, sont valables pour tout domaine d'application.

Quant au système BLAH, il dispose également d'informations portant sur ses connaissances opératoires. A chaque assertion sont associées, entre autres, une liste des assertions la justifiant, et une liste des assertions s'opposant à elle. A partir de là, si la réponse attendue par l'usager est fausse, BLAH explique pourquoi c'est faux, et ce qui est vrai, et pourquoi c'est vrai ; il utilise pour cela les techniques de structuration dynamique de la base de connaissances et de l'espace de travail selon des ensembles de prémisses initiales, et de manipulation d'arbre comme déjà mentionné en I.2.1. Toutefois, il n'y a pas de solution apportée aux problèmes rencontrés par B. Safar, et BLAH ne génère des explications satisfaisantes que sur des exemples relativement simples.

I.3.2.2 - Logique des prédicats

Le travail de Walker (/WAL 83/) s'applique au cas où les connaissances sont exprimées en clauses de Horn, interprétées en PROLOG. Or PROLOG ne permet pas de tracer un arbre de résolution en échec. L'idée est donc de modifier l'interpréteur PROLOG (nouvel interpréteur : PROLOG/EX1) de façon à pouvoir donner à l'utilisateur un arbre de résolution partielle du but en échec qui était à démontrer, faisant apparaître les points en échec.

D'une part, PROLOG/EX1 tente d'assurer les terminaisons des démonstrations en inversant éventuellement l'ordre des clauses du programme, dans le cas de la récursivité ; de plus, il mémorise à tout moment, pour chaque prédicat, les clauses définissant ce prédicat qui ont déjà été appliquées, pour éviter de les réutiliser dans la suite de la démonstration en cours. PROLOG/EX1 a ainsi plus de chances de faire aboutir une démonstration à un succès, dans des cas où PROLOG classique bouclerait ; d'autre part, en cas de succès, les démonstrations obtenues sont généralement plus courtes que celles qui auraient été générées par

PROLOG.

Dans le cas d'une démonstration en échec, PROLOG/EX1 va donner une tentative de résolution partielle, en ne s'arrêtant pas au premier fait en échec ; il note que ce fait est faux, mais le suppose vrai pour pouvoir continuer sa démonstration. Comme il utilise les mécanismes cités au-dessus, il obtient une démonstration sous conditions ayant les propriétés suivantes :

- longueur minimale (ce qui permet de limiter l'explosion combinatoire)

- points en échec signalés au niveau le plus global (ce qui permet aussi d'éviter l'explosion combinatoire)

Exemple :
soit le programme suivant :

chemin(x,y) V ¬ chemin(x,z) V ¬ chemin(z,y)
chemin(x,y) V ¬ vol(x,y)
vol(A,B)
vol(C,D)

A la question : y a-t-il un chemin de A à D ? - ou bien : pourquoi n'y a-t-il pas de chemin de A à D ?, PROLOG/EX1 répond par la démonstration suivante :

chemin(A,D)
si [vol(A,B)
   [chemin(B,D)

chemin(B,D)
si [vol(B,) ?
   [chemin(,D) ?

les points d'interrogation indiquant les faits en échec qui auraient <u>suffi</u> pour prouver : chemin(A,D), à partir de la base de clauses initiale.

En conclusion, l'explication fournie est un arbre de résolution sous hypothèses. Nous savons où la démonstration du but est en échec ; mais cette méthode ne met pas explicitement l'accent sur l'origine de l'échec (imaginons des démonstrations plus complexes ...).

Ce travail est à rapprocher de ceux de génération d'hypothèse. Nous le comparerons à celui de Pugin en I.3.4.2.

I.3.3 - Approche orientée concepteur de la base de connaissances

Il s'agit du debugging de la base de connaissances. Nous marquerons d'abord la différence entre cette approche et le debugging de logiciel au sens classique. Puis nous citerons deux exemples de systèmes dans cette optique, ayant l'interactivité comme caractéristique commune.

I.3.3.1 - Comparaison avec le debugging de programme en logique

Le but d'un outil dit "de debugging" est, de façon classique, de localiser une erreur (ou un échec) au cours de l'exécution d'un logiciel. Ensuite l'analyse de la cause de cet échec revient généralement à la charge de l'usager. Or c'est plutôt ce dernier aspect qui s'apparente au problème des explications négatives. Un système de génération d'explications négatives n'a donc rien à voir avec les debuggers traditionnels, qui ne sont que des traceurs plus ou moins sophistiqués, fonctionnant de façon interactive avec l'usager pour trouver l'origine de l'échec (/DUC 86/, /SHA 82/). Toutefois, actuellement certains outils tendent à réaliser aussi la fonction d'analyse de l'échec, donnant alors tout son sens au mot debugging (/BEL 87/).

D'autre part, dans le meilleur des cas, les outils de debugging s'expriment d'une façon qui ne peut être intelligible que par un programmeur. Or, il ne s'agit pas de parler d'instanciations et de noeuds d'arbre de résolution, mais de faire passer un message à un utilisateur qui n'est pas familier du langage de représentation interne des connaissances.

I.3.3.2 - Outils de modification de base de connaissances

En logique des propositions, l'exemple le plus connu est TEIRESIAS (/DAV 84/, /HAND/), travaillant sur la base de connaissances de MYCIN. Si la conclusion de MYCIN à un problème n'est pas celle attendue par l'utilisateur, TEIRESIAS va tenter de trouver d'où vient la différence. Pour cela, il va lui proposer la liste des règles qui auraient dû être ou ne pas être appliquées pour aboutir à la conclusion attendue ; l'usager précise alors si leurs premisses ont les valeurs que leur a accordées le système. S'il n'est pas d'accord, le processus est réitéré. S'il est d'accord, TEIRESIAS lui demande alors s'il ne manque pas une règle dans la base qui aurait permis de conclure à la valeur attendue. L'utilisateur propose alors une règle dans son propre vocabulaire ; par une suite d'itérations interactives, TEIRESIAS parviendra à exprimer cette règle avec son propre langage. Il s'agit donc de répondre à la question pourquoi-pas en terme d'acquisition de nouvelle connaissance opératoire. Pour ce faire, TEIRESIAS dispose des outils suivants :

- modèles des règles de la base de connaissances (rappelant les descripteurs de règles vus en I.3.2.1...)

- mémorisation de texte expliquant les raisons de l'introduction d'une règle dans la base par l'expert

- module de traitement de langage par "templates" (mots clés)

- mémorisation détaillée du mécanisme de résolution appliqué pour résoudre le problème ayant fait apparaître une erreur dans la base de connaissances.

Nous retrouvons là les caractéristiques des outils mentionnés en I.2.

En logique des prédicats, le système LKJ-ex a pour but de "debugger" la base de connaissances du système LKJ, formulée en clauses de Horn sans fait négatif (/MAZ 85/). Le problème est le même que dans

TEIRESIAS. Dans un premier temps, LKJ-ex recherche seul la conclusion de la règle manquante. Pour cela, il utilise des règles et des heuristiques, parmi lesquels :

1 - une règle ne peut être impuissante (i.e., qui aurait dû être appliquée mais ne l'a pas été) que si une de ses prémisses l'est.

2 - si une prémisse est impuissante, soit aucune règle n'est applicable, soit il s'agit d'un prédicat évaluable (erreur de donnée de problème.

3 - tant qu'une prémisse n'est pas terminale (i.e.: effaçable par un fait), il faut appliquer 1.

4 - si une règle a une prémisse non effaçable terminale, elle ne devait pas être appliquée ; il faut donc regarder les autres règles concluant au même terme.

Ainsi, la conclusion de la règle manquante pourrait être le premier terme rencontré tel que toutes les règles appliquées ont au moins une prémisse non effaçable terminale (si à chaque noeud-ou, a été choisie la "bonne" règle impuissante).

La deuxième étape consiste à fabriquer la règle manquante avec des heuristiques reposant sur la supposition que celle-ci ressemble à celles qui concluent sur un terme ayant le même prédicat, et sur l'hypothèse de monde fermé ; une règle contenant généralement trop de prémisses est alors proposée à l'utilisateur, qui la corrige ensuite de façon interactive. Si ce processus n'aboutit pas, on revient sur la dernière règle impuissante choisie à un noeud-ou, jusqu'à pouvoir relancer une exécution correcte sur la base de connaissances complétée par cette nouvelle règle.

Les limites de LKJ-ex sont donc liées au nombre plus ou moins important de règles existant déjà dans LKJ. D'autre part, la recherche de la conclusion de la règle manquante, reposant sur des heuristiques, ne mène pas de façon sûre directement au bon résultat. Malgré ses faiblesses et ses limites, ce travail montre une démarche intéressante

dans le contexte des clauses de Horn.

I.3.4 - Génération d'hypothèses

Un problème qui peut être considéré comme proche de celui des explications négatives est celui de la génération automatique d'hypothèses telles que, à partir d'une base de connaissances donnée, on puisse déduire un but donné : ces hypothèses manquantes peuvent être considérées comme une réponse possible dans le cadre des explications négatives. Nous mentionnerons deux travaux différents à ce sujet, dans le contexte de la logique des prédicats. Le premier a comme finalité première la génération d'hypothèses ; le point de vue du second est plutôt la résolution sous hypothèses : formellement, les problèmes sont identiques.

I.3.4.1 - Optique génération d'hypothèses

I.3.4.1.1 - Présentation de la méthode de Morgan

Le problème à résoudre est : soit un ensemble de formules C et un théorème à démontrer B ; trouver une formule H telle que $C \wedge H \vdash B$. H est donc un ensemble d'hypothèses à rajouter, suffisant pour conclure B à partir de C.

Pour résoudre ce problème, Morgan propose (/MOR 75/) une méthode à caractère inductif - de par son objectif - et cependant déductive.

Il travaille sur les formules transformées en f-clauses, ou mises sous forme normale prénexe disjonctive. C s'écrit donc :

$$C_1 \vee C_2 \vee \ldots \vee C_n$$

avec $\forall_i = 1,\ldots,n,\ C_i = \bigwedge_{j=1}^{n} C_{ij},\ C_{ij}$ f-clause

Toutes les variables quantifiées initialement universellement ont été transformées par la fonction de Skolémisation, et les variables demeurant dans les f-clauses sont donc implicitement quantifiées existentiellement.

A partir de là, il définit la f-résolution (f- car elle préserve le faux) comme duale à la résolution de ROBINSON :

si F_1 et F_2 sont deux expressions ne partageant pas de variable, si L_1 et L_2 sont deux littéraux de signes opposés de F_1 et F_2 tels que : $\exists\ \sigma$ unificateur / $\sigma\ L_1 = \neg / \sigma\ L_2$), alors : la f-résolvante f-R de F_1 et F_2 est $\sigma(F_1\text{-}L_1) \wedge \sigma(F_2\text{-}L_2)$ (où $F_1\text{-}L_1$ représente la f-clause obtenue en enlevant L_1 dans F_1). Il introduit également la notion de factorisation, comme en résolution classique. On note alors $F_1, F_2 \mid\!\!\!\vdash_F$ f-R. Et on a $F \mid\!\!\!\vdash_F R$ si et seulement si toute interprétation rendant F faux rend aussi R faux.

Morgan montre qu'une formule F est valide si et seulement si on peut générer [], la clause Vraie, en appliquant f-résolution sur F. D'autre part, en notant :

fC(E) la mise sous forme f-clausale de E

$fR^n(E)$ l'ensemble des f-résolvantes obtenues après n itérations de f-résolution à partir de E

si $\exists\ n /\ fC(E_2) \subseteq fR^n(fC(E_1))$ (i.e. $E_1 \mid\!\!\!\vdash_F E_2$),
alors on a $E_2 \vdash E_1$ au sens de la déduction classique.

En exploitant ces deux théorèmes, Morgan peut alors appliquer la f-résolution à la génération d'hypothèses. En effet, le problème initial s'écrit :
étant donnés C et B tels que $C \not\vdash B$,
trouver H telle que $C \wedge H \vdash B$
or $C \wedge H \vdash B$

$\Leftrightarrow\ H \vdash B \vee \neg C$

$\Leftrightarrow\ fC(B \vee \neg C) \mid\!\!\!\vdash_F fC(H)$

Morgan peut donc trouver un ensemble d'hypothèse H en

appliquant f-résolution. Pour cela, il utilise un algorithme d'unification étendu à la prise en compte de l'égalité entre des termes non variables, de noms différents (l'algorithme se terminant, car à chaque introduction d'égalité hypothétique entre termes, la substitution correspondante se fait en conservant le terme le plus court). A partir de fC(B V ¬C) on génère alors fC(H).

Le problème est ensuite, à partir de H' = fC(H), de retrouver les "bonnes" hypothèses. Morgan montre que toute f-clause de H', ou tout sous-ensemble de f-clauses de H', permet de générer une hypothèse possible (après re-écriture en forme clausale classique). Il utilise aussi un heuristique de "couverture" pour générer des hypothèses intéressantes ; cet heuristique s'intéresse plus particulièrement aux disjonctions des formules générées à partir de B seulement (et pas de ¬C)- une fois retranscrites sous forme normale classique - de façon à obtenir une expression ayant un sens le moins restrictif possible.

Cette méthode est particulièrement intéressante surtout si les connaissances initiales (C) étaient formulées en une conjonction de clauses ; en effet ¬C est alors une disjonction de conjonctions de littéraux, donc la transformation en f-clauses est assez simple : à une transformation implicite près sur le sens des variables, la négation d'une clause de C est une f-clause de fC(¬C).

Remarque : pour les explications négatives

Chaque hypothèse générée avec la méthode de Morgan constitue une condition suffisante pour permettre de déduire le but initial à partir des connaissances initiales. Si les H_i sont obtenues par f-résolution, on a :

$$\bigvee_i H_i, C \vdash B$$

En faisant l'hypothèse de monde fermé, et en supposant une stratégie de f-résolution <u>complète</u>, on aura donc

$$\bigwedge_i \neg H_i, C \vdash \neg B$$

$\bigwedge_i \neg H_i$ constitue alors une explication négative de l'échec de la démonstration de B à partir de C.

Notons que si la stratégie de f-résolution utilisée est complète, l'application de l'heuristique de couverture ne modifiera pas le contenu de l'explication négative.

(en effet, cela revenait à conserver $H_1 \vee H_2 \vee H_3$ en éliminant alors H_1, H_2 et H_3 ; or

$$\neg H_1 \wedge \neg H_2 \wedge \neg H_3 = \neg (H_1 \vee H_2 \vee H_3))$$

L'intérêt de la méthode de Morgan est donc qu'elle permet de générer directement des explications négatives intéressantes, au sens où l'on obtient effectivement et uniquement les causes possibles d'échec (par opposition par exemple au travail de Walker). Une de ses originalités est aussi que son principe est applicable à toute la logique des prédicats, sans restriction.

Dans le cas où l'on s'intéresse plus particulièrement à des réponses orientées vers le concepteur d'une base de connaissances, cette méthode à l'avantage d'être automatique ; en contrepartie, les nouvelles règles (ou hypothèses) qu'elle suggèrera ne porteront que sur des concepts existant déjà dans la base de formules initiales, alors que l'interactivité pouvait permettre d'introduire de nouveaux concepts via les nouvelles règles ajoutées.

I.3.4.1.2 - Exemple d'application de la f-résolution en génération d'hypothèses

Soit l'ensemble initial C de clauses de Horn suivant :

C_1 couple (x,y) $\vee \neg$ mêmesgoûts (x,y)

C_2 mêmesgoûts (x,y) $\vee \neg$ goûts (x,m,s) $\vee \neg$ goûts (y,m,s)

C_3 goûts (x,m,l) $\vee \neg$ aimemusique (x,m) $\vee \neg$ aimesport (x,s)

C_4 aimemusique (Paul,classique)

C_5 aimemusique (Marie,classique)

C_6 aimesport (Paul,foot)

C_7 aimesport (Paul,rugby)

C_8 aimesport (Marie,marche)

C_9 aimesport (Marie,escalade)

Soit B le but (indémontrable à partir de C) : couple (Paul,Marie)

Trouver H tel que $H \wedge C \vdash B$

En terme de f-résolution, cela se réécrit : $fC(B \vee \neg C) \mid\!\!\vdash_F fC(H)$

A - Transformation des clauses en f-clauses ("de Horn")

fC_1 ¬couple (x,y) ∧ mêmesgoûts (x,y)

fC_2 ¬mêmegoûts (x,y) ∧ goûts (x,m,s) ∧ goûts (y,m,s)

fC_3 ¬goûts (x,m,s) ∧ aimemusique (x,m) ∧ aimesport (x,s)

fC_4 ¬aimemusique (Paul,classique)

fC_5 ¬aimemusique (Marie,classique)

fC_6 ¬aimesport (Paul,foot)

fC_7 ¬aimesport (Paul,rugby)

fC_8 ¬aimesport (Marie,marche)

fC_9 ¬aimesport (Marie,escalade)

et fC (B) = couple (Paul,Marie)

B - Application de la f-résolution avec l'algorithme d'unification étendu à l'égalité entre termes non-variables :

(nous choisissons une LUSH f-résolution)

de fC (B) et fC_1 : fR_1 mêmesgoûts (Paul,Marie)

de fR_1 et fC_2 : fR_2 goûts (Paul,m,s) ∧ goûts (Marie,m,s)

de fR_2 et fC_3 : fR_3 aimemusique (Paul,m) ∧ aimesport (Paul,s) ∧ goûts (Marie,m,s)

de fR_3 et fC_4 : fR_4 aimesport (Paul,s) ∧ goûts (Marie,classique,s)

de fR_4 et fC_6 : fR_5 goûts (Marie, classique, foot)

de fR_5 et fC_3 : fR_6 aimemusique (Marie,classique) ∧ aime sport (Marie, foot)

de fR_6 et fC4 : fR_7 (Paul=Marie) ∧ aimesport (Marie,foot)

de fR_7 et fC_6 : fR_8 (Paul=Marie)

de fR_7 et fC_7 : fR_9 (Paul=Marie) ∧ (foot=rugby)

de fR_7 et fC_8 : fR_{10} (Paul=Marie) ∧ (foot-marche)

de fR_7 et fC_9 : fR_{11} (Paul=Marie)Λ (foot=escalade)

de fR_6 et fC_5 : fR_{12} aimesport (Marie,foot)

de fR_{12} et fC_6 : fR_{13} (Marie=Paul)

de fR12 et fC_7 : fR_{14} (Marie=Paul) Λ (foot=rugby)

de fR_{12} et fC_8 : fR_{15} (foot=marche)

de fR_{12} et fC_9 : fR_{16} (foot=escalade)

de fR_4 et fC_7 : fR_{17} goûts (Marie,classique,rugby)

etc....

C - Interprétation

Chacune des f-clauses générées par l'application de f-résolution est une hypothèse qui permettrait de conclure que Paul et Marie forment un couple.

En effet, si l'on retranscrit ces résultats sous forme normale, on retrouve :

fR_1' mêmesgoûts (Paul,Marie)

fR_2' ($\exists$m) ($\exists$s) goûts (Paul,m,s) Λ goûts (Marie,m,s)

fR_3' ($\exists$m) ($\exists$s) aimemusique (Paul,m) Λ aimesport (Paul,s) Λ goûts (Marie,m,s)

fR_4' ($\exists$s) aimesport (Paul,s) Λ aimesport (Marie,s)

etc...

Notons que certaines hypothèses se sumbsument entre elles ; par exemple, fR_{11} ==> fR_{16}, fR_7 ==> fR_8 etc... ou encore que certaines sont générées plusieurs fois (fR_8, fR_{13} ...).

Cet exemple montre bien qu'il est important de faire un tri parmi les formules générées pour ne pas tomber une explosion combinatoire. Pour cela, on peut par exemple : appliquer au moins les mesures suivantes :

- quand des hypothèses se sumbsument entre elles, ne conserver que les moins restrictives

  exemple : on élimine fR_{10}, on garde fR_{15}

  on élimine fR_{11}, on garde fR_{16}

on élimine fR_7, on garde fR_3 etc...

- les hypothèses conservées doivent être toutes différentes

- à partir des hypothèses conservées, on peut appliquer l'heuristique de couverture proposé par Morgan.
  exemple : on peut éliminer fR_{15} et fR_{16},
  et générer H_0 = (foot=marche) V (foot=escalade)
  de même, en continuant à appliquer f-résolution, on générerait
  H_1 = (rugby=marche) V (rugby=escalade)
  (Remarque : l'application de l'heuristique semble ici intéressant sur des hypothèses portant sur des égalités avec des constantes communes - introduites par l'algorithme d'unification étendu)

Enfin, la négation de chacune de ces hypothèses constitue bien un fragment d'explication négative, du fait que couple (Paul,Marie) est non démontrable (ou faux sous l'hypothèse de monde fermé) - certains fragments étant à des niveaux de détail différents.

En effet, on obtient par exemple :

$\neg fR_1'$ $\neg$ mêmesgoûts (Paul,Marie)
$\neg fR_4'$ $\exists$ s tq aimesport (Paul,s) $\wedge$ goûts (Marie,s)
$\neg H_1$ (rubgy $\neq$ marche) et (rugby $\neq$ escalade)
$\neg fR_{13}$ Marie $\neq$ Paul

Notons que les mesures d'élimination ou simplification d'hypothèses mentionnées ci-dessus sont compatibles avec la génération d'une explication négative.

I.3.4.2 - Optique résolution avec hypothèses

Ce travail est plus classique et concerne les bases de connaissances exprimées en clauses de Horn. Pugin (/PUG 87/) cherche à montrer un but, faux à priori sur la base de connaissances initiales, mais qui peut être vrai sous certaines hypothèses supplémentaire ; le

résultat cherché est donc de la forme (conjonction d'hypothèses, but).

D'autre part, des connaissances de niveau méta, énoncées par l'utilisateur, expriment l'interdiction de faire des hypothèses sur certains littéraux ; les hypothèses conditionnant le résultat doivent donc vérifier ces contraintes.

Pugin a écrit un interpréteur, inspiré de PROLOG, qui permet d'obtenir de tels résultats ; l'idée est de vérifier l'autorisation sur les hypothèses au fur et à mesure de leur création, en cours de résolution.

Nous n'avions pas connaissance de ce travail lorsque nous avons entrepris le nôtre. La démarche de PUGIN est intéressante ; comme chez MORGAN, les hypothèses conditionnant une résolution peuvent servir de point de départ pour générer des explications négatives concernant le but en question. Là aussi, un problème serait de déterminer les contraintes d'interdiction de certaines hypothèses qui permettraient d'obtenir des informations vraiment intéressantes pour l'utilisateur.

Comparaison entre les travaux de Pugin et de Walker (cf.I.3.2.2)

Ces deux travaux consistent à appliquer le principe de résolution en générant des hypothèses, lesquelles servent de réponses pour des explications négatives.

La particularité du travail de Walker est de générer des démonstrations les plus courtes, et de ne signaler l'échec qu'au niveau le plus élevé dans l'arbre de résolution ; c'est en ce sens qu'il est plus orienté "explications usager".

Par contre, le travail de PUGIN est à mi-chemin entre la génération d'hypothèse et le contrôle de types (ou application de contraintes sur les hypothèses susceptibles d'être générées pour fournir des réponses négatives).

Remarque :

Dans ces deux travaux, on fait l'hypothèse implicite que deux constantes de noms différents ont des valeurs différentes (hypothèse faite dans les interpréteurs PROLOG classiques). Ceci est à noter par rapport au travail de Morgan.

I.3.5 - Théorie des types pour un langage de type PROLOG

I.3.5.1 - Introduction

Le langage PROLOG a, par rapport aux langages de programmation classiques, la particularité de ne pas être typé. Un avantage de ceci réside dans la simplicité et l'uniformité de la définition de ce langage. En revanche, il est impossible de préciser la nature d'erreurs dues à des appels de littéraux sur des termes "incorrects" au point de vue type (selon une vision procédurale de PROLOG), ni avant ni pendant le lancement du processus de résolution.

Pour remédier à cette lacune, plusieurs travaux ont visé à définir une théorie des types pouvant permettre d'éviter ou d'expliquer des erreurs au cours d'une résolution, qui soit tout à fait compatible avec les concepts fondamentaux d'un langage manipulant des clauses de Horn comme PROLOG.

Les principales démarches de tels travaux sont de :

- donner une définition de la notion de Type, relative à un prédicat
- établir une théorie compatible avec cette définition, permettant de typer les littéraux ou les clauses
- donner un moyen pour montrer qu'un programme est - ou n'est pas - "bien typé" (i.e. : respecte - ou ne respecte pas - la théorie ci-dessus).

A partir de là, on peut obtenir des renseignements sur le

fonctionnement d'un programme, de façon statique (i.e. avant l'exécution de ce programme). Les interprétations en seront diverses, selon la théorie et la définition des types choisies.

De telles théories semblent donc présenter un intérêt pour la génération d'explications concernant un fait négatif ; en effet, le but est de savoir (à l'avance) si une résolution a des chances ou non d'aboutir à un succès, vis-à-vis des critères de type des arguments du littéral objet de la question.

Nous allons brièvement étudier deux travaux travaillant sur des théories complètement différentes, pour voir quelles diverses interprétations peuvent être données selon la théorie envisagée : celui de Mishra, d'une part, et celui de Mycroft et O'Keefe, d'autre part. Nous conclurons en comparant les intérêts de chaque théorie pour le problème des explications négatives.

I.3.5.2 - Une théorie des types "automatique" en PROLOG

Selon Mishra (/MIS 84/), le type d'un prédicat doit décrire de façon symbolique l'ensemble de tous les termes pouvant être des arguments de ce prédicat pour que le littéral correspondant puisse être effacé avec succès.

exemple :
soit le prédicat EDGE défini par

EDGE (A,B) ; EDGE (B,C) ; EDGE (C,D) ; EDGE (D,B)

le type du prédicat EDGE est ici

T(EDGE) = (A+B+C+D) x (B+C+D)

où x dénote une notion proche de "produit cartésien", et + dénote un OU.

Selon une telle définition du type T(P) d'un prédicat P, étant donné P(x), si x n'appartient pas à T(P) alors P(x) sera nécessairement en échec.

Une théorie des types basée sur cette définition s'applique à la classe des prédicats dont le type peut s'écrire à partir d'"arbres

clos réguliers". Intuitivement, la notion "d'arbre régulier" est à rapprocher de celle, plus classique, de "grammaire régulière" - à ceci près que l'on ne manipule pas des ensembles de chaînes de caractères appartenant à un alphabet donné, mais des ensembles d'arbres (ou termes) sur un alphabet donné. Un terme étant ici soit une constante, soit un symbole de fonction appliqué à un n-uple de termes, un "arbre clos régulier" (GRT) se définit donc formellement par les règles suivantes :

- l'arbre $\emptyset$ est un GRT
- toute constante est un GRT
- si $R_1,\ldots,R_n$ sont des GRT alors le n-uple $(R_1,\ldots,R_N)$ est un GRT
- si R est un GRT et f un symbole de fonction, alors f(R) est un GRT
- si T_1 et T_2 sont des GRT, alors $T_1 + T_2$ est un GRT
- si g(z) est un GRT, z étant une variable, alors z! g(z)

(i.e.: le plus petit ensemble X de termes tel que $X = \{x/g(x)\}$) est un GRT.

Par-dessus cette notion, Mishra introduit celle d'arbre régulier paramétré, correspondant à des ensembles de GRT, et contenant des <u>variables</u> comme types.

Pour la classe des prédicats considérée, il définit alors un jeu de règles d'inférence entre types, qui, appliqué à un ensemble de clauses définissant, sont un prédicat, génère simultanément :

- une définition du type du prédicat, T(P)

- un ensemble EI d'inéquations sur des GRT, à résoudre, tel que EI doit avoir une solution pour que la définition du type du prédicat soit correcte.

Mishra donne un algorithme de résolution des ensembles EI, et montre que si $D(P) = \{t/P(t)$ vrai,où t est un terme$\}$, alors $D(P) \subset T(P)$.

Un prédicat est alors dit être bien-typé si son ensemble EI a une solution, en termes d'arbres réguliers. Avec cette théorie, on a un résultat très intéressant : tout programme mal typé ne peut être exécuté avec succès.

Ce travail peut s'appliquer au problème des explications négatives. Soit $P(x_0...)$ un littéral vérifiant les hypothèses d'application de la théorie de Mishra. Si ce littéral est faux pour des raisons de type, on peut en avoir une explication en passant par les étapes α et B :

α - détermination du type T(P) si c'est possible ; si ça ne l'est pas, P est mal typé, donc $P(x_0,...)$ est faux, ceci ne dépendant pas de la valeur de x_0.

β - si P est bien typé, a-t-on : $x_0 \in T(P)$?. Si non, alors $x_0 \notin D(P)$, on sait donc pourquoi $P(x_0,...)$ est en échec. Si oui, cette procédure ne permet pas d'expliquer pourquoi $P(x_0,...)$ est en échec.

Cette méthode est automatique dans le sens où elle ne demande aucune déclaration supplémentaire (autre que celle des clauses du programme) de la part de l'utilisateur.

I.3.5.3 - Un schéma de types polymorphes en PROLOG

L'approche de Mycroft et O'Keefe (/MYC 84/) consiste à considérer la spécification du type d'un prédicat, ou d'une fonction, comme une restriction sur les arguments de ce prédicat ou de cette fonction.

exemple :

soit le prédicat PLUS (X,Y,Z) signifiant que Z est la somme de X et de Y. Une spécification du type de PLUS pourrait être que X,Y et Z sont des entiers.

Si les arguments d'un prédicat ne sont pas du type de ce prédicat, cela ne signifie pas que le littéral correspondant soit nécessairement en échec. Par contre, le littéral n'a pas de sens (sémantiquement).

La théorie bâtie par les deux auteurs ne s'applique plus à une classe restreinte de prédicats. On parle ici de types polymorphes ; en particulier, on peut avoir des variables de types quelconques.

Les types sur lesquels on travaille doivent s'écrire selon la grammaire suivante : type : = Tvar|Tcons(Type*), où Tvar est un type variable, et Tcons un constructeur de types (ex:liste).
exemples : liste (α), liste (liste(entier)).

Etant donné alors :

- une clause Cl déterminant un ensemble ES de symboles (de variables, fonctions, prédicats)
- les types des symboles de ES donnés par l'usager ($\overline{ES}$)

on écrit $\overline{Cl}$, clause typée de Cl, comportant en elle-même les informations des types des éléments de ES.

Exemple

Cl : app(cons(A,L),M,cons(A,N) $\vee \neg$ app(L,M,N) .

ES : {A,L,M,N,app,cons} .

$\overline{ES}$: $\{A^{\alpha}, L^{\tau}, M^{\tau}, N^{\tau}, app^{\tau,\tau,\tau}, cons^{(\alpha,\tau)\rightarrow\tau}\}$

où τ = liste(α)

$\overline{Cl}$: $app(cons(A^{\alpha},L^{\tau})^{\tau},M^{\tau},cons(A^{\alpha},N^{\tau})^{\tau}) \vee \neg app(L^{\tau},M^{\tau},N^{\tau})$

Une clause Cl est alors dite bien-typée selon $\overline{ES}$ si $\overline{Cl}$ satisfait à un ensemble de conditions défini par les auteurs(/MYC 84/). Un programme est ensuite dit bien typé selon un ensemble $\overline{ES}$ si chacune de ses clauses l'est, et si le but à montrer à partir de ces clauses l'est également. Si un programme est bien typé, cela signifie que les types sont préservés par la résolution, quelle que soit la stratégie de résolution ; cela ne donne pas de renseignement sur le succès ou l'échec de la résolution ; par contre un programme mal typé risque d'occasionner des erreurs de types (donc sera peut-être sémantiquement faux).

Concernant les explications négatives d'un fait $P(x_0,...)$ relativement à une base de clauses, les informations que peut apporter

cette théorie des types sont moins "fortes" qu'avec la théorie précédente ; on peut en effet savoir si le programme est ou n'est pas bien typé ; s'il n'est pas bien typé, le programme n'est pas correct au niveau sémantique. Toutefois, dans tous les cas, cette théorie ne peut pas apporter de réponse sur le succès ou l'échec de la résolution de $P(x_0,...)$.

I.3.5.4 - Conclusion sur l'intérêt des théories des types pour les explications négatives

Les deux théories que nous avons choisi de présenter sont très différentes sur de nombreux points, même si les démarches sont parallèles.

Tout d'abord, la définition et l'interprétation du type d'un prédicat sont propres à chaque auteur : Mishra travaille sur les valeurs possibles des arguments d'un prédicat, le type d'un prédicat étant un sous-ensemble de ces valeurs possibles. Mycroft et O'Keefe s'intéressent au problème de la préservation des types au cours de la résolution. Disons que si dans le premier cas on s'intéresse plutôt à l'aspect valeur, dans le second on s'intéresse plutôt à l'aspect sémantique. Cela se traduit par ailleurs dans les conclusions de chacun. Pour Mishra, un programme mal typé ne peut qu'échouer ; dans l'autre cas, on affirme seulement que, avec un programme bien typé, les résolvantes créées au cours de la résolution préservant les restrictions associées aux types. Dans les deux cas, l'application de la théorie des types permet d'empêcher, avant le lancement de la résolution - i.e. de façon statique -, qu'un programme ne soit exécuté (par risque d'erreur).

Les deux travaux diffèrent encore sur d'autres points importants. La particularité du premier est de ne nécessiter aucune déclaration supplémentaire de la part de l'usager. Par contre, la classe de prédicats à laquelle s'applique la théorie est plus générale dans le second cas. De part et d'autre, des problèmes théoriques, non mentionnés ici, demeurent encore ouverts.

Nous avons montré que, selon la théorie utilisée, l'intérêt de cette approche pour le problème des explications négatives est plus ou

moins fort. En particulier, si l'on montre que le prédicat objet de la question, ou le programme, sont mal typés, cela répond bien à notre problème avec la théorie de Mishra, mais pas avec celle de Mycroft et O'Keefe ; dans ce dernier cas l'information obtenue est intéressante mais ne permet pas de conclure quant au succès ou à l'échec de la démontration.

Enfin, dans le cas où une théorie des types peut répondre à notre problème, une différence fondamentale avec les approches précédentes est que l'on travaille en amont de la résolution de problème.

I.4 - CONCLUSION

Nous avons donné un aperçu des principales techniques utilisées pour générer des explications sur les systèmes à base de connaissances. En réalité, la majeure partie des travaux présentés se limitent plutôt à un commentaire "intelligent" du raisonnement du système en s'appuyant sur les seules connaissances utilisées en résolution de problèmes. Les autres essayent d'atteindre vraiment un niveau d'explications paraissant plus naturelles à l'utilisateur, reflétant moins les séquences de code mises en oeuvre pendant la résolution ; pour cela, des connaissances supplémentaires, voire une nouvelle architecture des systèmes à base de connaissances, semblent nécessaires.

Par ailleurs, nous avons trouvé relativement peu de travaux concernant aussi de véritables explications négatives. Le problème est plus délicat, d'une part, parce que la réponse à y donner peut être très variable selon le type d'utilisateur, d'autre part, parce qu'il s'agit d'un raisonnement qui n'a pas été suivi (et lequel ?). Aussi, les travaux dans ce but se limitent en général à des opérations sur la résolution. Ils restent alors souvent liés aux problèmes de contrôle (traitement de l'échec, backtracking "intelligent", etc...), eux-mêmes pas toujours bien résolus. Pour l'instant, on ne vise généralement que le niveau "bons commentaires", et non pas véritable explication.

Toutefois, nous avons noté certaines études dont les

applications avouées sont un peu différentes, comme le debugging (au sens complet du terme), la génération d'hypothèse ou la résolution sous hypothèses, ou encore certaines théorie des types ; elles peuvent donner des résultats intéressants et former une base de solution au problème des explications négatives.

II - RESOLUTION DU PROBLEME DES EXPLICATIONS NEGATIVES

Dans cette seconde partie, nous préciserons d'abord quel genre d'explications négatives nous voulons produire. Ensuite, nous présenterons une première ébauche de solution qui s'est avérée inintéressante en pratique ; nous montrerons sur un exemple concret pourquoi cette voie a conduit à une impasse, et ceci nous permettra de justifier l'approche que nous avons finalement adoptée. Nous présentons donc une méthode s'appuyant sur la représentation de la résolution d'un problème par un arbre ET-OU,en restant à un niveau théorique sans rentrer dans les problèmes d'implémentation ; pour éclaicir l'exposé, nous donnons divers exemples - entre autres, nous reprendrons celui déjà utilisé comme application de la f-résolution de Morgan (cf. I.3.4), et de notre première ébauche de solution, pour donner un élément de comparaison entre ces diverses techniques.

II.1 - DEFINITION ET CONTEXTE DE RESOLUTION DE NOTRE PROBLEME

Nous présenterons dans un premier temps l'optique générale de notre travail, en précisant le type d'informations que nous voulons utiliser et les raisons de ce choix, et les caractéristiques de la réponse que nous cherchons à obtenir. Nous présenterons enfin nos principales hypothèses de travail.

II.1.1 - Informations utilisées

Un de nos objectifs est de générer nos réponses de façon dynamique, sans entraver le concept d'évolutivité lié à la base de connaissances. Nous excluons donc toute technique consistant à faire appel à des informations déjà emmagasinées destinées à générer des explications relatives aux éléments constituant la base de connaissances : la mise à jour de la base "pour explications" doit se faire automatiquement et systématiquement lors de celle de la base de connaissances ; d'autre part, nous désirons que nos réponses intègrent entièrement les caractéristiques du problème particulier de l'utilisateur. Nos réponses seront donc générées, par hypothèse, à

partir des informations qui servaient à faire de la résolution de problème. Cela implique qu'elles seront liées au mode de représentation des informations choisi par le concepteur de la base de connaissances.

Toutefois, nous pouvons éventuellement utiliser des informations portant sur celles de la base de connaissances, mais qui ne sont pas utilisées en résolution de problème. Ces informations peuvent nous permettre de contrôler les réponses obtenues à partir de la base de connaissances vis-à-vis de certains critères.

D'autre part, nous voulons que toutes les informations utilisées soient représentées par un formalisme unique, celui de la logique des prédicats restreinte aux clauses de Horn, pour bénéficier à la fois de la puissance de ce langage et des résultats théoriques acquis sur ce langage.

II.1.2 - Caractéristiques de la solution recherchée

Premièrement, nous désirons raisonner sur l'état de la base de connaissances au moment où l'usager pose sa question au système ; nous essaierons d'être, à partir de là, le plus autonome possible vis-à-vis de l'usager ; ce critère nous permettra d'être objectif et d'examiner toutes les raisons possibles qui peuvent rendre $P(x_0,...)$ indémontrable. Pendant la recherche de ces réponses, l'usager ne devra donc pas être sollicité.

Ensuite, la solution que nous cherchons devra être orientée vers l'usager, dans les sens suivants.

Tout d'abord, nous cherchons à faire porter la responsabilité de l'échec sur $P(x_0,...)$ par l'usager du système, plutôt que par le concepteur de la base de connaissances (i.e. l'expert, dans un système expert) ; autrement dit, on suppose l'échec lié aux données mêmes fournies par l'usager auparavant au système, et nos réponses iront dans ce sens. Le cas d'exception est celui où l'échec est inhérent à la base de connaissances, indépendamment de toute résolution de problème, auquel cas cela doit être expliqué à l'usager.

exemple :

(notations : dans toute la suite, les arguments commençant par des majuscules représentent des constantes, ceux commençant par des minuscules représentent des variables).

Soient les connaissances énoncées initialement par un expert :

$$\begin{cases} \text{traite}(m,a) \vee \neg\, \text{méthode}(m) \vee \neg\, \text{adaptée}(m,a) & (1) \\ \text{méthode}(M_1) & (2) \\ \text{adaptée}(M_1,a) \vee \neg\, \text{symétrique}(a) \vee \neg\, \text{bande}(a) & (3) \end{cases}$$

exprimant les conditions de choix d'une méthode m pour traiter une matrice a,
et les données d'un problème de l'usager, portant sur une matrice A :

$$\begin{cases} \text{triangulaire}(A) \\ \text{bien-conditionnée}(A) \end{cases}$$

Si l'usager cherche à savoir pourquoi M_1 ne peut pas traiter sa matrice A, notre approche vise à lui répondre que cela est dû au moins au fait que A n'est pas symétrique. Une autre approche vise à remettre en cause des connaissances de la base, en supposant qu'il manque peut-être une règle permettant de conclure au choix de M_1 avec d'autres propriétés sur la matrice, ou encore en supposant que la règle (3) est peut-être fausse ; cette approche remet en cause plutôt l'expert.

Supposons maintenant que l'usager cherche à savoir pourquoi le système ne peut pas montrer que M_2 traite A ; cette fois, l'échec est inhérent aux connaissances énoncées par l'expert. Il faut donc signaler à l'usager que le système ne connait pas la méthode M_2.

Dans le cas où l'échec est dû aux données de l'usager et au contenu de la base initiale, notre réponse ne détaillera que les premières raisons de l'échec, à priori (s'il n'y a pas de raison de les

éliminer).

Dans certains cas, il sera même intéressant de suggérer par notre réponse des données que l'usager a pu oublier de préciser au système, et qui auraient pu peut-être permettre de montrer $P(x_0,...)$. En fait, nous aimerions savoir d'une part où le système est en échec, mais aussi pourquoi il est en échec - i.e., les diverses origines possibles de cet échec.

D'autre part, puisque nous nous adressons à l'usager (et non au concepteur de la base de connaissances initiales), nous devons lui fournir des réponses compréhensibles. Par exemple, ne devront pas figurer dans une réponse des éléments de type utilitaire (au sens de la programmation).

Nous nous attacherons surtout à donner des conditions suffisantes (dépendant de l'usager, s'il y en a) d'échec de $P(x_0,...)$. Des conditions nécessaires et suffisantes (sous l'hypothèse de monde fermé) sont intéressantes en théorie, mais moins importantes pour l'usager en pratique.

Ainsi, dans l'exemple précédent :

- le fait que A ne soit pas déclarée comme étant symétrique suffit à expliquer, selon les informations contenues dans la base initiale, pourquoi on ne peut pas montrer que M_1 traite A.

- toutefois, pour pouvoir affirmer que M_1 traite A, A doit être symétrique et bande.

Par ailleurs, nous essaierons de présenter nos explications structuréees par niveaux de détail, plus ou moins fins selon le désir de l'utilisateur, pour pouvoir donner des renseignements clairs et concis.

Enfin, d'un point de vue technique, nous cherchons à ne pas nous éloigner de la formulation classique des connaissances en clauses, pour que notre travail soit directement applicable à la plupart des systèmes à bases de connaissances. Plus précisément, nous voulons

travailler en résolution classique.

II.1.3 - Hypothèses de travail

- la base de connaissances est formalisée en clauses de Horn régulières.
- cet ensemble de clauses forme un programme qui ne boucle pas.
- la stratégie de résolution qui a conclu à l'échec sur $P(x_0,...)$ était complète - sans quoi le problème n'aurait pas de sens.
- le problème portant sur $P(x_0,...)$, P appartient au langage de la base de connaissances (il existe au moins une clause ayant pour littéral positif P(x,...)).
- $P(x_0,...)$ n'est pas inconsistant avec la base de connaissances : cela est assuré par le fait qu'elle ne contient pas de clause non régulière.
- hypothèse de monde fermé.
- deux constantes de noms différents ont des valeurs différentes ; de même, des fonctions dont les symboles sont différents sont supposées différentes.

II.2 - UNE PREMIERE APPROCHE, A MI-CHEMIN ENTRE LA GENERATION D'HYPOTHESES ET LE CONTROLE DE TYPE

Une première ébauche de solution a consisté à donner une réponse en deux temps à la question pourquoi-pas (/SAU 87/). Dans un premier temps, nous présenterons brièvement les idées générales mises en oeuvre dans cette solution. Ensuite, nous la critiquerons, pour expliquer pourquoi nous n'avons pas poursuivi dans cette voie, et justifier celle que nous avons finalement choisie.

Cette approche n'utilise pas l'hypothèse de monde fermé.

II.2.1 - Présentation générale de la méthode

Dans la suite, nous noterons :

BdC la base de connaissances à un instant donné (clauses de Horn)
BdR la base initiale des connaissances, exprimées par le concepteur

BdF la base des données de l'utilisateur, relative au problème pour lequel il attendait une solution différente $P(x_0,...)$ que celle proposée par le système (clauses unitaires)

(on a BdC = BdR U BdF)

BdnF un ensemble d'hypothèses (clauses de Horn)

BdCt un ensemble de clauses de Horn, "contraintes", qui ne seront pas utilisées en résolution de problème, mais seulement en génération d'explications.

L'idée était de procéder en deux grandes étapes ; dans un premier temps, on génère des hypothèses "intéressantes" pour l'usager, uniquement à partir des informations utilisées en résolution de problème (BdC) ; dans un second temps, on élimine les hypothèses violant des contraintes de BdCt. Les hypothèses restantes permettront de générer des explications négatives.

II.2.1.1 - Terminologie adoptée

prédicat usager (resp. littéral) :

les littéraux constitués à partir d'un tel prédicat ne peuvent être effacés que par l'usager ; ils correspondent aux données de problème, et définissent donc des clauses unitaires.

(exemple dans Pélagie : bande, symétrique...)

Un prédicat usager P s'écrira par exemple us-P'

prédicat significatif (resp. littéral)

le symbole d'un tel prédicat a une sémantique pour l'usager.

Cela s'oppose aux prédicats introduite pour des commodités de programmation (ex: prédicats utilitaires, de calcul, etc...). On suppose que le système sait quels sont les prédicats significatifs.

Remarque :

Un prédicat usager est nécessairement significatif.

On notera :

ppas $P(x_0,...)$: pourquoi $P(x_0,...)$ est-il indémontrable sur BdC ?

$\sim P(x_0,...)$: BdC $\not\vdash P(x_0,...)$

II.2.1.2 - Génération d'un ensemble de faits manquants

L'ensemble initial de clauses de BdC étant consistant (puisqu'il ne comporte que des clauses régulières), on cherche des clauses Cl telles que BdC U Cl U $\neg P(x_0,...)$ soit inconsistant, en raisonnant par réfutation. Si l'on lance une réfutation à partir de $\neg P(x_0,...)$ et de BdC, en appliquant le principe de résolution de Robinson (/ROB 65/) avec une stratégie d'exploration complète, toute résolvante générée est par hypothèse en échec puisque $P(x_0,...)$ est indémontrable.

$$\text{Soit } R = \bigvee_{i=1}^{MR} \neg P_i(xi_1, xi_2, ...)$$

Pour que la démonstration sur $P(x_0,...)$ soit un succès, il suffit que soit vraie en axiome la formule

$$FR = (\forall_i = 1...,MR)\ \exists(yi_1\ yi_2...)\ tq \bigwedge_{i=1}^{MR} P_i(yi_1, yi_2...)$$

telle que

$\forall_i = 1...MR$, $P_i(yi_1, yi_2, ...)$ et $P_i(xi_1, xi_2...)$ soient unifiables.

FR peut être vu comme un ensemble de clauses de Horn unitaires. Donc à priori, toute hypothèse formée par la négation d'une résolvante constitue une condition suffisante pour pouvoir montrer $P(x_0,...)$ à partir de BdC.

Bien sûr, si l'on considère toutes les hypothèses ainsi générées, on tombe dans une explosion combinatoire. Il faut donc opérer une première sélection des formules susceptibles d'intéresser l'usager.

On peut donc appliquer par exemple les mesures suivantes :

- en distinguant les prédicats "significatifs" (dont le nom traduit clairement un concept lié au domaine, par opposition aux prédicats de calcul par exemple), on peut décider de ne retenir que les

résolvantes dont tous les prédicats sont significatifs.

- si l'on a utilisé par exemple la LUSH résolution, on peut décider de ne retenir que

(1) les résolvantes en échec terminal, c'est-à-dire dont le premier littéral ne peut s'effacer par aucune clause

(2) les résolvantes dont le premier littéral est un littéral-usager (i.e. ne pouvant être effacé que grâce à des données de l'usager)

(3) ces deux dernières mesures peuvent être combinées comme suit : si le premier littéral d'une résolvante en échec terminal porte sur certaines variables instanciées par l'usager, on peut décider de ne s'intéresser qu'aux résolvantes de type(2) dont le premier littéral a permis d'instancier ces variables.
etc... les mesures à appliquer ne pouvant être que de type syntaxique.

A l'issue de cette opération, nous avons généré un ensemble BdnF d'hypothèses, ou de faits manquants.

II.2.1.3 - Réduction de l'ensemble des faits manquants

On suppose que le concepteur de BdC a exprimé un ensemble de contraintes sur les littéraux, BdCt, tel que BdCt U BdC soit consistant. Il s'agit donc d'éliminer les éléments f de BdnF, tels que f U BdC U BdCt soit inconsistant.

On dispose de deux types de contraintes : celles s'exprimant par des clauses régulières (type T1), et celles s'exprimant par des clauses non régulières (type T2) ; ces dernières sont en fait des informations négatives, pouvant par exemple exprimer l'impossibilité d'avoir simultanément certaines relations entre des objets donnés. Ces contraintes de type T2 peuvent être interprétées comme la négation d'un théorème à déduire par réfutation d'un ensemble de clauses donné.

On peut alors lancer des réfutations à partir des éléments de BdCt de type T2 ; BdC U BdCt étant consistant, la seule façon de faire

aboutir ces réfutations est d'utiliser des éléments de BdnF comme clauses parents. On éliminera donc de BdnF chaque hypothèse ayant permis de faire aboutir une réfutation à la clause vide.

II.2.1.4 - Problèmes stratégiques liés à la méthode

La construction de BdnF étant coûteuse, il faut essayer d'exploiter les contraintes dès que possible, et en fait panacher les deux opérations précédentes, en :

- lançant d'abord des réfutations sur des contraintes de type T1 ayant P(x,...) comme un de leurs littéraux, si possible

- si P(x,...) n'est jamais apparu dans une résolvante d'une réfutation, construire BdnF

- lancer des réfutations en priorité sur les contraintes de type T2 ayant des littéraux semblables à ceux d'éléments de BdnF. Ceci n'est qu'une ébauche des problèmes à résoudre en réalité.

Remarques

Ce travail présente beaucoup de similitudes, dans ses idées de base, avec celui de Pugin (/PUG 87/), décrit en I3. Cependant, ils ont été développés en parallèle, sans connaissance l'un de l'autre.

Au point de vue stratégie, une différence importante est que dans /PUG 87/, la satisfaction des contraintes est vérifiée au fur et à mesure de la création des hypothèses ; une autre différence est que Pugin ne crée que les hypothèses dont il a réellement besoin, alors que nos hypothèses sont plus larges : ceci sera l'objet d'une des critiques opposées à notre méthode dans le paragraphe suivant.

D'autre part, la technique mise en oeuvre pour la génération des faits manquants est semblable, dans ses idées de base, à celle développée dans /CHO 87/ et /PAS 87/ pour résoudre le problème de l'interrogation intensionnelle d'une base de règles dans le cadre de base de données déductibles ; ce sont principalement les

caractérisations des résolvantes "intéressantes" qui diffèrent.

II.2.2 - Critique de la méthode

II.2.2.1 - *Exemple*

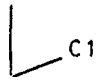
Cet exemple ne montrera que l'application de la partie génération d'hypothèses, pour servir de référence à la critique qui suivra.

BdR
- C_1 couple (x,y) V ¬ mêmesgoûts (x,y)
- C_2 mêmesgoûts (x,y) V ¬ goûts (x,m,s) V ¬ goûts (y,m,s)
- C_3 goûts (x,y) V ¬ aimemusique (x,m) V ¬ aimesport (x,s)

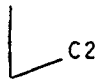
BdF
- C_4 aimemusique (Paul, classique)
- C_5 aimemusique (Marie, classique)
- C_6 aimesport (Paul, foot)
- C_7 aimesport (Marie, marche)

Question : Pourquoi-pas couple(Paul,Marie) ?
La réfutation sur ¬ couple(Paul, Marie) avec LUSH résolution est :
(notations : les clauses parents dont les noms sont encadrés correspondent à des clauses de BdF - ou littéraux - usager).

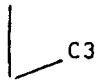
R0 ¬couple (Paul, Marie)



R1 ¬mêmesgoûts (Paul,Marie)



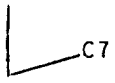
R2 ¬goûts(Paul,m,s) V ¬goûts(Marie,m,s)



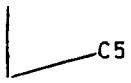
R3 ¬aimemusique(Paul,m) V ¬aimesport(Paul,s)
V ¬goûts (Marie,ms)

R4 ¬aimesport(Paul,s) V ¬goûts(Marie, classique,s)

R5 ¬goûts(Marie, classique, foot)



R6 ¬aimemusique(Marie, classique) V ¬aimesport(Marie, foot)



R7 ¬aimesport(Marie, foot) ←——— échec terminal

L'ensemble complet des négations des résolvantes nous donne :
{aimesport(Marie,foot), aimemusique(Marie,classique) ∧ aimesport(Marie,foot),
goûts(Marie,classique,foot),
(∃s) tq aimesport (Paul,s) ∧ goûts (Marie,classique,s),
(∃m) (∃s) tq aimemusique (Paul,m) ∧ aimesport (Paul,s) ∧ goûts (Marie,m,s),
(∃m) (∃s) tq goûts (Paul,m,s) ∧ goûts (Marie,m,s),
mêmesgoûts (Paul,Marie)

Les mesures de réduction nous font retenir seulement :

{aimesport(Marie,foot), (∃s) tq aimesport(Paul,s) ∧ goûts(Marie,classique,s)}

Or on aurait aimé retenir aussi au moins ¬R1 et ¬R2.

II.2.2.2 - Critiques à partir de l'exemple

Elles portent sur la construction des hypothèses principalement.

En effet, l'exemple simple présenté ci-dessus montre plusieurs défauts, sur un cas concret.

Le premier ensemble généré nous montre que prendre la négation d'une résolvante ne permet pas de mettre directement l'accent sur la cause de l'échec, dans la plupart des cas.

exemple : (∃m) (∃s) aimemusique (Paul,m) ∧ aimesport (Paul,s) ∧ goûts (Marie,m,s)

ici, le littéral aimemusique (Paul,m) n'apporte rien d'intéressant, car il est effaçable et ne participe pas du tout à l'échec.

Nos mesures de réduction sont trop abruptes, car elles éliminent des littéraux exprimant des informations moins détaillées, qui pourraient constituer un premier niveau d'explication négative.

En fait, avec la stratégie que nous avions d'abord choisie, nous avons perdu des informations.

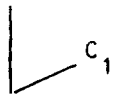
A - Profondeur de détail des informations

Nous ne voyons plus apparaître les littéraux "ancêtres" d'un littéral donné. Or, c'est ainsi que nous pourrions récupérer les négations de R_1, R_2, et R_5, à partir de celle de R_7.

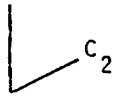
Pour remédier à cela, on peut introduire cette information dans l'arbre des résolvantes, en gardant trace des littéraux ancêtres déjà effacés (principe de la OL résolution, /CHA 73/).

On obtiendrait alors l'arbre suivant :

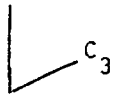
R_0' ¬couple (Paul,Marie)



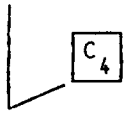
R_1' ¬mêmesgoûts (Paul,Marie) [couple (Paul,Marie)]



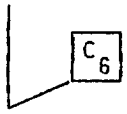
R_2' ¬goûts (Paul,m,s) V ¬goûts (Marie,m,s) [mêmesgoûts (Paul,Marie)]
[couple (Paul, Marie)]



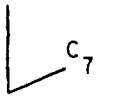
R_3' ¬aimemusique (Paul,m) V ¬aimesport (Paul,s) [goûts (Paul,m,s)]
V goûts (Marie,m,s) [mêmesgoûts (Paul,Marie)] [couple (Paul,Marie)]



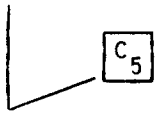
R_4' ¬aimesport (Paul,s) [goûts (Paul,classique,s)] V ¬ goûts (Marie,classique,
[mêmesgoûts (Paul,Marie)] [couple (Paul,Marie)]



R_5' ¬goûts (Marie,classique,foot) [mêmesgoûts (Paul,Marie)] [couple (Paul,Marie



R_6' ¬aimemusique (Marie,classique) V ¬aimesport (Marie,foot)
[goûts (Marie,classique,foot)] [mêmesgoûts (Paul,Marie)]
[couple (Paul, Marie)]



R_7' ¬aimesport (Marie,foot) [goûts (Marie,classique,foot)]
[mêmesgoûts (Paul,Marie)] [couple (Paul,Marie)]

Cette fois, on peut bien obtenir des réponses par niveau de détail plus ou moins fin à partir des résolvantes retenues, il suffit alors de faire attention à ne pas générer plusieurs fois la même réponse.

exemple :

à partir de R_7' : aimesport (Marie,foot)
goûts (Marie,classique,foot)
mêmesgoûts (Paul,Marie)

à partir de R_4' ; (∃s) aimesport (Paul,s) ∧ goûts (Marie,classique,s)
(∃s) goûts (Paul,classique,s) ∧
goûts (Marie,classique,s)

Donc on peut arriver à produire des informations intéressantes par niveau de détail, à condition de savoir éliminer les littéraux éventuels non concernés par les sources de l'échec (ce qui doit pouvoir se faire de façon acrobatique, via un algorithme d'unification assez sophistiqué).

B - Structures OU de l'arbre des résolvantes

Dans l'exemple précédent, supposons de plus que Paul aime le sport course ; cela engendre des hypothèses supplémentaires. Or la structure OU des clauses est complètement perdue avec une représentation par résolvantes. Il reste donc assez difficile de produire les réponses par niveau de détail dans le cas général, par cette méthode.

II.2.2.3 - *Conclusion*

Le problème principal dans cette approche, est lié au désir de produire des explications structurées par niveaux de détail, et ne faisant apparaître que des informations "utiles".

Or, en ne disposant que des résolvantes, nous perdons l'information qui nous permettrait de répondre selon nos critères.

C'est pourquoi nous avons abandonné cette voie en pratique. Nous nous sommes donc orientés vers un formalisme qui nous permet de conserver et retrouver directement les informations de structuration ET-OU contenues dans les clauses de la base de connaissances initiale.

II.3 - UNE METHODE DE GENERATION D'EXPLICATIONS NEGATIVES STRUCTUREES, BASEE SUR LE PRINCIPE DE RESOLUTION

II.3.1 - Motivations

Notre problème des explications négatives s'est posé à partir de la réalisation d'un système qui utilisait le principe classique de résolution. C'est pourquoi nous avons essayé de trouver une solution basée sur ce même principe, par opposition par exemple à l'approche de Morgan en génération d'hypothèses.

Notre première tentative de résolution de ce problème (cf. paragraphe précédent) a bien mis en évidence :

- le besoin de structurer les informations à présenter à l'utilisateur, par exemple par niveaux de détail
- la nécessité d'éviter de rentrer dans le détail de toutes les tentatives échouées entreprises au cours de la résolution de $P(x_0,...)$ (objet du pourquoi-pas)
- et surtout, de l'intérêt de disposer des informations de type ET-OU, permettant de structurer les clauses du programme, et donc les différentes étapes de la résolution qui a échoué.

Nous avons donc été amenés à utiliser un arbre ET/OU pour représenter la tentative de résolution de $P(x_0,...)$, structure nous permettant de répondre aux trois points précédents.

Dans un premier temps, nous montrerons qu'il est possible de bâtir une méthode, basée sur ce formalisme, et répondant aux caractéristiques souhaitées en II.I.2, dans le cas particulier de la logique des propositions. Ensuite, nous généraliserons le processus, pour prendre en compte la logique des prédicats.

Signalons d'ores et déjà que les problèmes d'implémentation n'ont pas été abordés.

II.3.2 - Cas particulier de la logique des propositions

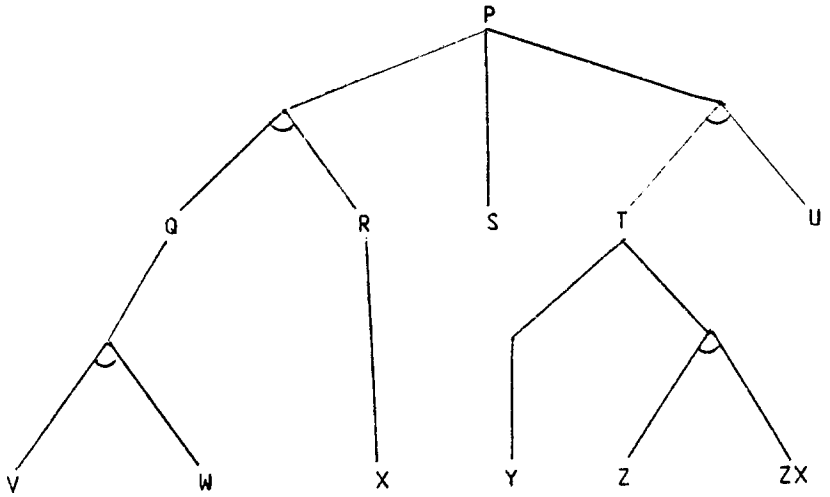
Dans ce premier paragraphe, nous allons montrer que la formalisation de la résolution d'un problème P, à partir de règles en logique d'ordre 0, en un arbre ET-OU (/FAR 82/), permet de générer aisément des explications négatives structurées par niveau de détail.

II.3.2.1 - Principe général de la méthode

Soit un arbre ET-OU de racine P, fini par hypothèse (selon l'hypothèse d'une stratégie de résolution complète). Un tel arbre définit une expression logique que l'on peut écrire de multiples façons : soit en fonction uniquement des feuilles de l'arbre, soit en fonction de ses noeuds.

exemple :

soit l'arbre suivant : (1)



où l'on note un noeud OU et un noeud ET. Cet arbre définit l'expression logique suivante.

$$(Q \wedge R) \vee S \vee (T \wedge U)$$

qui peut s'écrire également, en développant les noeuds R et T

$$L = (Q \wedge((V \wedge W) \vee X)) \vee S \vee ((Y \vee (Z \wedge ZX)) \wedge U)$$

Cette expression logique L peut s'interpréter comme suit : pour que P soit vrai, il suffit que L soit vraie. Si elle est fausse, on ne peut pas conclure à priori sur P.

Si, de plus, nous supposons que P est défini entièrement par l'arbre donné, on a alors équivalence entre P et L, donc $\neg L \Rightarrow \neg P$(2). Nous ferons cette hypothèse dans tout ce qui suit.

L'idée est alors de chercher une expression logique L' telle que $L \Rightarrow L'$, et L' fausse. D'après (2) nous aurons donc $P \Rightarrow L$. $\neg L'$ permettra donc d'expliquer pourquoi P est en échec (condition suffisante d'échec sur P).

Or, nous supposons connues les valeurs de vérité associées aux feuilles de l'arbre ; d'autre part, sachant que :

- un noeud ET est en échec s'il existe au moins un fils en échec de ce noeud, et est en succès si tous ses fils sont en succès
- un noeud OU est en succès s'il existe au moins un fils en succès de ce noeud, et est en échec sinon.

  Nous pouvons donc bâtir une expression logique à partir des feuilles en échec de l'arbre (L') et, en faisant "remonter" les évaluations des feuilles de l'arbre aux noeuds intermédiaires, jusqu'à sa racine P, nous pourrons obtenir ainsi une démonstration structurée de $\neg P$.

On peut appliquer cette idée dans le cas où P est défini par un ensemble de propositions, avec l'hypothèse de monde fermé, pour s'assurer que P est entièrement défini par l'ensemble des propositions donné. En effet, on peut toujours définir un arbre ET-OU de résolution d'un prédicat P, comme suit :

soit l'ensemble des propositions définissant P :

$$P \vee \neg Q \vee \neg R \ , \ P \vee S \ , \ P \vee \neg T \vee \neg W \quad ;$$

on peut écrire P = (OU(ET Q R) S (ET T W)), et ainsi de suite pour S,Q,R,T,W ..., ce qui permet de définir un arbre ET-OU de racine P.

Dans les paragraphes suivants, nous établirons des règles de calcul pour former l'expression logique L' qui servira à expliquer $\neg P$, ensuite, nous les appliquerons sur un exemple.

II.3.2.2 - Calcul d'une expression logique permettant d'expliquer $\neg P$, sous les hypothèses décrites en II.3.2.1

Notation : dans toute la suite de ce chapitre II.3, nous utiliserons la notation : X-Y pour désigner une "expression complexe", où X est une expression complexe "détaillant" l'expression Y, (Y => X).

Nous allons définir une fonction E(N), qui évaluera ce qui doit être vrai (condition suffisante) pour que le noeud N soit un succès. Pour cela, nous utiliserons les connecteurs logiques classiques $\wedge$ et $\vee$, et les connecteurs $\wedge s$ et $\vee s$ (ET- et OU- séquentiels), comme suit :

1. si $N = \bigvee_{i=1}^{n} N_i$, $E(N) = \begin{cases} = \text{vrai si } \bigvee s_{i=1}^{n} E(N_i) = \text{vrai} \\ = \left[\bigvee s_{i=1}^{n} E(N_i) \right] - N \text{ sinon} \end{cases}$

E(N) est ici une liste de deux expressions, dont l'une est le nom du noeud lui-même (pour mémoire).

2. si $N = \bigwedge_{i=1}^{n} N_i$, $E(N) = \bigwedge s_{i=1}^{n} E(N_i)$ (une seule expression)

3. si N est une feuille de l'arbre,

E(N) = N si N est en échec

= VRAI sinon

(ce qui peut s'unifier en E(N) = N $\forall N$, en utilisant comme convention que VRAI est une feuille possible de l'arbre).

De plus,

4. $\left[\underset{i=1}{\overset{n}{Vs}}\ E(N_i)\right] - N = VRAI$ si $\exists i_o \leq n$ tq $E(Ni_o) = VRAI$

5. $\underset{i=1}{\overset{n}{\Lambda s}}\ E(N_i) = E(Ni_o)$ si $\exists i_o \leq n$ tq Ni_o est en échec

et $\forall_j < i_o$, N_j est en succès.

(= VRAI sinon)

L'intérêt des opérateurs Vs et Λs est de n'avoir à évaluer, à tout moment, que le minimum d'informations nécessaires.

exemple : soit $A = A_1 \Lambda A_2 \Lambda A_3$; si l'on sait que A_1 est vrai, et A_2 en échec, il est inutile d'évaluer la valeur de vérité de A_3 puisque l'on peut déjà conclure que A est en échec (i.e.: il est inutile d'explorer le sous-arbre dont la racine est A_3).

A chaque développement d'un noeud N (règles 1 ou 2), on commencera donc par vérifier si des feuilles de l'arbre n'ont pas été introduites, ce qui permettra éventuellement de conclure directement sur E(N), sans avoir à développer les noeuds restants (estimation en largeur, de façon locale) ; d'autre part, si l'on fait apparaître un noeud P dont on a déjà calculé l'expression de E(P), cela peut permettre éventuellement d'accélérer l'évaluation de E(N). Si une conclusion directe n'est pas possible, on commencera alors par évaluer E pour le noeud le plus à gauche de l'expression (et ainsi de suite).

Remarque :

Quand on applique la règle 1, on associe à l'expression évaluant E(N) le nom du noeud N ; ceci nous servira à donner les différentes versions de l'expression logique L' associée à l'arbre, correspondant à des niveaux de profondeur différents de cet arbre. Cela nous permettra de générer ensuite une réponse structurée par niveaux de détails.

Une fois définie la fonction E(N), étant donné un arbre ET-OU de racine P (dont on sait, par ailleurs, pour notre application, que P est en échec), il suffit de calculer E(P) pour obtenir une expression logique exprimant ce qui doit être vrai (au moins) pour que P soit vrai (P => E(P)).

Cette expression "composée" est de la forme :

$(V\ expr)_i - N$, où $(V\ expr)_i$ est l'expression logique qui doit être vraie pour que N soit vrai, et où expr est de la même forme. (liste de deux éléments)

Remarque :

- Les règles de définition de E assurent bien que E(N) est une expression L' telle que, L étant l'expression logique associée à l'arbre ET-OU de racine N, nous avons bien L => L'. En effet, si

  $N = \bigwedge_{i=1}^{n} N_i$, $\forall i_o \leq n$, $N \Rightarrow Ni_o$; or la règle 5 préserve bien ceci.

  Par construction on a donc L => L', et de plus L' fausse puisque tous les noeuds - ou feuilles - figurant dans L' - sont faux (et l'opérateur $\neg$ ne figure jamais dans L').

- Notons que l'expression L' est définie en fonction du sens de parcours de l'arbre de résolution. Ceci se retrouve dans la définition de l'opérateur $\wedge s$ (ici, on commence à parcourir l'arbre par la gauche).

Pour un sens de parcours de l'arbre donné, L' est la condition minimale telle que L => L' et L' fausse, ceci étant également assuré par la règle 5.

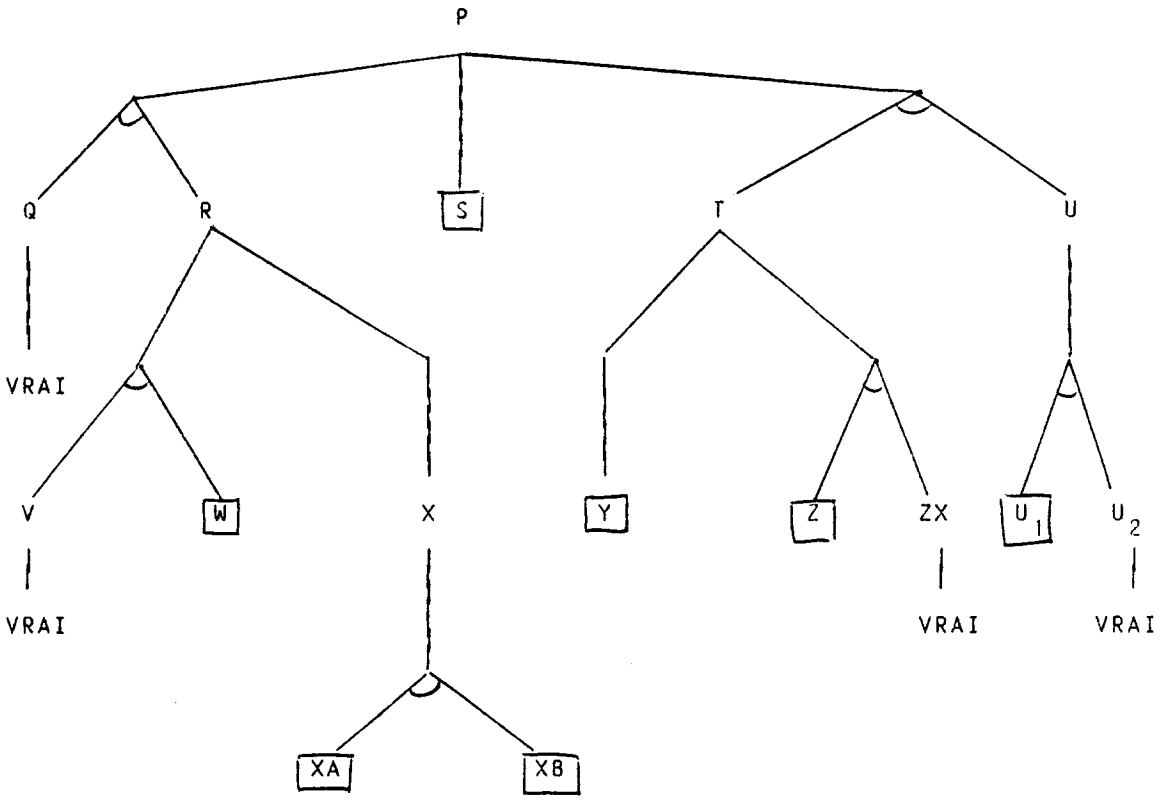
En relisant la négation de cette expression, en commençant toujours par l'expression logique formée à partir des termes de droite de chaque sous-expression (et ainsi de suite, récursivement), on obtient une démonstration structurée de $\neg P$.

II.3.2.3 - Application sur un exemple

Soit l'ensemble de propositions suivant :

{P V ¬Q V ¬R , R V ¬V V ¬W , T V ¬Y ,
P V ¬S , R V ¬X , T V ¬Z V ¬ZX ,
P V ¬T V ¬U , X V ¬XA V ¬XB , Q, V, ZX,
U V ¬U_1 V ¬U_2 , U_2 }

On voudrait savoir pourquoi P est faux - (en faisant l'hypothèse de monde fermé)
L'arbre ET-OU de résolution de P est le suivant :



(les feuilles encadrées sont en échec, et on a remplacé chaque proposition unitaire PU par la proposition PU V ¬VRAI, donc :
E(PU) = VRAI)

E(P) = [(E(Q) Λs E(R)) Vs E(S) Vs (E(T) Λs E(U))]- P
= [E(R) Vs S Vs (E(T) Λs E(U))]- P
= [{[E(V) Λ E(W)] Vs E(X)}- R Vs S Vs (E(T) Λs E(U))]- P

= [(W V E(X))- R Vs S Vs (E(T) As E(U))]- P

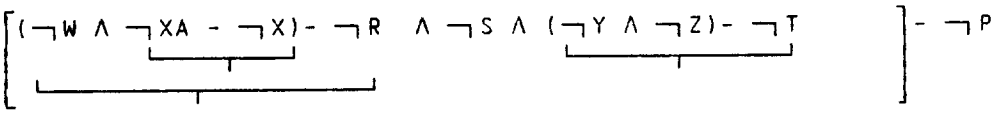
= [(W V (E(XA) As E(XB))- X)- R Vs S Vs (E(T) As E(V))]- P

= [(W V XA-X)-R V S V (E(T) As E(U))]- P

= [(W V XA-X)- R V S V
([E(Y) Vs (E(Z) As E(ZX))]- T As E(U))]- P

= [(W V XA-X)- R V S V (Y V Z)- T]- P

La négation de cette expression complexe est :



L'explication par niveaux recherchée est donc :

¬R ∧ ¬S ∧ ¬T => ¬P

¬W ∧ ¬X => ¬R

¬XA => ¬X

¬Y ∧ ¬Z => ¬T

Remarques :

1) comme prévu, toutes les branches de l'arbre n'ont pas été explorées (exemple : sous-arbre issu de U)

2) le remplacement de Vs (respectivement As) par V (respectivement par ∧) est justifié, une fois ses arguments évalués, par la définition même de Vs (respectivement de As).

II.3.2.4 - Conclusion

Nous avons montré qu'une formalisation de la résolution d'un but en calcul des propositions avec un arbre ET-OU était tout à fait adéquate avec nos objectifs initiaux ; nous avons pu donner un système de règles de réécriture permettant, à partir de l'arbre ET-OU, de générer des explications négatives structurées par niveaux de détails (c'est-à-dire, par niveaux de profondeur des noeuds de l'arbre).

Avec le formalisme des arbres ET-OU, le problème des

explications négatives est donc résolu en logique des propositions.

Dans la suite, nous essaierons de généraliser ce processus à la logique des prédicats. Dans ce cas, le problème est bien plus complexe car, intuitivement, les environnements au cours de la résolution ont une place prépondérante à jouer, et il faut toujours éviter de rentrer dans des problèmes d'explosion combinatoire (dans la mesure du possible).

II.3.3 - Généralisation du processus à la logique des prédicats restreinte aux clauses de Horn

Nous allons brièvement énoncer la succession d'opérations à réaliser pour généraliser la méthode proposée dans le paragraphe précédent. Puis nous détaillerons chacune de ces opérations, en donnant les algorithmes associés. Enfin, nous appliquerons la méthode sur deux exemples : l'un sera tout à fait formel ; l'autre, plus simple, sera celui que nous avons utilisé déjà plusieurs fois dans les paragraphes précédents ; cela nous permettra de comparer nos résultats avec , en particulier, ceux obtenus par le principe de f-résolution de Morgan.

II.3.3.1 - Le principe dans ses grandes étapes

La démarche reprend exactement celle que nous avons donnée en II.3.2, et nous travaillons également sous l'hypothèse de monde fermé. Mais cette fois, l'arbre ET-OU associé à la résolution du but $P(x_0,..)$ est plus complexe ; un noeud sera ici défini par trois composantes : un littéral, un environnement d'entrée (i.e., l'environnement obtenu par la résolution avant la tentative d'effacement du littéral), et un environnement de sortie (i.e., l'environnement obtenu par la résolution après la tentative d'effacement du littéral).

De façon globale, nous procédons en deux étapes. La première consiste à construire un arbre de résolution ET-OU d'un but donné $P(x_0,...)$ à partir de l'ensemble de clauses constituant la base de connaissances initiale ; pour pouvoir construire un arbre le plus proche possible de celui utilisé en logique des propositions, et le plus simple

possible, nous sommes amenés auparavant à appliquer une normalisation à l'ensemble initial de clauses. L'arbre que nous construisons présente les particularités suivantes : toutes ses feuilles correspondent à des littéraux dont le prédicat est commun, et les modifications "instantanées" des environnements ne se font qu'au niveau des feuilles.

L'arbre étant supposé en échec, la seconde grande étape est celle de la génération des explications négatives. Nous définissons d'abord ce qu'est une feuille en échec dans l'arbre. Puis, comme pour la logique des propositions, nous constituons une première expression à partir des feuilles en échec dans l'arbre, permettant d'expliquer l'échec sur $P(x_0,...)$. Il s'agit ensuite de pouvoir présenter cette expression selon les différents niveaux de l'arborescence de l'arbre - ceci correspondant en fait à une structuration par niveaux de détails.

Dans la suite, nous présenterons chacune de ces opérations de façon détaillée, en les justifiant au besoin.

II.3.3.2 - Normalisation des clauses formant la base de connaissances initiale BdC

II.3.3.2.1 - Motivations

Nous avons deux grandes exigences, qui justifient la normalisation que nous proposerons :

D'une part, comme en logique des propositions, nous désirons pouvoir rassembler toutes les clauses de BdC définissant un prédicat P sous un même littéral de tête. En d'autres termes, si P est défini par :

$$(1)\quad \begin{cases} Cl_1 : P(...) \vee \neg(\text{corps-clause-}Cl_1) \\ Cl_2 : P(...) \vee \neg(\text{corps-clause-}Cl_2) \\ Cl_3 : P(...) \vee \neg(\text{corps-clause-}Cl_3), \end{cases}$$

les littéraux de tête n'étant pas forcément identiques (au renommage des variables près), nous voulons transformer nos clauses Cl_i en Cl_i' de telle façon que nous puissions écrire

(2) $P(...) \vee \neg(\text{corps-clause-}Cl_1') \wedge \neg(\text{corps-clause-}Cl_2') \wedge \neg(\text{corps-clause-}Cl_3'))$

Une façon de pouvoir le faire de façon automatique est de transformer chaque clause Cl_i de manière à ce que, dans le littéral de tête de la clause transformée, ne figurent que des variables, toutes différentes ; alors, à un renommage près des variables, les littéraux de tête des clauses transformées définissant P seront identiques. Alors l'écriture (2) sera possible.

D'autre part, nous aimerions générer un arbre de résolution tel que les modifications directes de l'environnement associé à un littéral (par opposition à modification par répercussion dans les branches de l'arbre) ne puissent avoir lieu que au niveau des feuilles de l'arbre, et non, par exemple, directement lors des appels entre littéraux (vu sous l'aspect procédural). Une façon de le faire est de transformer aussi les corps de clauses de telle façon que, comme arguments d'un littéral ayant comme prédicat Q, Q(...) figurant déjà dans la clause non transformée, ne figurent maintenant que des variables, toutes différentes. En fait, c'est une transformation tout à fait similaire à celle appliquée sur les têtes de clauses.

Pour faire ces transformations nous introduirons le prédicat prédéfini E (pour EGAL), à deux arguments, dont le sens est : E(x,y) est vrai si et seulement si il y a identité entre x et y. Comme l'on fait de plus l'hypothèse que deux constantes de noms différents ont des valeurs différentes, ce prédicat définit même l'égalité.

Alors, dans les clauses transformées, puisque les littéraux dont le prédicat est différent de E ne portent que sur des variables toutes différentes, une clause ne pourra être en échec que parce qu'il y a échec sur un littéral E(...), ou sur un littéral ayant un prédicat prédéfini (cas particulier des prédicats arithmétiques). Nous supposons bien sûr que les prédicats non prédéfinis sont tous définis.

NB : Dans la suite de cette études, nous ne traiterons pas les cas particulier des prédicats prédéfinis autres que E (ex: prédicats

arithmétiques).

II.3.3.2.2 - Règles de normalisation des clauses

Nous utiliserons les conventions d'écriture suivantes pour écrire les règles :

clause = littéral-tête . corps-clause

littéral-tête = prédicat . liste-termes

corps-clause = $prem_1 . prem_2 \ldots prem_n$

(u ← v) dénote le remplacement de u par v.

R1 - si ∃ t ∈ liste-termes tq t est une constante ou un terme fonctionnel alors

. v:= nouvelle variable
. clause1:= (t ← v) (clause)
. corps-clause':= ¬(E v t) . corps-clause1
. clause':= littéral-tête1 . corps-clause'

exemple : P(x,A,z) V ¬Q(x,z)

R1
=> P(x,y,z) V ¬E(y,A) V ¬Q(x,y)

R2 - si $\exists(t_1,t_2) \in (\text{liste-termes})^2$ tq t_1 et t_2 sont des variables de même nom, alors

. v2:= nouvelle variable
. littéral-tête':= (t_2 ← v_2) . (littéral-tête)
. corps-clause':= ¬(E v_2 t_1) . corps-clause
. clause':= littéral-tête' . corps-clause'

exemple : P(x,x,z) V ¬Q(x,z)

R2
=> P(x,u,z) V ¬E(x,u) V ¬Q(x,z)

Remarque :

Les règles R1 et R2 ci-dessus répondent à la première des motivations exprimées en II.3.3.2.1. Les règles R3 et R4, définies

de façon symétrique par rapport à R1 et R2, vont répondre à la seconde motivation.

R3 -

si corps-clause = $prem_1$. $prem_2$. -- . $prem_n$
  si ∃ i ∈ [1...n] tq $prem_i$ = prédicat . liste-termes
    et tq ∃ t ∈ liste-termes, t étant une constante ou un terme fonctionnel,
  alors

. v:= nouvelle variable
. reste-corps-clause:= (t ← v)((prem$_i$. prem$_{i+1}$... . prem$_n$))
. corps-clause':= prem$_1$.. . prem$_{i-1}$. ¬(E v t). reste-corps-clause
. clause':= littéral-tête . corps-clause'

exemple : P(x,y,z) ∨ ¬Q(x,f(y,z))

R3
=> P(x,y,z) ∨ ¬E(u,f(y,z)) ∨ ¬Q(x,u)

R4 -

si corps-clause = $prem_1$. $prem_2$ . ... . $prem_n$
  si ∃i ∈ [1...n] tq $prem_i$ = prédicat.liste-termes
    et tq ∃(t_1,t_2) ∈ liste-termes^2, t_1 et t_2 étant de même nom,
    alors

. v_2:= nouvelle variable
. prem$_i$':= (t_2 ← v_2) (prem$_i$)
. corps-clause':= prem$_1$ . ... . prem$_{i-1}$. ¬(E v_2 t_2).
  prem$_i$'. prem$_{i+1}$...prem$_n$
. clause':= littéral-tête . corps-clause'

exemple : P(x,y,z) ∨ ¬Q(x,y) ∨ ¬R(y,y)

R4
=> P(x,y,z) ∨ ¬Q(x,y) ∨ ¬E (y,v) ∨ ¬R(y,v)

Ordre d'application des règles

Nous dirons qu'un littéral est normalisé s'il ne contient que des termes variables tous différents.

Soit une clause Cl à normaliser. Un ordre d'application des règles de normalisation peut être le suivant (en vue d'une normalisation automatique) :

- si littéral-tête (Cl) n'est pas normalisé, appliquer R1 ou R2, en suivant l'ordre d'apparition des termes dans le littéral = on obtient une clause Cl' ; normaliser Cl'.
- si corps-clause (Cl) n'est pas normalisé, appliquer R3 ou R4 en suivant l'ordre d'apparition des termes dans le corps de clause : on obtient une clause Cl' ; normaliser Cl'.

Exemple de normalisation d'une clause

Soit à normaliser la clause :

$P(x,A,f(x,z),x) \vee \neg g(x,x) \vee \neg h(f(x,z)) \vee \neg i(z,x)$

R2
$\Rightarrow P(x,A,f(x,z),x_1) \vee \neg E(x_1,x) \vee \neg g(x,x) \vee \neg h(f(x,z) \vee \neg i(z,x)$

R1
$\Rightarrow P(x,x_2,f(x,z),x_1) \vee \neg E(x_2,A) \vee \neg E(x_1,x) \vee \neg g(x,x) \vee \neg h(f(x,z))$
$\vee \neg i(z,x)$

R1
$\Rightarrow P(x,x_2,x_3,x_1) \vee \neg E(x_3,f(x,z)) \vee \neg E(x_2,A) \vee \neg E(x_1,x)$
$\vee \neg g(x,x) \vee \neg h(x_3) \vee \neg i(z,x)$

R4
$\Rightarrow P(x,x_2,x_3,x_1) \vee \neg E(x_3,f(x,z)) \vee \neg E(x_2,A) \vee \neg E(x_1,x)$
$\vee \neg E(x_4,x) \vee \neg g(x,x_4) \vee \neg h(x_3) \vee \neg i(z,x)$

laquelle clause est normalisée.

II.3.3.3 - Elaboration de l'arbre ET-OU de résolution de $P(x_0,\ldots)$

Dans toute la suite, nous supposons les clauses de notre programme normalisées de la façon indiquée en II.3.3.2.2. Nous supposons également que le seul prédicat prédéfini est le prédicat E, introduit en II.3.3.2.

Dans un premier temps, nous allons décrire de façon précise l'arbre sur lequel nous voulons travailler ; nous nous attacherons particulièrement à la caractérisation des noeuds et des feuilles en échec de cet arbre. Ensuite nous donnerons un algorithme permettant de générer l'arbre désiré à partir d'un ensemble de clauses et d'un but posé, en utilisant une stratégie de résolution complète (selon nos hypothèses de travail).

II.3.3.3.1 - **Caractéristiques de l'arbre souhaité**

II.3.3.3.1.1 - Structure générale de l'arbre

Si l'on fait abstraction des informations portant sur les environnements de résolution, la structure de notre arbre reproduit fidèlement l'arbre d'appel des clauses. Cela est rendu possible grâce à la forme normalisée des clauses (cf. première motivation pour la normalisation). On suppose que l'on fait le renommage adéquat des variables au cours de la construction de l'arbre.

Par rapport à l'arbre ET-OU utilisé en logique des propositions, les différences principales sont que :

- les noeuds, ou feuilles de l'arbre ne sont pas seulement des littéraux mais des triplets <littéral, environnement de résolution en entrée, environnement de résolution en sortie>, où le littéral est écrit tel qu'il figure dans la base de clauses (au renommage près).
- les feuilles de l'arbre portent toutes sur le prédicat E, ceci étant assuré par la normalisation des clauses (cf. seconde motivation).

Dans le paragraphe suivant, nous expliquerons comment gérer les environnements dans l'arbre.

II.3.3.3.1.2 - Gestion des environnements de résolution

II.3.3.3.1.2.1 - Généralités

Associé à une feuille de notre arbre, un environnement se

définit comme une expression logique formée de sous-expressions de la forme $(t_1 = t_2)$, où t_1 et t_2 sont des termes (constantes, variables, ou termes fonctionnels), et où l'expression $(t_1 = t_2)$ est soit obtenue lors de l'effacement d'un littéral $(E\ t_1\ t_2)$, soit figurait déjà dans l'environnement d'entrée de la racine de l'arbre ; dans le premier cas, il sera d'ailleurs intéressant de conserver l'association entre l'expression $t_1 = t_2$ et la feuille de l'arbre associée, pour le traitement des échecs.

Soit en effet à effacer le littéral $(E\ t_1\ t_2)$, avec l'environnement d'entrée env-e (consistant). Si l'expression env-e $\wedge$ $\{(t_1 = t_2)\}$ n'est pas inconsistante, l'environnement de sortie env-s associé à $(E\ t_1\ t_2)$ s'écrira : env-s = env-e $\wedge$ $(t_1 = t_2)$. Si env-e $\wedge$ $(t_1 = t_2)\}$ est inconsistant, $(E\ t_1\ t_2)$ n'est pas effaçable, et l'on pourra noter env-s = $\square$.

Notons que l'ordre des expressions élémentaires $(t_1 = t_2)$ dans un environnement dépend de la stratégie de résolution appliquée ; ici, nous avons choisi une stratégie à la PROLOG, l'arbre est parcouru de gauche à droite. Ceci est vrai pour les environnements des feuilles ou des noeuds de l'arbre.

D'après ce qui précède, un environnement d'entrée est donc :

- soit vide (i.e., aucune contrainte sur les termes n'a été introduite depuis le début de la résolution - ceci implique que le littéral but P(x,...), correspondant à la racine de l'arbre, ne comporte pas de variable instanciée) - ce qui sera noté : Ø - ,
- soit formé d'un ensemble non vide consistant d'expressions élémentaires.

D'autre part, un environnement de sortie est soit formé d'un ensemble non vide consistant d'expressions élémentaires, soit inconstant (noté $\square$).

Nous noterons env-e(N) (respectivement : env-s(N))

l'environnement d'entrée (respectivement : de sortie) associé à un noeud N. Par définition d'un arbre ET-OU, et selon le principe de résolution, la gestion des environnements associés aux noeuds de l'arbre est donnée par les règles suivantes :

- si $N = \bigvee_{i=1}^{n} N_i$

  $\forall_i = 1..n, \quad env\text{-}e(N) = env\text{-}e(N_i)$

  $env\text{-}s(N) = \bigvee_{i=1}^{n} env\text{-}s(N_i)$

- si $N = \bigwedge_{i=1}^{n} N_i$

  $env\text{-}e(N) = env\text{-}e(N_1)$

  $\forall_i = 2,...n$ si $env\text{-}s(N_{i-1}) \neq \square$, alors $env\text{-}s(N_{i-1}) = env\text{-}e(N_i)$

  si $\exists_{io}$ tq $env\text{-}s(N_{io}) = \square$ alors $env\text{-}s(N) = \square$, et on n'évaluera pas $env\text{-}e(N_j)$ pour $j \geq i_o$

  sinon $env\text{-}s(N) = env\text{-}s(N_n)$.

Remarques :

- pour tout noeud N tel que env-s(N) n'est pas inconsistant, on peut toujours écrire env-s(N) sous la forme env-e(N) $\wedge$ f, où f est une expression logique traduisant les contraintes sur les termes introduites par la résolution du littéral du noeud N
- un noeud de l'arbre est dit être en échec si la résolution du littéral correspondant a échoué. N est donc en échec si et seulement si env-s(N) = $\square$ (ou bien, si les environnements associés à N n'ont pas été évalués, selon notre stratégie de résolution choisie). De plus, si N est un noeud OU, tous ses fils sont en échec ; si c'est un noeud ET, il existe au moins un de ses fils en échec et ses ancêtres sont en échec,

II.3.3.3.1.2.2 - Application sur un exemple.

Pour alléger les écritures, nous convenons de supprimer les parenthèses entourant habituellement les arguments des littéraux.

Soit l'ensemble de clauses normalisées suivant :

Px ⟵ P1x ∧ P2xy

Py ⟵ P3y ∧ P4y

P1x ⟵ Q1x ∧ Q2xz

P2xy ⟵ Q3x ∧ Q3y ∧ Q4xy

P3x ⟵ P31xz ∧ Q3z

P1x ⟵ ExF

Q1x ⟵ ExA

Q1x ⟵ ExB

Q2xz ⟵ ExC ∧ EzD

Q3x ⟵ ExA

Q3x ⟵ ExE

Q4xz ⟵ ExA ∧ EzD

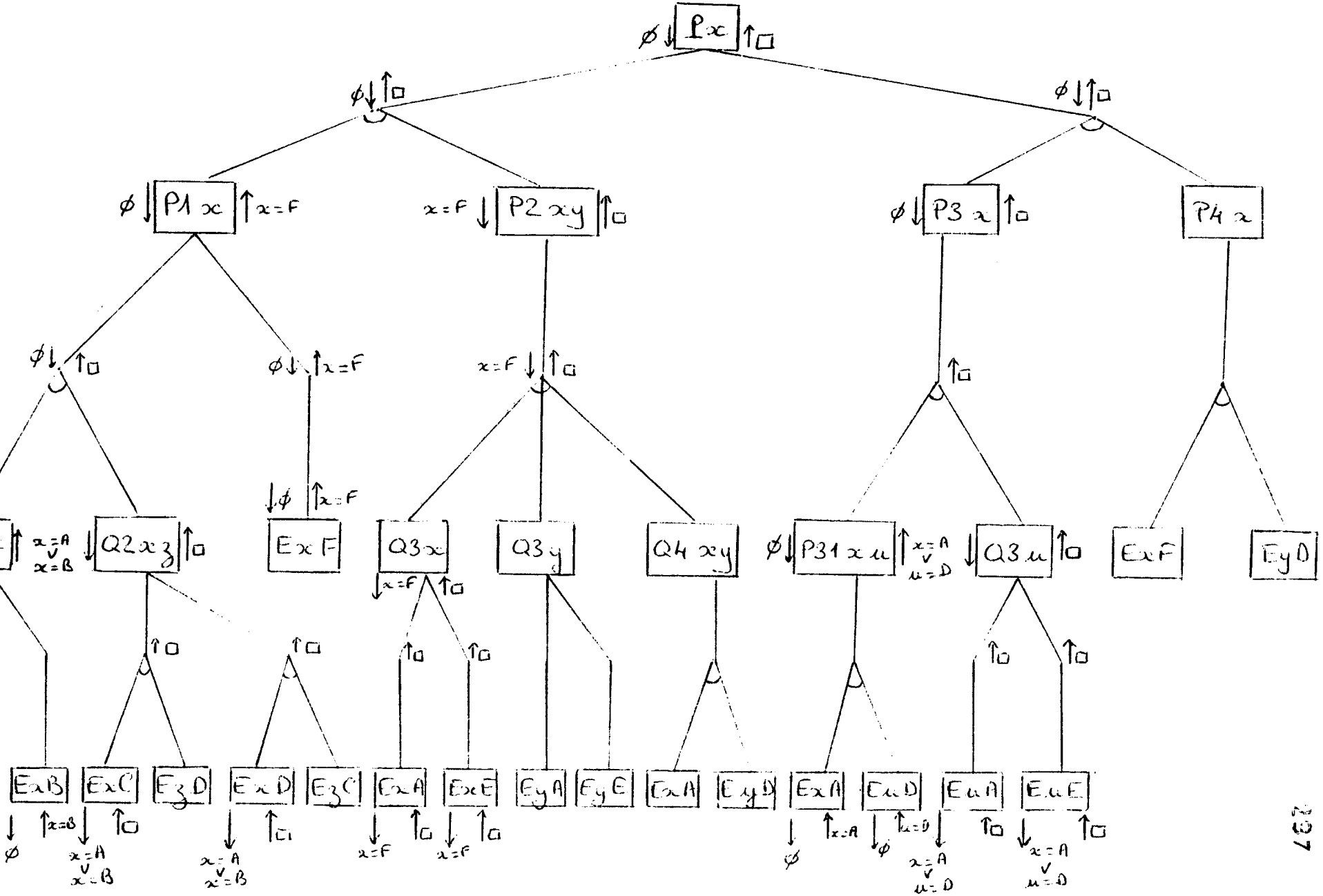
P4x ⟵ ExE

P31x ⟵ ExA ∧ EyD

P2xy ⟵ ExF ∧ EyA

et le but (supposé en échec) Px

Voici la forme de l'arbre de résolution de Px que nous voulons obtenir.
(notations : les environnements d'entrée sont précédés de la flèche ↓, ceux de sortie sont précédés de la flèche ↑).


Px
P1 x
P2 xy
P3 x
P4 x
Q2 xz
ExF
Q3 x
Q3 y
Q4 xy
P31 xu
Q3 u
ExF
EyD
ExB
ExC
EzD
ExD
EzC
ExA
ExE
EyA
EyE
ExA
EyD
ExA
EuD
EuA
EuE

II.3.3.3.2 - Constitution de l'arbre de résolution ET-OU de racine P(x,...)

II.3.3.3.2.1 - Généralités

Dans ce paragraphe, nous donnerons un algorithme d'élaboration de l'arbre que nous avons caractérisé en II.3.3.3.1.

Nous avons choisi de faire la gestion des environnements au fur et à mesure de la construction du squelette de l'arbre. Cela nous permettra d'avoir un arbre fini dans le cas où l'arbre des résolvantes généré par PROLOG serait fini.

En effet, dans le cas de récursivité, si les clauses sont "bien ordonnées" dans la base de clauses, l'évaluation des environnements dans l'arbre au fur et à mesure de sa construction permet de limiter cette construction, en utilisant le fait qu'un noeud OU est en succès

(i.e., son environnement de sortie est différent de $\square$) si il existe un de ses fils qui soit un noeud succès.

La stratégie de résolution utilisée est donc celle de PROLOG.

Les clauses étant normalisées, nous avons également choisi de renommer les variables des clauses au cours de leur intégration dans l'arbre, de façon à ne pas alourdir inutilement les environnements de résolution avec des expressions de la forme (variable 1 = variable 2).

L'algorithme que nous présentons ci-dessous est récursif. Par commodité et pour en faciliter la lecture, nous l'avons écrit dans un dialecte proche du langage LISP.

II.3.3.3.2.2 -Algorithme de construction de l'arbre

Par souci de clarté le prédicat E est noté ici EGAL.

La fonction Exp traduit l'algorithme d'expansion du sous-arbre

associé à un noeud (OU, ET, ou feuille : EGAL (t_1, t_2)). Ses arguments d'entrée sont donc, dans l'ordre :

- les littéraux associés au noeud
- l'environnement d'entrée associé au noeud

Cette fonction rend comme résultat la structure nd suivante :

- environnement d'entrée associé au noeud
- sous-arbre ayant le noeud comme racine
- environnement de sortie associé au noeud

Cet algorithme ne traite que des noeuds binaires (à deux branches), mais il est facilement généralisable aux noeuds n-aires.

NB : dans les algorithmes que nous présenterons, nous utiliserons les notations suivantes :

- let* (u = u1 , v = v1 ...) in Expr: traduit la notion d'affectations multiples à des variables u,v, etc... apparaissant dans Expr (notation héritée de LISP)
- f(...) <= Expr : Expr est le résultat de f(...).

```
Exp(lit,env) <= let* x V y = substitution (renommage(équivalence(lit)),lit)
                     r1 = nd (env,a1,env-s1) = Exp (x,env)
                     r2 = nd (env,a2,env-s2) = Exp (y,env)
                in  nd (env,OR(r1,r2), env-s1 V env-s2)

Exp(x ∧ y,env) <= let r1 = nd(env,a1,env-s1) = Exp (x,env)
                  in  si env-s1 ≠ □ alors
                          let  r2 = nd(env-s1,a2,env-s2) = Exp(y,env-s1)
                          in   nd(env,ET (r1,r2), env-s2)
                      sinon  nd(env,r1,□)

Exp(Egal(x,y),env) <= nd(env,Egal(x,y), unifsimpl(env ∧ (x=y)))
```

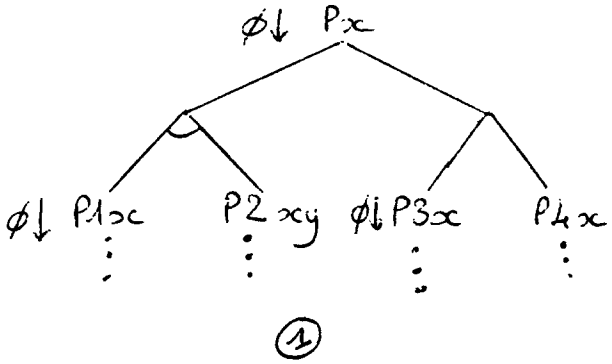
où

équivalence (lit)	désigne les corps des clauses définissant le prédicat de lit (lit : pour littéral)
renommage (c)	désigne le renommage des corps de clauses c avec de nouvelles variables
substitution (c,lit)	désigne l'opération de substitution dans les corps de clauses c (définissant le prédicat de lit) des variables apparaissant dans les têtes de clauses par celles du littéral d'appel lit.
inifsimpl(env Λ (x=y))	désigne l'opération d'unification, telle que :

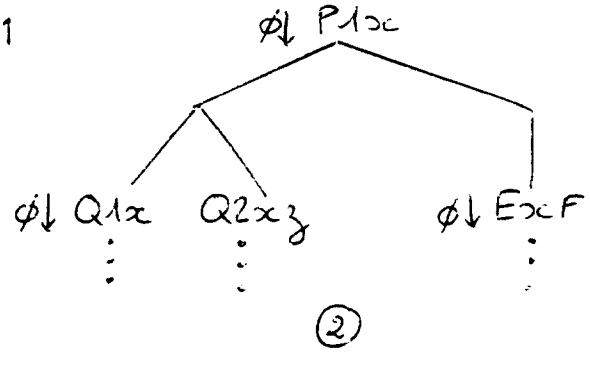
- si x est unifiable à y, selon un environnement initial env, unifsimpl renvoie l'environnement après l'unification ; si celle-ci était déductible de l'environnement initial, celui-ci reste inchangé -sinon, on effectue une opération de mise à jour en rajoutant le minimum d'équations nécessaires à l'environnement initial : c'est l'unification simplifiée classique.
- si l'unification est impossible, unifsimpl renvoie □ comme environnement de sortie. D'autre part, elle doit renvoyer également f un ensemble $\bar{E}$ d'équations minimal consistant, inclus dans l'environnement initial, tel que la conjonction logique entre les éléments de $\bar{E}$ et (x=y) soit inconsistante ; la détermination de $\bar{E}$, qui n'est pas mémorisée dans l'algorithme classique d'unification, fait l'objet du paragraphe suivant.

II.3.3.3.2.3 - Exemple d'application de l'algorithme

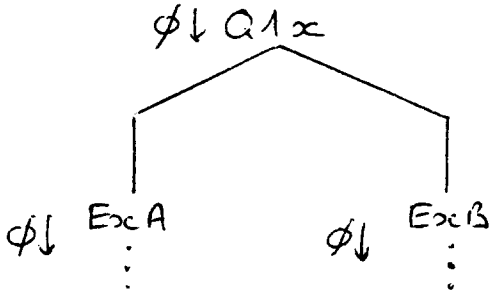
Nous allons appliquer notre algorithme de construction de l'arbre ET-OU relatif à un programme donné, sur un sous-ensemble des clauses de l'exemple précédent ; nous ne générerons donc qu'un sous-arbre du précédent (partie gauche). Nous noterons seulement les sous-arbres générés dans l'ordre dans lequel ils sont construits.



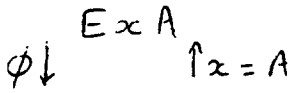
①



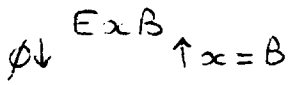
②



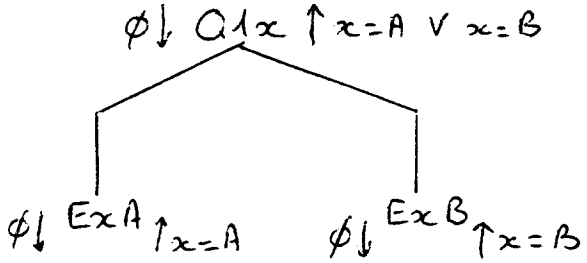
③



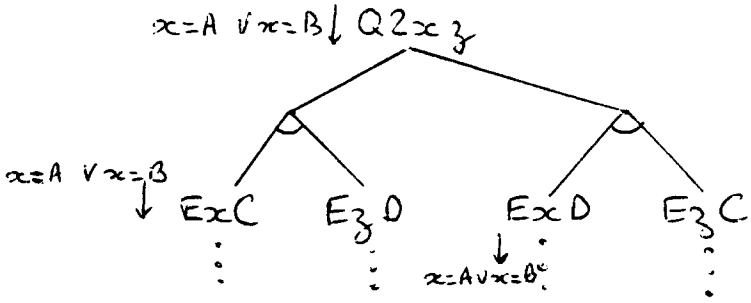
④



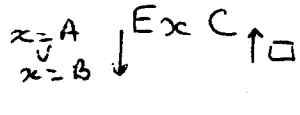
⑤



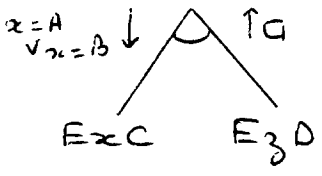
⑥



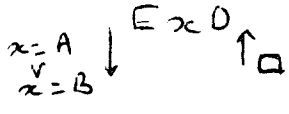
⑦



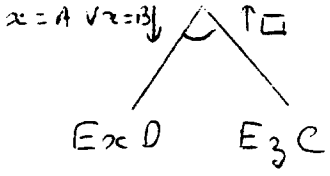
⑧



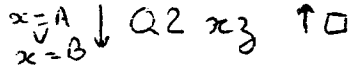
⑨



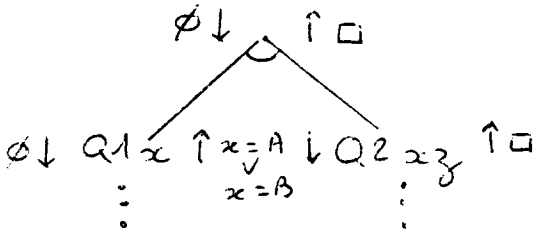
⑩



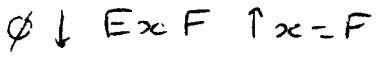
⑪



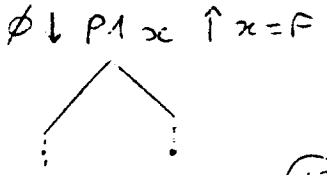
(12)



(13)

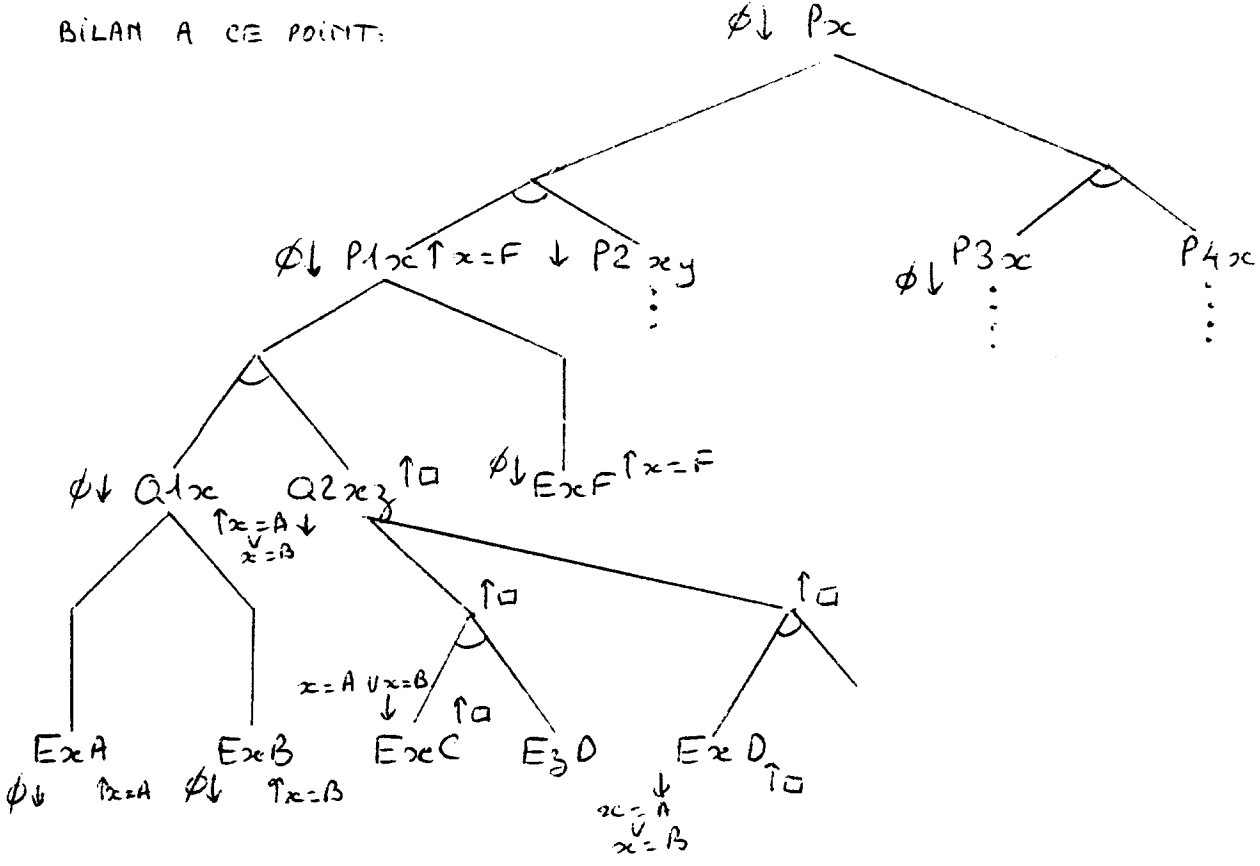


(14)



(15)

BILAN A CE POINT:



ETC

II.3.3.4 - Explication de l'échec sur $P(x_0,...)$, générée à partir de la connaissance des feuilles en échec de l'arbre

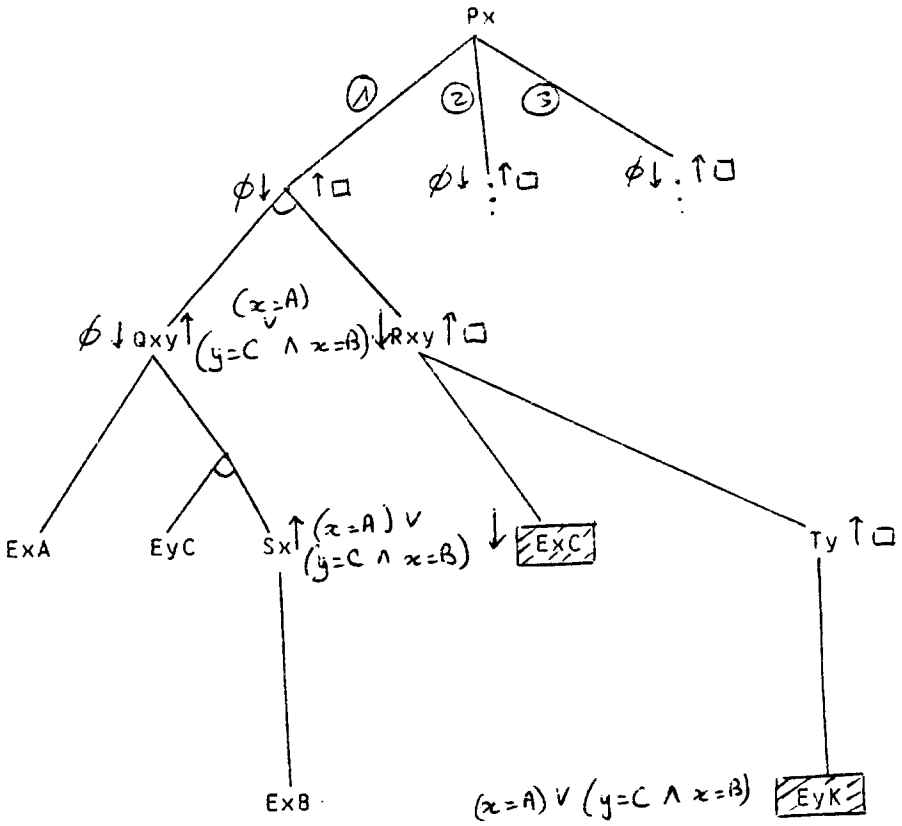
II.3.3.4.1 - Introduction : l'idée intuitive

Nous présentons ici l'idée intuitive qui motivera les paragraphes suivants.

Un arbre ET-OU étant en échec, il existe un sous-ensemble de feuilles en échec nécessaire et suffisant pour déterminer l'échec sur l'arbre entier. Or à une feuille en échec correspond un sous-arbre inconsistant, que l'on peut bâtir à partir d'une sous-expression de l'environnement d'entrée de la feuille, provoquant l'échec (environnement de sortie inconsistant) ; ce sous-arbre va fournir un élément de réponse pour l'explication négative : en effet, il suffit de considérer n'importe quelle tranche du sous-arbre pour avoir cet élément de réponse selon différents niveaux de détails. Une réponse complète s'obtiendra donc à partir des tranches de tous les sous-arbres inconsistants correspondant aux feuilles en échec du sous-ensemble décrit plus haut.

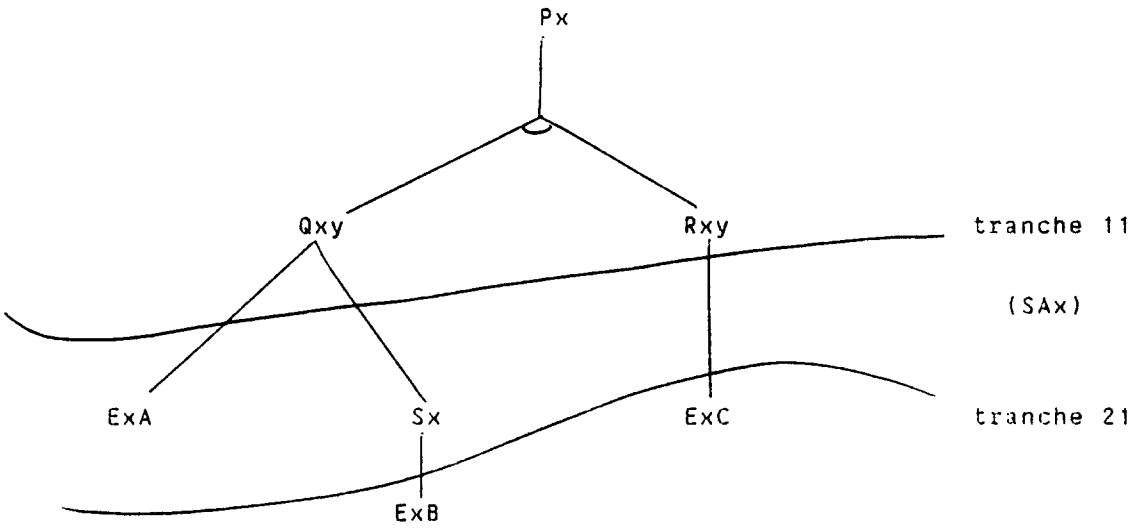
Exemple

Soit l'arbre de résolution de P(x) suivant :



Sur ce schéma, les feuilles hachurées appartiennent au sous-ensemble des feuilles en échec responsables de l'échec sur l'arbre (branche ① issue de la racine).

La feuille ExC est en échec car la sous-expression (x=A V x=B) Λ x=C est inconsistante. Le sous-arbre inconsistant SAx correspondant est :



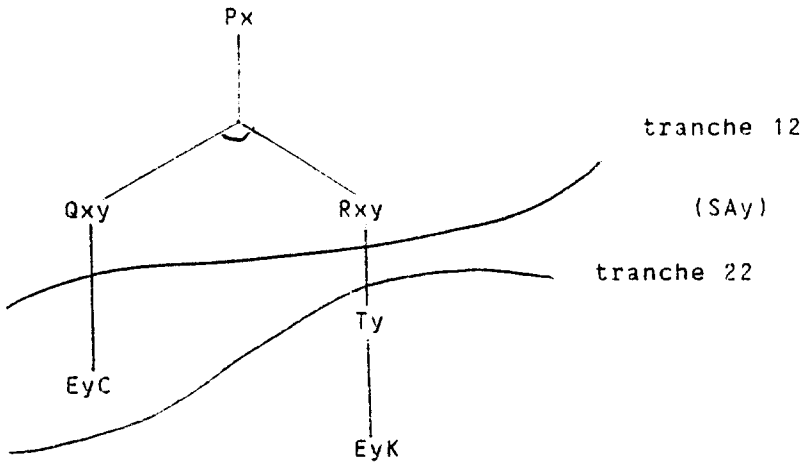
Sur SAx, nous avons représenté deux tranches possibles de l'arbre (entre d'autres) ; l'expression logique formée à partir des noeuds situés juste au-dessus de n'importe quelle tranche constitue un élément de la réponse qui permettra de générer l'explication négative (pourquoi Px est en échec).

tranche 11 Qxy Λ Rxy

tranche 21 (ExA V Sx) Λ Rxy

(en réalité, il faudra nier ces expressions pour obtenir la formule finale).

La feuille EyK est en échec car la sous-expression y=CΛy=k est inconsistante. Le sous-arbre inconsistant SAy correspondant est :



Les sous-expressions associées aux tranches représentées sur SAy sont :

tranche 12 Qxy ∧ Rxy

tranche 22 EyC ∧ Rxy

A partir de ces expressions, on peut construire différents éléments qui permettront de constituer l'expression finale. Ainsi :

* tranche 11, tranche 12 :

  $\forall x,y \longrightarrow \neg (Qxy \wedge Rxy)$

* tranche 11, tranche 22

  $\forall x,y \longrightarrow \neg (Qxy \wedge Rxy)$

  (ici, l'expression issue de la tranche 22 "détaille" celle de la tranche 11)

* tranche 21, tranche 12

  $\forall x,y \longrightarrow \neg (Qxy \wedge Rxy)$

  (même remarque que précédemment)

* tranche 21, tranche 22

  $\forall x,y \longrightarrow \neg [((ExA \vee Sx) \wedge Rxy) \vee (EyC \wedge Rxy)]$

Nous avons ainsi obtenu un élément de réponse, donné selon différents niveaux de détails, correspondant à la branche 1 en échec dans l'arbre de racine Px.

Dans les paragraphes suivants, nous essaierons de formaliser cette idée, en donnant une façon de procéder pas à pas pour former une expression complexe (au sens défini en II.3.2.2, dans le cas de la logique des propositions) qui contiendra toute l'information nous permettant de donner des explications par niveaux de détails.

Comme dans le cas de la logique des propositions, cette explication sera générée à partir de l'expression logique bâtie sur les feuilles en échec de l'arbre de résolution (en échec) ET-OU de $P(x_0,...)$.

Dans un premier temps, le problème est donc de pouvoir former, pour une feuille en échec, l'expression logique caractérisant cet échec ; en effet, en logique des propositions, il s'agissait simplement du nom du prédicat en échec ; ici, il faut également prendre en compte une sous-expression de l'environnement d'entrée de la feuille qui, avec le littéral associé à la feuille, provoque l'inconsistance.

Dans une seconde étape, il faudra écrire l'expression S' d'échec (en reprenant les notations utilisées en logique des propositions) de l'arbre, et pouvoir en donner une version selon les différents niveaux de profondeur de l'arbre (expression structurée par niveaux) ; c'est ce que nous appelons plus vulgairement "faire remonter l'expression S' dans l'arbre ET-OU".

II.3.3.4.2 - Constitution de l'expression logique associée à une feuille en échec de l'arbre

Nous allons préciser l'idée générale qui permet d'exprimer cette formule associée à un échec. Puis nous donnerons un algorithme permettant de la générer - ce même algorithme permettant d'ailleurs, plus généralement, de déterminer un environnement de sortie associé à un littéral et son environnement d'entrée (lors de la constitution de l'arbre ET-OU de résolution).

II.3.3.4.2.1 - Principe général

Soit une feuille N en échec dans l'arbre. On suppose que son environnement d'entrée env-e n'est pas inconsistant et qu'il a été évalué et marqué sur le noeud, au cours de la construction de l'arbre ; son environnement de sortie env-s sera donc marqué $\square$.

Par construction (et d'après le principe de résolution), si litt(N) est le littéral associé à la feuille N (litt(N) étant de la forme (E t_1 t_2) où t_1 et t_2 sont des termes), nous aurons :
env-e $\wedge$ litt(N) $\vdash \square$.

Comme env-e était consistant, il existe une sous-formule $\bar{E}(N)$ de env-e, de longueur minimale (i.e., constitué d'un nombre minimal d'équations de type (ta = tb)), telle que

$$\begin{cases} E(N) \text{ consistant} \\ E(N) \wedge litt(N) \vdash \square \end{cases}$$

Si l'on est capable de trouver un tel $\bar{E}(N)$, l'expression logique (fausse) exprimant l'échec sur N sera donc $\bar{E}(N) \wedge litt(N)$.
exemple :

Soit N la feuille en échec déterminé par :

- le littéral E(x,C)
- l'environnement d'entrée $(x=A \vee x=B) \wedge y=D$
- l'environnement de sortie $\square$

Dans ce cas, si l'on prend $\bar{E}(N) = x=A \vee x=B$, $\bar{E}(N)$ vérifie les critères énoncés ci-dessus, car

$x=A \vee x=B$ est consistant

mais $f = (x=A \vee x=B) \wedge (x=C)$ ne l'est plus

f est la formule, exprimée au niveau le plus bas dans l'arbre, qui

caractérise l'échec sur la feuille N.

Nous donnerons un algorithme permettant de déterminer une sous-formule à partir de laquelle on puisse écrire directement l'expression logique f caractérisant l'échec en N. Cet algorithme doit être intégré dans l'algorithme d'unification simplifiée classique (cas d'échec).

Ensuite, il s'agit de pouvoir exprimer f relativement aux différents niveaux de profondeur de l'arbre. Pour cela, on suppose que lors de la mise-à-jour d'un environnement, au niveau d'une feuille de l'arbre, on associe à la nouvelle équation $(t_1=t_2)$ obtenu par l'effacement de la feuille $E(t_1,t_2)$, la liste des littéraux caractérisant tous ses noeuds ancêtres dans l'arbre, du plus jeune ancêtre (i.e.$(E\ t_1\ t_2)$ au plus vieux $(P(x,\ldots)$, racine de l'arbre). On pourra alors "remonter" l'expression logique f selon différents niveaux de détail (i.e., de profondeur dans l'arbre).

Cas particulier

Dans le cas où l'algorithme d'unification simplifiée est en succès sur une feuille (env-e, Exy, env-s), mais où la modification résultante de env-e ne consiste pas directement à lui rajouter l'équation (x=y), il faudra attacher la liste des ancêtres de Exy à la nouvelle équation eq telle que env-s = env-e $\wedge$ eq.

Exemple :

env-e = $(x=f(u)) \wedge (y = f(z)) \wedge (u=B)$
Soit à effacer E(x,y) (noté Exy)

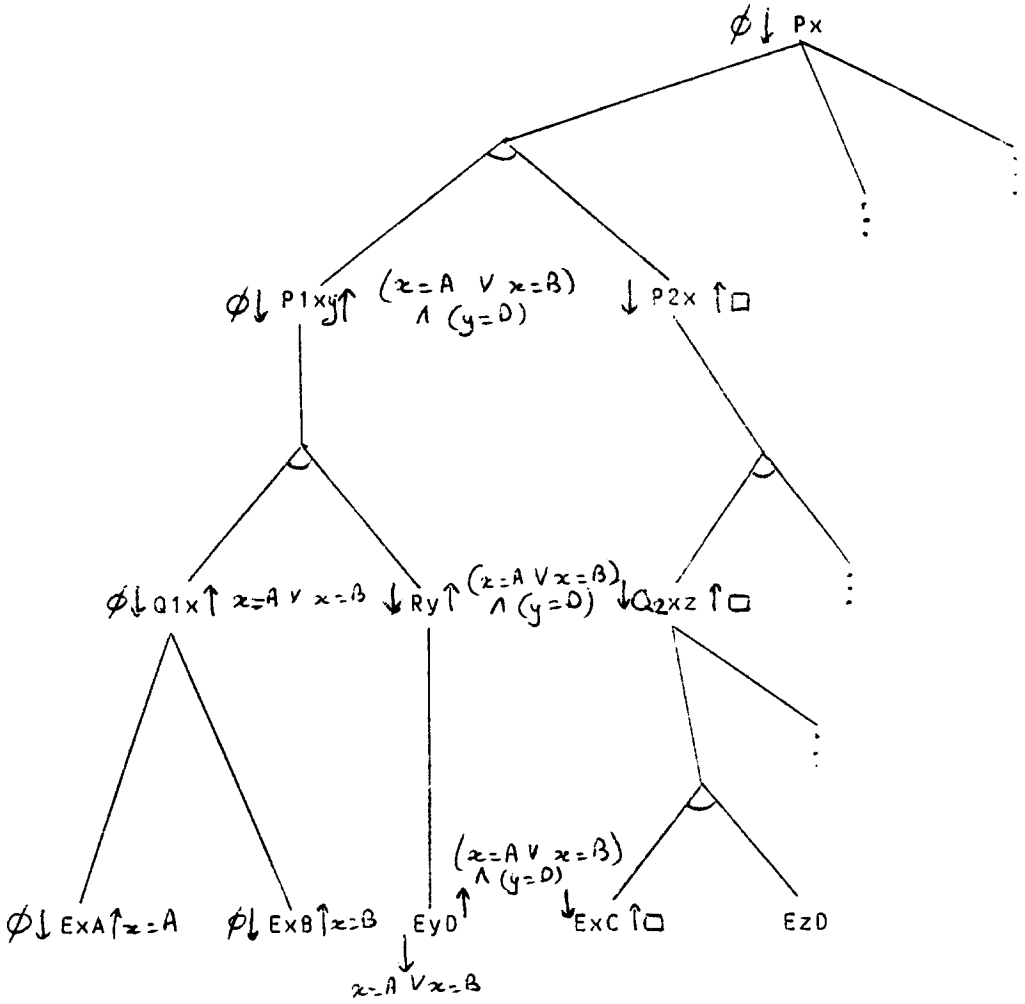
L'algorithme d'unification simplifiée classique consiste à transformer env-e en
env-s = $(x=f(u)) \wedge (y = f(z)) \wedge (u=B) \wedge (\underline{z=B})$
il faut donc attacher la liste des ancêtres de E(x,y) à l'équation (z=B).

Nous indiquerons d'abord comment modifier légèrement l'algorithme de construction d'arbre de résolution ET-OU pour pouvoir

garder trace des listes ancêtres. Puis nous donnerons un algorithme permettant de "remonter" dans l'arbre les formules exprimant des échecs sur des feuilles.

exemple d'expressions logiques associées à des échecs :

Supposons la portion d'arbre suivante :



Soit la feuille N en échec de littéral ExC, d'environnement d'entrée (x=A V x=B) Λ y=D.

Au niveau le plus bas de l'arbre, si f caractérise l'échec :

f = (x=A V x=B) Λ (x=C) ;

Les littéraux ancêtres de x=A sont ExA , Q1x , P1xy, Px.

Les littéraux ancêtres de x=B sont ExB , Q1x, P1xy , Px.

Les littéraux ancêtres de x=C sont ExC , Q2xz , P2x , Px.

Les expressions "remontées" de f sont :

(ExA ∨ ExB)∧ ExC (niveau feuilles)

Q1x ∧ ExC , Q1x ∧ Q2xz

P1xy ∧ ExC , P1xy ∧ Q2xz , P1xy ∧ P2x

... Px

II.3.3.4.2.2 - Association aux noeuds de l'arbre d'une liste ordonnée de leurs ancêtres

Nous avons vu que théoriquement, un noeud de l'arbre était caractérisé (ou identifié) par les trois informations suivantes :

- un environnement d'entrée
- un littéral
- un environnement de sortie.

(en pratique, nous le désignons souvent par le littéral associé).

Pour pouvoir associer de façon simple à un noeud la liste de ses noeuds ancêtres dans l'arbre, il suffit de modifier l'algorithme de construction de l'arbre de résolution. Les informations associées à un noeud (OU) comprendront donc en plus :

- un numéro (unique) permettant de l'identifier plus rapidement en pratique ; par exemple, les noeuds peuvent être numérotés selon leur ordre de création dans l'arbre.
- la liste des noeuds ancêtres, chaque élément de la liste pouvant être composé du couple (numéro de noeud, littéral associé au noeud).

Dans toute la suite, nous simplifierons les présentations en ne faisant pas apparaître explicitement ces nouvelles informations associées aux noeuds, si cela ne facilite pas la compréhension. On représentera souvent un noeud par son littéral associé (si cela ne donne pas lieu à des ambiguités).

II.3.3.4.2.3 - Algorithme permettant de déterminer une sous-formule minimale consistante "responsable" de l'échec sur une feuille

N étant une feuille d'environnement d'entrée env-e, et de littéral Ett', nous allons évaluer $\bar{E}(N)$, noté de façon plus détaillée $\bar{E}(t,t',env\text{-}e)$.

Par construction, env-e est une expression logique, dans laquelle l'ordre d'apparition des équations (ti = tj) est significatif. La démarche globale est la suivante :

1. Ecrire env-e sous forme normale disjonctive

On sait que ceci est toujours possible, en utilisant la distributivité de $\wedge$ par rapport à $\vee$. On obtient alors $env\text{-}e = \bigvee_{i=1}^{n} env\text{-}e_i$, où $env\text{-}e_i = \bigwedge_{j=1}^{p} eq_{ji}$, où les expressions élémentaires eq_{ji} sont de la forme (tx=ty).

2 - Pour chaque $env\text{-}e_i$, calculer $\bar{E}(t,t',env\text{-}e_i)$ (noté $\overline{Ei}$) avec l'algorithme présenté ci-dessous.

3 - Former $\bar{E}1(t,t',env\text{-}e) = \bigvee_{i=1}^{n} \bar{E}(t,t',env\text{-}e_i)$

Justification :

$\forall i=1...n$, $\bar{E}i$ est minimal consistant tel que $\bar{E}i \wedge litt(N) \vdash \square$

donc $\bigvee_{i=1}^{n} \bar{E}i$ est consistant

et $(\bigvee_{i=1}^{n} \bar{E}i) \wedge litt(N) \vdash \square$ (distributivité)

4 - $\bar{E}1(t,t',env\text{-}e)$ étant une disjonction de conjonctions d'expressions élémentaires (ti=tj), factoriser cette expression pour obtenir une forme normale conjonctive.

Ceci est possible grâce à la distributivité de $\vee$ par rapport à $\wedge$.

Exemple :

env-e = ((x=A) V (x=B)) Λ (y=x) Λ ((z=E) V (z=F))
litt(N) = E(y,C)

étape 1 :

env-e = [(x=A) Λ (y=x) Λ (z=E)] ; (env-e_1)
V [(x=A) Λ (y=x) Λ (z=F)] ; (env-e_2)
V [(x=B) Λ (y=x) Λ (z=E)] ; (env-e_3)
V [(x=B) Λ (y=x) Λ (z=F)] ; (env-e_4)

étape 2 :

$\bar{E}$(y,C,env-e_1) = {(x=A), (y=x)}

$\bar{E}$(y,C,env-e_2) = {(x=A), (y=x)}

$\bar{E}$(y,C,env-e_3) = {(x=B), (y=x)}

$\bar{E}$(y,C,env-e_4) = {(x=B), (y=x)}

étape 3 :

$\bar{E}1$(y,C,env-e) = [(x=A) Λ (y=x)] V [(x=B) Λ (y=x)]
étape 4 :
$\bar{E}$(y,C,env-e) = [(x=A) V (x=B)] Λ (y=x)

La suite de ce paragraphe concerne donc le cas où env-e s'écrit comme une conjonction de sous-termes de la forme (ti=tj) ; nous l'écrirons donc comme un ensemble ordonné d'équations.

<u>Algorithme de calcul de $\bar{E}$ (t,t',env)</u>

rappel des hypothèses :
- env est un ensemble d'équations de la forme variable = terme. (cf. normalisation des clauses)
- deux constantes de noms différents ont des valeurs différentes
- dans le cas où env Λ (t=t') est inconsistant, $\bar{E}$ (t,t',env) rendra le couple (inconsistant, ensemble $\bar{E}$ d'équations minimal consistant tel que la forme conjonctive associée à $\bar{E}$ U (t=t') soit inconsistante.

Un algorithme de calcul de $\bar{E}$ (t,t',env) est donc le suivant :

si env ≠ ∅ alors si t'=cste, alors A
si t'=terme fonctionnel, alors B
si t'=variable, alors C
sinon unification avec succès % env-s:= {(t=t')} %

A - Si t' est une constante
si ∃ eq ∈ env tq eq = (t=k), où k n'est pas une variable et k ≠ t'
alors (inconstant, {(t=k)})
sinon si ∃ eq ∈ env t.q. eq = (t=v) où v est une variable
alors si $\bar{E}$(v,t',env\{eq}) = (inconsistant, ensemble)
alors $\bar{E}$ (t,t',env):=(inconsistant, ensemble U {eq})
sinon $\bar{E}$ (t,t',env\{eq})
% sinon unification avec succès %

B - Si t' est un terme fonctionnel $f(t'_1,\ldots,t'_n)$

si ∃ eq ∈ env t.q. eq = (t=g(...)), où g est un symbole de fonction diférent de f
alors (inconsistant, {eq})
sinon si ∃ eq ∈ env t.q. eq = (t=k), où k est une constante
alors (inconsistant, {eq})
sinon si ∃ eq ∈ env tq eq = $(t=f(t_1,\ldots,t_n))$
alors si $\underset{i=1}{\overset{n}{\text{Suite}}}$ $\bar{E}$ $(t_i,t_i',env\backslash\{eq\})$=(inconsistant,ens)
alors (inconsistant, ens U {eq})
% sinon unification en succès %
sinon si ∃ eq ∈ env tq eq=(t=v) où v est une variable
alors si $\bar{E}$ (v,t',env\{eq}) = (inconsistant,ens)
alors (inconsistant, ens U {eq})
% sinon, unification en succès %
% sinon unification avec succès %

où $\underset{i=1}{\overset{n}{\text{Suite}}}$ $\bar{E}$ (t_i,t_i',env) est défini comme suit :

si $i \leq n$

si $\bar{E}(t_1, t_1', env) = (\text{inconsistant, ensemble})$

alors $\underset{i=1}{\overset{n}{\text{Suite}}}\ \bar{E}(t_i, t_i', env) = (\text{inconsistant, ensemble})$

sinon si $\underset{i=2}{\overset{n}{\text{Suite}}}\ \bar{E}(t_i, t_i', env) = (\text{inconsistant, ensemble})$

alors $\underset{i=1}{\overset{n}{\text{Suite}}}\ \bar{E}(t_i, t_i', env) = (\text{inconsistant, ensemble})$

% sinon unification avec succès %

C - Si t' est une variable

si $\exists$ eq $\in$ env tq eq = (t'=k) où k est une constante

alors si $\bar{E}(t, k, env \setminus \{eq\}) = (\text{inconsistant, ensemble})$

alors (inconsistant, ensemble $\cup$ {(t'=k)})

% sinon unification avec succès %

sinon si $\exists$ eq $\in$ env tq eq = $(t' = f(t'_1, \ldots, t'_n))$

alors si $\bar{E}(t, f(t'_1, \ldots, t'_n), env \setminus \{eq\}) = (\text{inconsistant, ensemble})$

alors (inconsistant, ensemble $\cup$ {eq})

% sinon unification avec succès %

sinon si $\exists$ eq $\in$ env tq eq = (t'=v) où v est une variable

alors si $\bar{E}(t, v, env \setminus \{eq\}) = (\text{inconsistant, ensemble})$

alors (inconsistant, ensemble $\cup$ {(t=t')})

sinon $\bar{E}(t, t', env \setminus \{eq\})$

% sinon unification avec succès %

NB : Nous ne garantissons pas que les ensembles générés par cet algorithme soient strictement minimaux. Cependant ils ne contiennent aucun élément "inutile" (i.e. ne participant pas à la raison de l'échec).

Exemples d'application de l'algorithme ci-dessus

Exemple 1

env = {(x=A),(u=c),(x=y)(z=B)}

litt (N) = (E y f(z,u))

Ē (y,f(z,u), env)

- eq = (x=y)
- Ē (x,f(z,u), {(x=A),(u=C),(z=B})
  - eq = (x=A)
  - → {x=A)}

→ {(x=A),(x=y)} = solution (inconsistant)

Exemple 2

env = {(y=x),(u=v),(x=A),(x=u),(v=w)}

litt(N) = (E u 5)

Ē (u,5,env)

- eq = (u=v)
- Ē (v,5,{(y=x),(x=A),(x=u),(v=w)}) (consistant)
  - eq = (v=w)
  - Ē (w,5, {(y=x) (x=A) (x=u)}) (consistant)
  - Ē (v,5,{(y=x),(x=A),(x=u)} (consistant)

Ē (u,5,{(y=x),(x=A),(x=u),(v=w)}

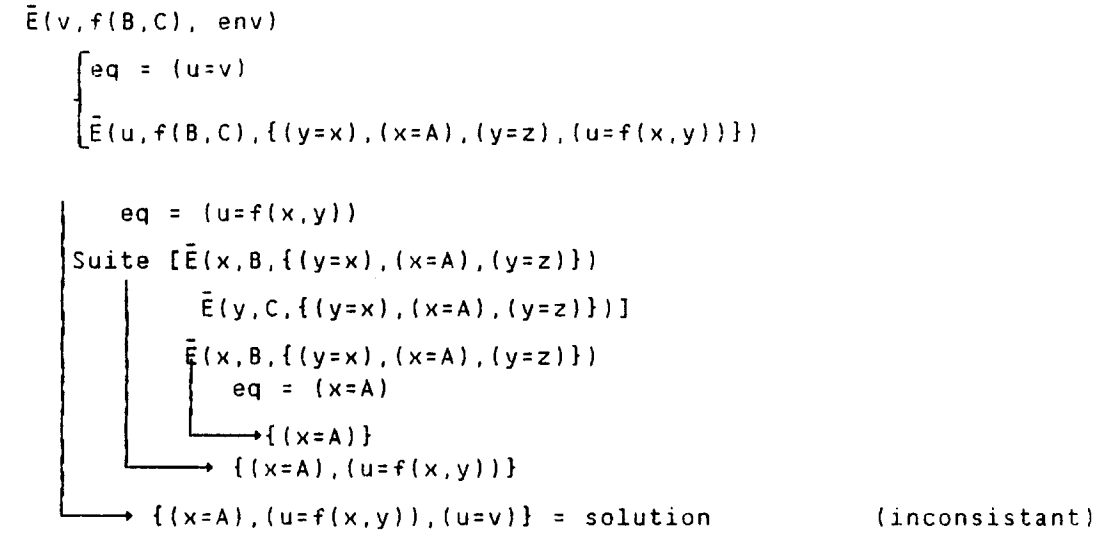
- eq (x=u)
- Ē (x,5,{(y=x),(x=A),(v=w)})
  - eq = (x=A)
  - → {(x=A)}

→ {(x=A),(x=u)} = solution (inconsistant)

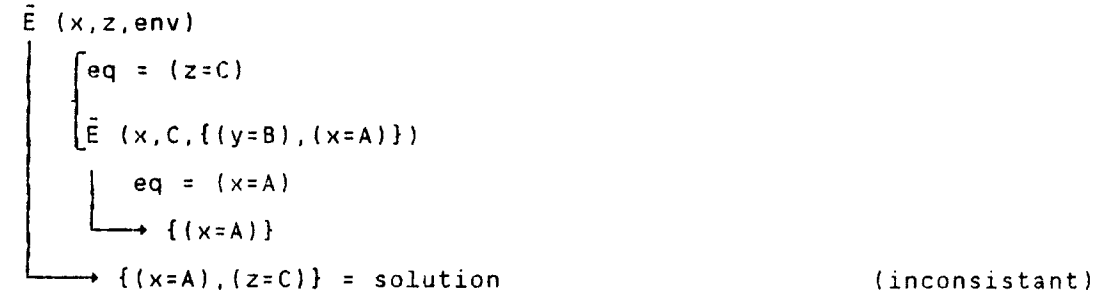
Exemple 3

env = {(y=x),(x=A),(y=z),(u=f(x,y)),(u=v)}

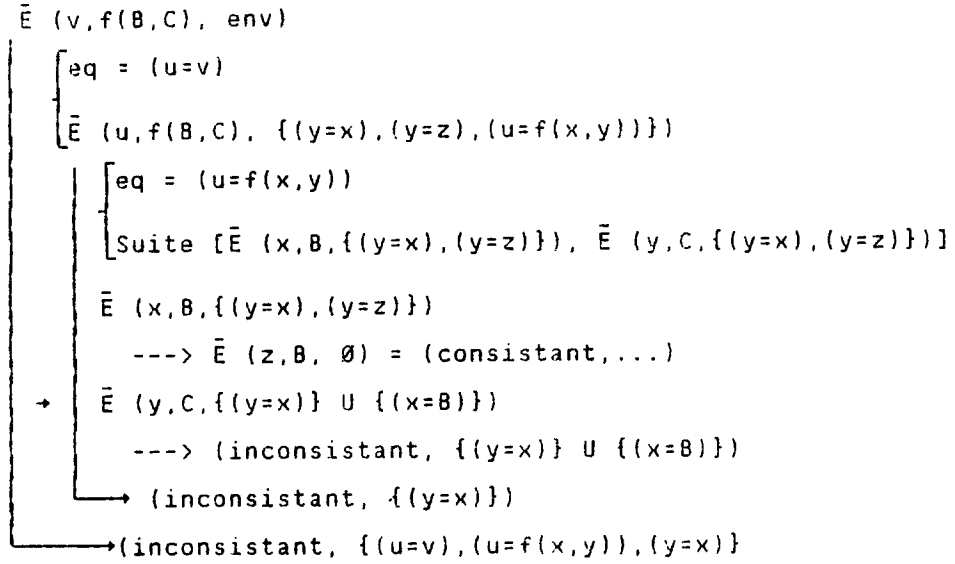
litt (N) = (E v f(B,C))



Exemple 4 env = {(y=B),(x=A),(z=C)}
litt(N) = (E x z)



Exemple 5 env = {(y=x),(y=z),(u=f(x,y)),(u=v)}
litt(N) = (E v f(B,C))

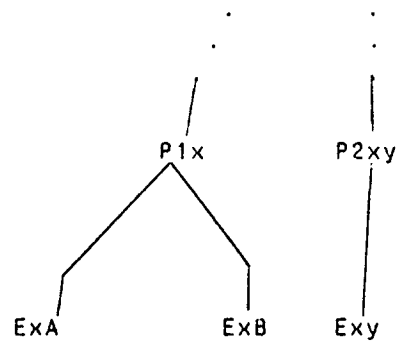


II.3.3.4.2.4 - <u>Présentation de la formule traduisant un échec feuille par niveaux de profondeur dans l'arbre</u>

L'expression logique caractérisant un échec feuille, obtenue en II.3.3.4.2.2., est une forme normale conjonctive C.

Dans ce paragraphe, nous présenterons un algorithme dont le but est de donner les différentes versions de cette expression, selon les différents niveaux de profondeur de l'arbre de racine P(x,...).
Pour "intégrer" ou "faire remonter dans l'arbre" l'expression de départ, on suppose que l'on dispose des informations permettant d'associer à chaque élément de la conjonction une liste d'ancêtres, ordonnée de celui de plus bas niveau dans l'arbre, jusqu'à la racine de l'arbre (cf. II.3.3.4.2.2) ; ainsi, si l'on remplace dans C chaque élément de la conjonction par son premier ancêtre associé, on obtient une forme C' conjonctive pure (ie dont aucun terme n'est une disjonction).

Exemple :



si C = (x=A V x=B) Λ (x=y) Λ ...

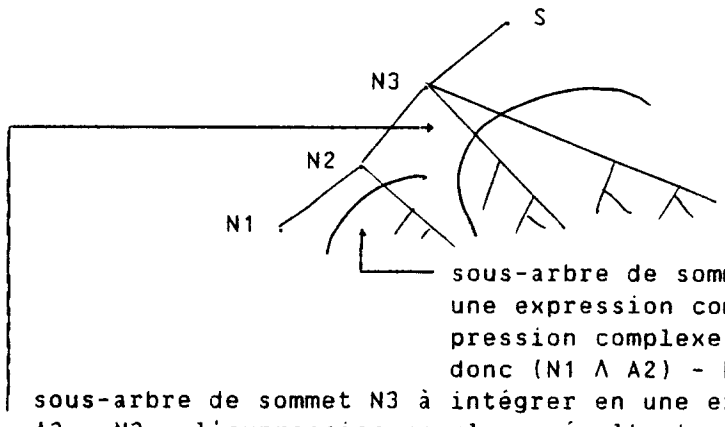
C' = P1x Λ P2xy Λ ...

L'idée de base de l'algorithme suivant est similaire à ce que nous avions proposé pour la logique des propositions. Il s'agit d'écrire une expression complexe de la forme

$$Q'(N) = Q'_1 - N$$

où N est le nom du noeud par rapport auquel l'expression est présentée, et Q'_1 est de la forme de Q'(N) ; c'est une formule "structurée par niveaux".

Pour constituer cette expression, nous raisonnons de façon récursive comme suit. La formule initiale définissant un arbre ET, on "intègre" l'expression par rapport à la branche la plus à gauche de l'arbre,, successivement de la feuille vers le sommet (celui-ci étant la racine de l'arbre ET-OU de résolution initial).



Algorithme d'intégration d'une expression conjonctive

Notations employées :

E = ensemble des littéraux (feuilles)
S = sommet de l'arbre (Et - forme normalisée)
Ai = ancêtres (dans l'ordre du plus jeune au plus vieux)

Remonter - arbre (E,S) <== let* l_1 = car(E)
A_1 = ancêtre le plus récent de l_1
in Remonter - exp (l_1 - A_1, E\l_1,S)

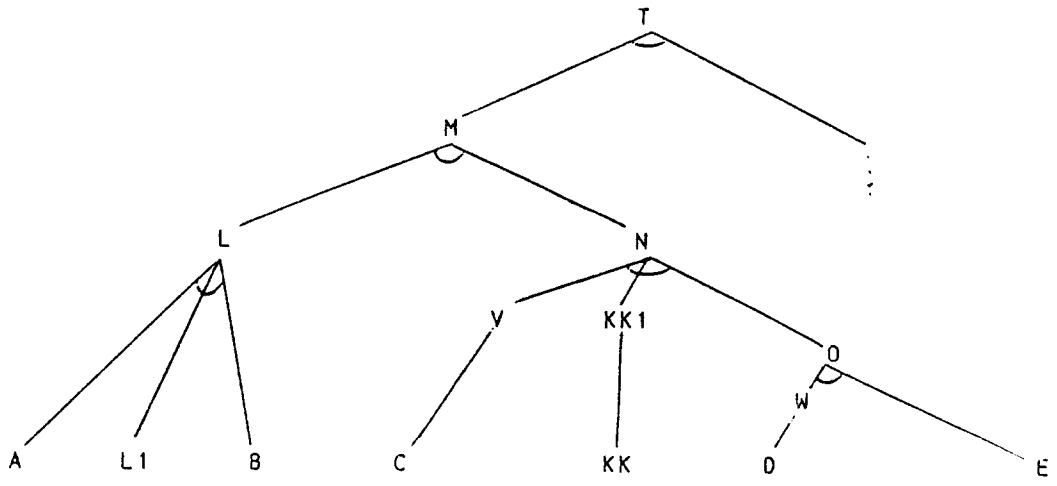
Remonter - exp (l_1 - A_1,E,S) <====

let E' = {descendants de A_1} $\cap$ E

si E' $\neq \emptyset$ alors let* l_2 - A_1 = R.arbre(E',A_1)
e'_1 = ($l_1 \wedge l_2$)
in Remonter - exp (e'_1 - A_1,E\E',S)
sinon % sous-exp. de racine A_1 constituée %
let* A_2 = ancêtre le plus récent de A_1
e'_1 = l_1 - A_1
in Remonter - exp (e'_1 - A_2,E,S)

Exemple d'application de cet algorithme

Soit l'arbre suivant, où seuls les noeuds ancêtres des feuilles formant l'expression de base sont représentés.



E = A ∧ L1 ∧ B ∧ C ∧ KK ∧ D ∧ E ∧ ...

(seules les feuilles intervenant dans E ont été représentées)

L'expression cherchée se construira pas à pas comme suit ; en partant de A :

B - L

(L1 ∧ B) - L

(A ∧ L1 ∧ B) - L

C - V

KK-KK1

D - W

E - O

(D-W ∧ E) - O

(KK-KK1 ∧ (D-W ∧ E)- O)- N

(C-V ∧ KK-KK1 ∧ (D-W ∧ E)- O))- N

((A ∧ L1 ∧ B)- L ∧ (C-V ∧ KK-KK1 ∧ (D-W ∧ E)- O))- N)- M

etc...

II.3.3.4.3 - Constitution de l'expression générale exprimant que l'arbre de résolution est en échec

Nous allons appliquer ici exactement le même principe que pour le calcul des propositions. Il s'agit donc de définir une expression logique S' à partir des expressions caractérisant les feuilles en échec de l'arbre ; ensuite nous intègrerons la formule résultante, de façon à pouvoir la présenter de façon structurée à différents niveaux de profondeur de l'arbre.

Rappelons que dans l'arbre de résolution ET - OU du but en échec initial, les noeuds ou feuilles en échec sont caractérisé(e)s par un environnement de sortie noté FAUX ou $\square$ (inconsistant)

Nous allons d'abord indiquer comment constituer cette expression, puis nous ferons une application sur un petit exemple.

II.3.3.4.3.1 - Constitution de l'expression complexe

Soit ex'(N) l'expression complexe relative à un noeud N de l'arbre, supposé en échec.

Par propriétés des arbres ET-OU, on peut définir ex'(N) comme suit :

- si $N = \bigwedge_{i=1}^{n} Ni$, si $\exists io \leq n$ tq Nio soit en échec,

  alors ex'(N) = ex'(Nio)

- si $N = \bigvee_{i=1}^{n} Ni$, $ex'(N) = (\bigvee_{i=1}^{n} ex'(Ni)) - ex(N)$

* ex (N) = exp1(N) $\wedge$ litt(N), où
  - litt(N) est le littéral associé au noeud N
  - exp1 indique les "responsables" de l'échec à partir de l'environnement d'entrée de N.

(Rappel : en calcul des propositions, exp1 n'existait pas puisqu'il n'y avait pas de variable, et N était identifiable par litt(N).

* $\bigvee_{i=1}^{n}$ ex'(Ni) représente le détail de ex(N), à des profondeurs plus basses que N dans l'arbre de résolution.

(Rappel : c'était pareil en logique des propositions).

ex'(N) et ex'(Ni) ont la même forme.

Les expressions ex'(N) se constituent de façon ascendante, à partir des feuilles de l'arbre, comme suit :

a - Cas d'une feuille N (en échec dans l'arbre)

eq'(N) = eq(N) = exp1(N) $\wedge$ litt(N) ;

dans exp1(N), on encadrera tous les littéraux ancêtres de N dans l'arbre (ils sont situés à droite) ; ces littéraux encadrés n'interviendront pas dans une explication de l'échec sur N.

b - Cas d'un noeud N = $\bigvee_{i=1}^{n}$ Ni

si $\forall i = 1,\ldots,n$ ex'(Ni) = α - (exp1(Ni) $\wedge$ litt(Ni))

alors exp1(N) = $\bigvee_{i=1}^{n}$ intégrée (exp1(Ni)),

où intégrée (exp, N) calcule l'expression "intégrée" de exp par rapport au noeud N (ie, présentation de exp selon le "point de vue" N).

Or exp est de la forme A $\wedge$ B $\wedge$ ... $\wedge$ C ; cela traduit, au plus bas niveau, les responsables de l'échec dans l'environnement d'entrée, obtenus selon la stratégie employée en appliquant le principe de résolution.

Le calcul de intégrée (exp, N) consiste à supprimer de l'expression complexe exp tous les littéraux correspondant à des noeuds définissant un sous-arbre de racine N, s'il en existe (car ils ne font que "détailler" N).

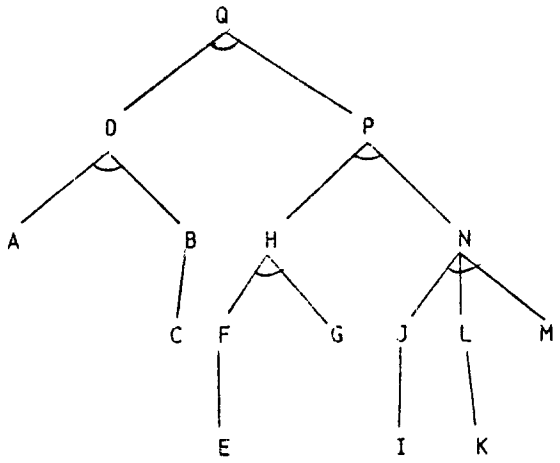
Pour calculer intégrée (exp(N)) = expi

- lire exp de droite à gauche ("à plat") jusqu'à repérer N (s'il y est) ; N étant un noeud sur la branche la plus à droite du sous-arbre, il suffit de ne chercher que parmi les littéraux encadrés.
- s'il y est, simplifier exp en enlevant la sous-expression complexe associée à N, et N ; on obtient le résultat cherché.
- sinon intégrée (exp(N)) = exp

Exemple :

exp = ((A ∧ B - C) - D ∧ ((E - F ∧ G) - H ∧ ((I - J ∧ K - L ∧ M) - [N] - [P]) - [Q]

représentée par l'arbre :



intégrée (exp, N) =

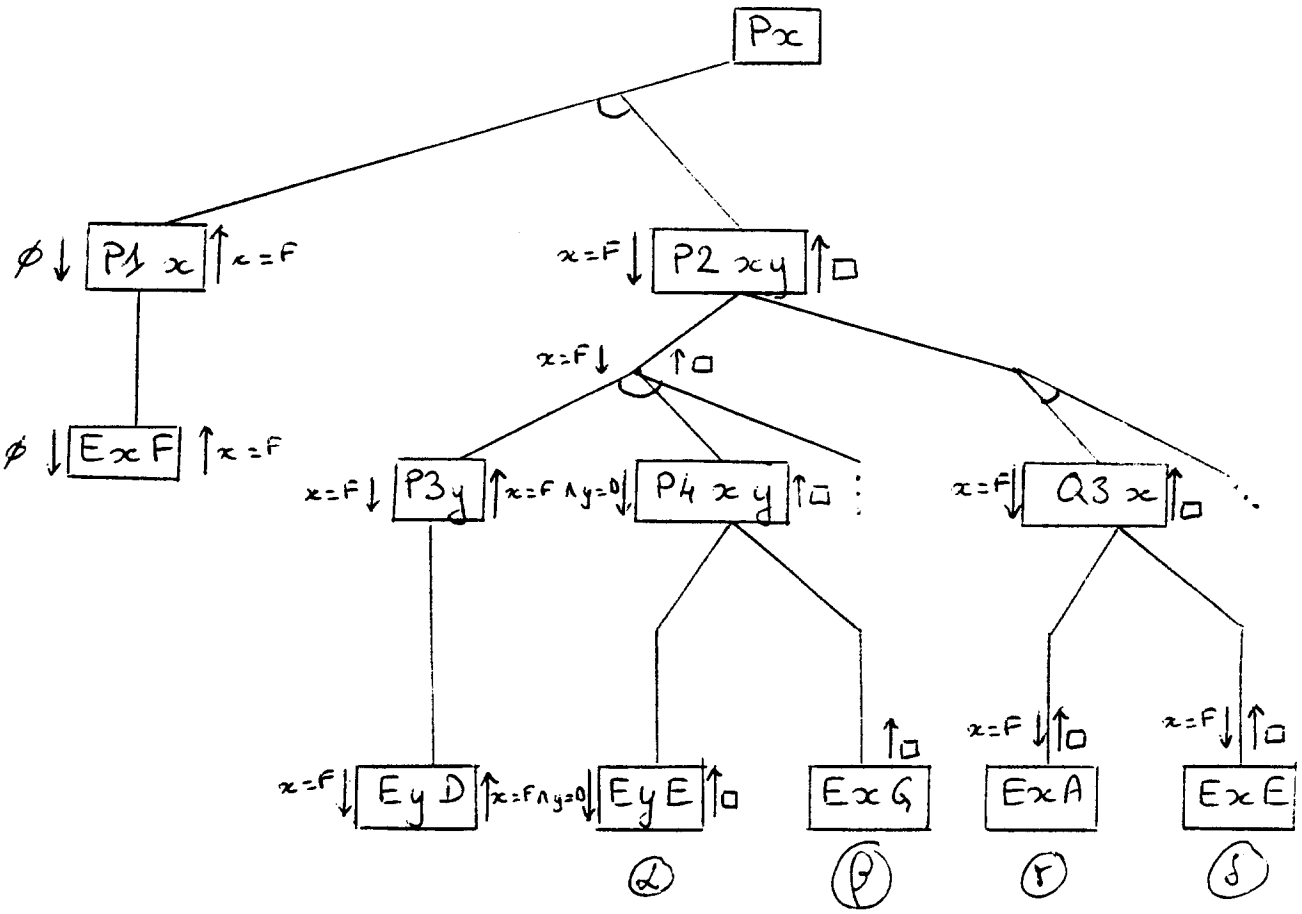
((A ∧ B - C) - D ∧ (E - F ∧ G) - H - [P]) - [Q]

Remarque :

Lorsque l'on génèrera les explications associées à ces expressions à l'utilisateur (en partant du sommet de l'arbre de résolution), on ignorera la présence des littéraux encadrés (ils ne servent qu'à faire des simplifications au cours des intégrations d'expressions au cours de la récursivité).

II.3.3.4.3.2 - Exemple d'application

Soit la portion d'arbre ET - OU de résolution de P(x,...) suivante :



Les formules d'échec associées aux feuilles en échec sont :

ex'(α) : EyD - P3y - $\boxed{P2xy}$ - $\boxed{Px}$ ∧ EyE

ex'(β) : ExF - P1x - $\boxed{Px}$ ∧ ExG

ex'(γ) : ExF - P1x - $\boxed{Px}$ ∧ ExA

ex'(∂) : ExF - P1x - $\boxed{Px}$ ∧ ExE

ex'(P4xy) :

[(EyD - P3y - $\boxed{P2xy}$ - $\boxed{Px}$ ∧ EyE) ∨ (ExF - P1x - $\boxed{Px}$ ∧ ExG)] -

(EyD - P3y - $\boxed{P2xy}$ - $\boxed{Px}$ ∨ ExF - P1x - $\boxed{Px}$) ∧ P4xy

ex'(Q3x) :

[(ExF - P1x - $\boxed{Px}$ ∧ ExA) ∨ (ExF - P1x - $\boxed{Px}$ ∧ ExE)] -

(ExF - P1x - $\boxed{Px}$ ∨ ~~ExF - P1x - $\boxed{Px}$~~) ∧ Q3x

ex' (P2xy) =

= (ex'(P4 xy) ∨ ex' (Q3x)) -

(~~∅~~∧(ExF - P1x - $\boxed{Px}$) ∨ (~~ExF - P1x - Px~~)) ∧ P2xy

= (ex'(P4 xy) ∨ ex' (Q3x)) - (ExF - P1x - $\boxed{Px}$ ∧ P2xy)

ex'(Px) = ex' (P2xy)

Nous pouvons alors expliquer l'échec en Px comme suit (en anticipant un peu sur la prochaine étape ...).

∃ x tq Px car

∃(x,y)tq P1x ∧ P2xy

(ou ∃(x,y)tq ExF ∧ P2xy)

car

d'une part :

∃(x,y) tq (P3y (ou EyD) ∨ P1x (ou ExF)) ∧ P4xy

(on peut raffiner l'explication)

d'autre part :

∃(x,y) tq P1x (ou ExF) ∧ Q3x

(on peut raffiner l'explication)

etc ...

II.3.3.5 - Formation de la formule finale expliquant l'échec

L'expression complexe obtenue en II.3.3.4 permet de donner une condition nécessaire pour que P(x ...), racine de l'arbre, soit vrai. Cette expression est structurée par niveaux de détails.

En suivant exactement le même principe qu'en logique des propositions, il ne reste plus qu'à nier cette expression pour obtenir une condition suffisante (et en même temps, une démonstration, et une explication) pour l'échec.

Remarque : 1 - c'est ce que nous avons fait dans l'exemple précédent.

2 - dans l'expression niée, les variables sont implicitement quantifiées universellement

(l'expression non niée correspondait à un but qu'il aurait été nécessaire d'effacer pour prouver la racine de l'arbre).

II.3.3.6 - Un exemple complet d'application de la méthode

Nous reprenons ici l'exemple sur lequel nous avons appliqué la méthode de Morgan (cf. I.3.4.1.2).

La question est : pourquoi-pas couple (Paul, Marie) ?

II.3.3.6.1 - Normalisation des clauses du programme

C1 couple (x,y) V ¬ mêmesgoûts (x,y)

C2 mêmesgoûts (x,y) V ¬goûts (x,m,s) V ¬goûts (y,m,s)

C3 goûts (x,m,l) V ¬aimemusique (x,m) V ¬aimesport (x,s)

C4 aimemusique (x,y) V ¬E (x,Paul) V ¬E (y,classique)

C5 aimemusique (x,y) V ¬E (x,Marie) V ¬E (y,classique)

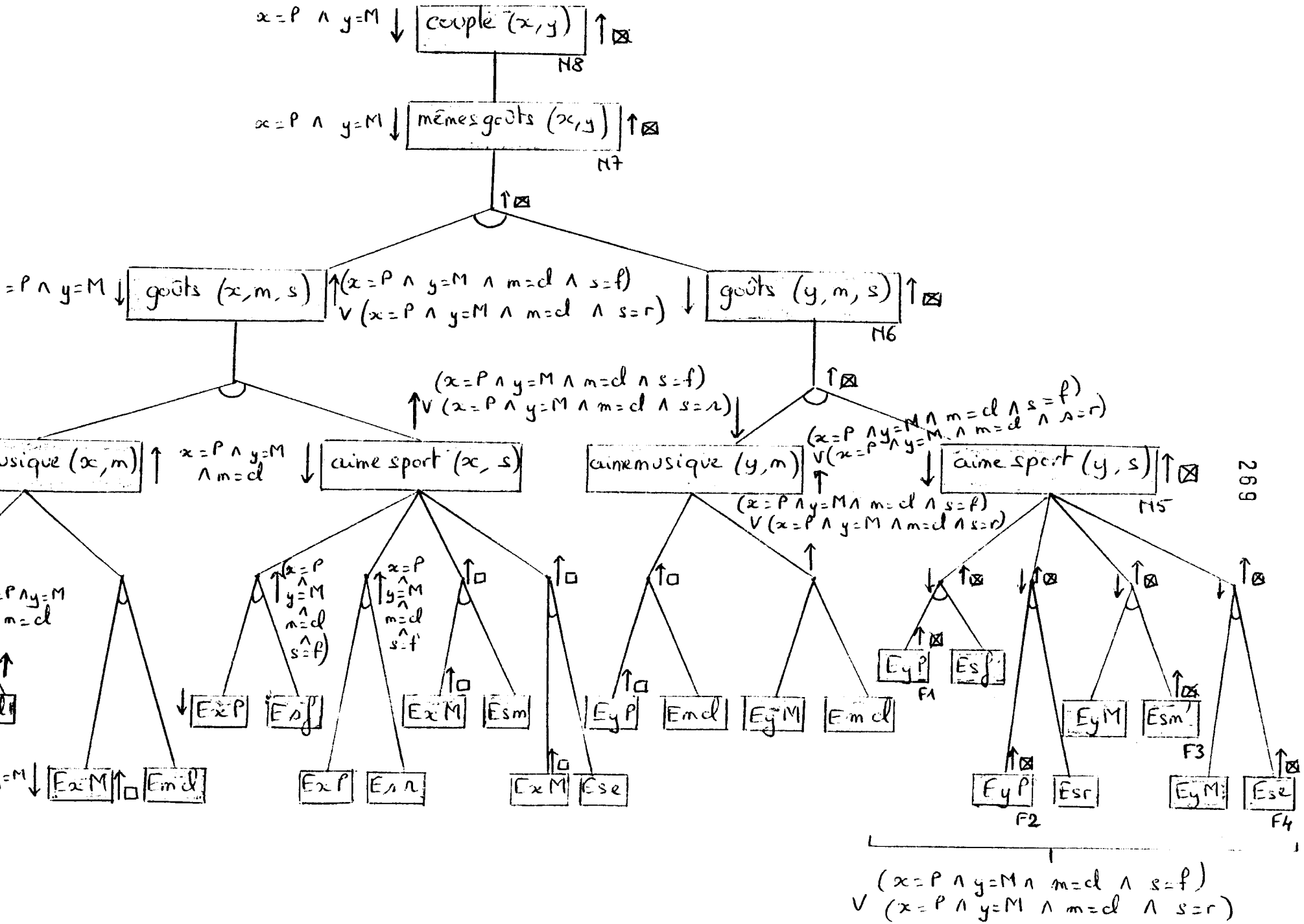
C6 aimesport (x,y) V ¬E (x,Paul) V ¬E (y,foot)

C7 aimesport (x,y) V ¬E (x,Paul) V ¬E (y,rugby)

C8 aimesport (x,y) V ¬E (x,Marie) V ¬E (y,marche)

C7 aimesport (x,y) V ¬E (x,Marie) V ¬E (y,escalade).

II.3.3.6.2 - Construction de l'arbre ET-OU de résolution de couple (Paul, Marie)


x=P ∧ y=M
couple (x,y)
N8
x=P ∧ y=M
mêmes goûts (x,y)
N7
goûts (x,m,s)
(x=P ∧ y=M ∧ m=cl ∧ s=f)
V (x=P ∧ y=M ∧ m=cl ∧ s=r)
goûts (y,m,s)
N6
(x=P ∧ y=M ∧ m=cl ∧ s=f)
V (x=P ∧ y=M ∧ m=cl ∧ s=r)
x=P ∧ y=M
∧ m=cl
aime sport (x,s)
aimemusique (y,m)
(x=P ∧ y=M ∧ m=cl ∧ s=f)
V (x=P ∧ y=M ∧ m=cl ∧ s=r)
aime sport (y,s)
N5
Ey P
Es f
F1
Ey M
Esm
F3
Ey P
Esr
F2
Ey M
Ese
F4
(x=P ∧ y=M ∧ m=cl ∧ s=f)
V (x=P ∧ y=M ∧ m=cl ∧ s=r)

II.3.3.6.3 - Expressions logiques caractérisant les feuilles en échec

Les feuilles en échec sont caractérisées par un environnement de sortie inconsistant ($\square$). Une formule minimale impliquée par celle qui définit l'environnement d'entrée, et telle que sa conjonction avec le littéral associé à la feuille en échec soit inconsistante, est évaluée au cours de l'élaboration de l'arbre, pour chaque feuille en échec (lors de la détermination de l'environnement de sortie. (cf. fonction "inconsistant") (cf. II.3.4.4.1.2).

On va bâtir, à partir de là (et de la liste des ancêtres associée à chaque sous-terme de l'expression formée), une expression complexe caractérisant la feuille en échec (cf. II.3.4.4.1.3). Si on le fait au cours de l'élaboration de l'arbre, on le fera systématiquement pour toutes les feuilles en échec ; or ne c'est pas utile (voir schéma III.3.3.6.2 : les feuilles en échec marquées d'un * n'interviendront pas dans l'explication). Une autre solution consiste donc à le faire au cours de la seconde passe sur l'arbre, destinée uniquement à bâtir l'expression générale traduisant l'échec sur l'arbre.

Dans la suite de ce paragraphe, nous ne formerons que les expressions caractérisant les feuilles en échec "responsables" de l'échec général sur l'arbre. Les noeuds participant à l'échec sont marqués sur le schéma du symbole $\square$ (environnement de sortie).

Constitution des sous-formules

Notons env-e l'expression

$[(x=P) \wedge (y=M) \wedge (m=cl) \wedge (s=f)] \vee [(x=P) \wedge (y=M) \wedge (m=cl) \wedge (s=r)]$

$F1 = \bar{E}\ (y,P,env\text{-}e) = (y=M)$

$F2 = \bar{E}\ (y,P,env\text{-}e) = (y=M)$

$F3 = \bar{E}\ (s,m,env\text{-}e) = (s=f) \vee (s=r)$

$F4 = \bar{E}\ (s,e,env\text{-}e) = (s=f) \vee (s=r)$

D'autre part, les listes des ancêtres associés aux équations (ti=tj) sont :

pour (y=M) : ∅ (environnement d'entrée de la racine)

pour (s=f) V (s=r) :

aimesport(x,s) - goûts(x,m,s) - mêmesgoûts(x,y) - couple(x,y)

Expressions complexes caractérisant les feuilles en échec

pour F1 et F2 : y = Marie

pour F3 et F4 :

(s=f)v(s=r) - aimesport(x,s) - goûts(x,m,s) - mêmesgoûts(x,y) - couple(x,y)

Remarque : Les noeuds mêmesgoûts(x,y) et couple(x,y) sont communs aux "responsables de l'échec" dans l'environnement d'entrée et aux littéraux de feuilles en échec.

II.3.3.6.4 - Expression complexe traduisant l'échec sur l'arbre

associée à F1 :

f1 = (y=Marie) ∧ (y=Paul)

associée à F2 :

f2 = (y=Marie) ∧ (y=Paul)

associée à F3 :

f3 = (s=foot)V(s=rugby) - aimesport(x,s) - goûts(x,m,s) -

mêmesgoûts(x,y) - couple(x,y) ∧ (s=marche)

associée à F4 :

f4 = (s=foot)Vv(s=rugby)-aimesport(x,s)-goûts(x,m,s)-

mêmesgoûts(x,y) - couple(x,y) ∧ (s=escalade)

associée à N5 :

(f1 V f2 V f3 V f4) - (f'1 V f'2 V f'3 V f'4) ∧ aimesport (y,s)

en posant :

fi = f'i ∧ X

Remarque : Ici les intégrées des f'i sont égales aux f'i.
On peut simplifier cette expression complexe, obtenant :
(f1 V f3 V f4) - (f'1 V f'3) Λ aimsport (y,s)

associée à N6 :
(f1Vf3Vf4) - (f'1Vf'3) Λ aimesport(y,s) - (f'1 V f'3) Λ goûts (y,m,s).

Remarque : Ici les intégrées des f'i sont égales aux f'i.

associée à N7
(f1Vf3Vf4) - (f'1Vf'3) Λ aimesport(y,s) - (f'1Vf'3) Λ goûts(y,m,s)
- ∅ Λ mêmesgoûts(x,y)

Remarque : par l'intégration, f'1 et f'3 disparaissent car elles détaillent le littéral mêmesgoûts(x,y) associé à N7

associée à N8 :
expression associée à N7.

II.3.3.6.5 - Expression expliquant l'échec sur couple (Paul, Marie)

Pour l'obtenir, il suffit de nier l'expression associée à N8, cette première expression constituant une condition nécessaire pour que couple (Paul, Marie) soit vrai.

Les variables étant implicitement quantifiées universellement dans la formule finale, nous obtenons l'expression complexe suivante :

([(y≠Marie)v(y≠Paul)] Λ [[(s≠foot) Λ (s≠rugby)] -
¬aimesport(x,s) - ¬goûts(x,m,s) V s≠marche] Λ [[(s≠foot)
Λ (s≠rugby)] - ¬aimesport(x,s) - ¬goûts(x,m,s) V s≠escalade])
((y≠Marie) V [[(s≠foot) V (s≠rugby)] - ¬aimesport(x,s) -
¬goûts(x,m,s)]) V ¬aimesport(y,s)
((y≠Marie) V [[(s≠foot) V (s≠rugby)] - ¬aimesport(x,s) -
¬goûts(x,m,s)]) V ¬goûts(y,m,s)
¬mêmesgoûts(x,y).

En tenant compte de l'environnement d'entrée de la racine de

l'arbre, nous pouvons simplifier cette équation complexe. Pour en simplifier l'interprétation, nous allons la réécrire en utilisant l'écriture $\alpha \Rightarrow \beta$ pour $\neg\alpha \vee \beta$. Nous obtenons donc :

(Marie ≠ Paul)
∧[[(s-foot ∨ s=rugby) - aimesport(Paul,s) - goûts(Paul,m,s)] =>
s≠marche]
∧[[(s-foot ∨ s=rugby) - aimesport(Paul,s) - goûts(Paul,m,s)] =>
s≠escalade]
-
[[(s=foot ∨ s=rugby) - aimesport(Paul,s) - goûts(Paul,m,s)] =>
¬aimesport(Marie,s)]
-
[[(s=foot ∨ s=rugby) - aimesport(Paul,s) - goûts(Paul,m,s)] =>
¬goûts(Marie,m,s)]
-
¬mêmesgoûts(Paul,Marie).

II.3.3.6.6 - Exploitation de cette expression complexe pour générer des explications négatives structurées par niveau de couple(Paul,Marie)

* Pourquoi-pas couple(Paul,Marie) ? (ou:pourquoi ¬couple(Paul,Marie)?])
parce que : ¬mêmesgoûts(Paul, Marie)
* Pourquoi ¬mêmesgoûts(Paul, Marie) ?
parce que : ∀(m,s) goûts(Paul,m,s) => ¬goûts(Marie,m,s) (α)
(ou : ∃ m,s tq goûts(Paul,m,s) ∧ goûts(Marie,m,s)

Remarque : A ce niveau, on peut détailler sur goûts (Paul,m,s) :
∀s, aimesport(Paul,s) => ¬goûts(Marie,m,s)
ou : (s=foot ∨ s=rugby) => ¬goûts(Marie,m,s).

* Pourquoi α ?
parce que : ∀m,p goûts(Paul,m,s) => ¬aimesport(Marie,s) (β)
(détail : ∀s aimesport(Paul,s) => ¬aimesport(Marie,s)
ou : (s=foot ∨ s=rugby) => ¬aimesport(Marie,s)).

* Pourquoi β ?

parce que :

Marie ≠ Paul

et ∀s,m goûts(Paul,m,s) => s≠marche

(détail : ∀s aimesport(Paul,s) => s≠marche

(s=foot V s=rugby) => s≠marche)

et ∀s,m goûts(Paul,m,s) => s≠escalade

(détail : ∀s aimesport(Paul,s) => s≠escalade

(s=foot V s=rugby) => s≠marche)

II.3.3.7 - Résultats obtenus avec cette méthode sur l'exemple pris en II.3.3.1.2.2.

Dans ce paragraphe nous ne donnerons que les expressions complexes associées aux différents noeuds en échec d'un arbre de résolution, et pas encore niées.

Remarque : Nous marquerons les expressions comme associées aux littéraux des noeuds (pour simplifier l'écriture) ; mais il s'agit bien de noeud et non de littéral.

(Exemple : Le littéral ExA est associé à plusieurs feuilles de l'arbre, celui dont nous parlons est attaché à la feuille en échec encadrée dans la représentation de l'arbre faite en II.3.3.3.1.2.2).

Pour ExA : (x=F) - P1x - $\boxed{Px}$ ∧ (x=A) (f1)

Pour ExE : (x=F) - P1x - $\boxed{Px}$ ∧ x=E (f2)

Pour Q3x : (f1 V f2) - ((x=F) - P1x - $\boxed{Px}$ ∧ Q3x) (f3)

Pour P2xy : (f1 V f1) - ((x=f) - P1x - $\boxed{Px}$ ∧ Q3x) -

((x=F) - P1x - $\boxed{Px}$ ∧ P2xy) (f4)

Pour EuA : (u=D) - P31xu - $\boxed{P3x}$ - $\boxed{Px}$ ∧ (u=A)) (f5)

Pour EuE : (u=D) - P31xu - $\boxed{P3x}$ - $\boxed{Px}$ Λ (u=E)) (f6)

Pour Q3u : (f5 V f6) - ((u=D) - P31xu - $\boxed{P3x}$ - $\boxed{Px}$ Λ Q3u) (f7)

Pour P3x : (f5 V f6) - ((u=D) - P31xu - $\boxed{P3x}$ - $\boxed{Px}$ Λ Q3u) - P3x (f8)

Pour Px : (f4 V f8) - Px (f9)

II.4 - CONCLUSION

Dans un premier temps, nous avons fait un tour d'horizon sur les diverses techniques mises en oeuvre pour générer des explications dans les systèmes à base de connaissances exprimées en logique. En ce qui concerne les explications négatives, et surtout en logique des prédicats, les travaux portant directement sur ce thème sont très rares. Par contre, nous avons pu effectuer des rapprochements intéressants avec des travaux dont l'objectif général était autre, mais qui pouvaient se rattacher à notre problème.

Pour notre part, nous avons réussi à répondre à la question pourquoi pas, en travaillant à partir du principe de résolution de Robinson. Nous avons proposé une méthode de génération d'explications négatives s'appuyant sur une représentation de la résolution d'un problème (en échec) par un arbre ET-OU, sur les hypothèses de monde fermé, et de non-égalité entre des constantes de noms différents.

L'idée sous-jacente à cette méthode est assez simple et a été exposée dans le cadre de la restriction à la logique des propositions. Son extension à la logique des prédicats restreinte aux clauses de Horn s'est révélée être tout à fait envisageable - même si elle a pu paraître longue à exposer ...

Les explications d'échec que nous pouvons générer sont structurées par niveaux de détails, de façon modulaire (ie : chaque composant, de la réponse peut être détaillé). Par ailleurs, nous avons d'une certaine mesure évité le problème de l'explosion combinatoire ; en effet nous ne nous intéressons pas à tous les échecs rencontrés au cours de la résolution ; mais nous en faisons la synthèse, pour ne considérer que ceux qui déterminent réellement l'échec général de la résolution, la formalisation de celle-ci en un arbre ET-OU nous permettant de le faire.

Nous obtenons des résultats satisfaisants, qui sont en particulier comparables avec ceux obtenus avec la f-résolution de Morgan. Toutefois, si notre méthode est intuitivement plus "directe" à mettre en oeuvre, elle n'en est pas moins laborieuse. D'autre part, rappelons des différences importantes entre notre méthode et celle de

Morgan ;

- étant donnés une base de connaissances BdC et un but B en échec, Morgan cherche quelques informations il suffirait de rajouter à BdC pour pouvoir montrer B ; nous cherchons plutôt quelles informations il est nécessaire de rajouter à BdC pour pouvoir montrer B (mais elles ne seront pas généralement suffisantes).

- Morgan traite le problème de l'égalité entre constantes par le biais de son algorithme d'unification étendu ; nous le traitons en normalisant nos clauses, et par une hypothèse à ce sujet.

Pour permettre une application sur des cas concrets, de nombreux points peuvent être étudiés encore dans le cadre de notre méthode. Par exemple, nous ne traitons pas les prédicats prédéfinis (exemple : calcul arithmétique). D'autre part, certains problèmes particuliers évoqués à la suite de l'élaboration de Pélagie n'ont pas été abordés dans ce travail (prédicats significatifs, prédicats usagers etc ...). Il resterait aussi, bien sûr, à implémenter la méthode et pouvoir juger de son efficacité.

CONCLUSION GENERALE

Nous nous sommes intéressés à un problème courant en Génie Logiciel, celui de l'assistance à la conception de logiciels à l'aide de bibliothèques de sous-programmes. Nous avons d'abord montré que certaines techniques, utilisées en Intelligence Artificielle, pouvaient être là d'un grand intérêt. Nous avons alors donné les spécifications détaillées d'un système expert pour le domaine de l'analyse numérique ; les informations à traiter sont formalisées en logique des prédicats restreinte aux clauses de Horn, le moteur d'inférences est basé sur PROLOG. Nous avons ensuite conçu une maquette dont les résultats ont été jugés très positifs par des utilisateurs, puisqu'ils doivent conduire à l'élaboration d'un système grandeur nature ; toutefois, toutes les fonctions n'ont pas été réalisées : en particulier, l'interface avec le gestionnaire du système a été complètement délaissé ; d'autre part, certains problèmes que nous avons rencontrés dans cette étude ont été résolus d'une façon qui pourrait être améliorée : par exemple, le traitement des "coefficients de réussite" affectés à une solution, ou celui de critères d'égale importance sur la solution attendue par l'utilisateur. Un intérêt du modèle général de représentation des informations que nous avons choisi, d'un point de vue conceptuel, est de ne pas sembler exclusivement applicable au domaine de l'analyse numérique. Nous avons donc montré que les systèmes experts pouvaient être d'un grand intérêt pour l'aide à la conception de logiciels. Depuis le début de cette étude, nous avons d'ailleurs rencontré d'autres besoins dans cette voie ... (/LAU 87/).

Au cours du développement de ce système, nous avons rencontré des difficultés pour expliquer pourquoi un sous-programme escompté par l'usager n'était pas recommandé par le système. Nous nous sommes donc penchés sur le problème de la génération des explications négatives, dans le cas où les connaissances peuvent être formalisées en clauses de Horn. Nous avons donc montré qu'une première approche, ne s'appuyant que sur la gestion des résolvantes d'une résolution, était vouée à l'échec, car la structure du raisonnement est partiellement perdue avec un tel formalisme. Nous avons finalement proposé une méthode s'appuyant sur le

concept des arbres ET-OU pour représenter une résolution de problème. Nous parvenons ainsi à générer des explications négatives structurées par niveau de détail - donc, nous pouvons éviter le problème de l'explosion combinatoire introduit par la logique des prédicats. Par cette méthode, nous obtenons des résultats comparables en particulier à ceux que l'on peut obtenir avec la f-résolution de MORGAN en génération d'hypothèses. Par rapport à nos objectifs initiaux, nous n'avons pas étudié tous les problèmes : cas de prédicats arithmétiques, prise en compte de la sémantique des littéraux ... lors de la formulation des explications. D'autre part, la méthode que nous avons proposée ne peut s'appliquer que lorsque la stratégie de résolution de problème est complète ; elle ne peut donc pas s'appliquer directement sur des bases de connaissances destinées à être interprétées par un outil comme PROLOG, la présence de certains prédicats de contrôle (comme le fameux SLASH) ne permettant plus de faire cette hypothèse. Enfin, les problèmes d'implémentations liés à notre méthode n'ont pas été abordés. Cette méthode donne donc un premier niveau de réponse à la question "pourquoi-pas?", ne dépendant strictement que des informations contenues dans la base de connaissances utilisée en résolution de problème. Il reste maintenant à relever le niveau de ces réponses de façon à les rendre plus synthétiques, et plus proches de la façon naturelle de penser de l'homme.

BIBLIOGRAPHIE

/ARO 78/ F. O'ARODES
Analyse et comparaison de performances des modules des bibliothèques d'interpolation.
Rapport de diplôme ENSEEIHT, 1978.

/BAR 77/ Avron BAAR
Meta-knowledge and cognition.
Proc. IJCAI p. 31, Vol 1.

/BEL 87/ Ali BELKHIRI
Une technique de backtracking intelligent basée sur un algorithme d'unification étendue et sur la datation du processus de résolution.
Journées de programmation en logique (CNET)
Trégastel, Mai 1987.

/BEN 86/ A. BENSAID, F. RECHENMANN, A. SIMONET, P. VIGNARD
Mécanismes d'inférences et d'explications dans les bases de connaissances centrées objet.
8èmes Journées francophones sur l'informatique. Bases de données et bases de connaissances.
Grenoble, 14, 15 janvier 1986.

/BON 81/ A. BONNET
Application de l'intelligence artificielle : les systèmes experts.
RAIRO Informatique, vol 15, n° 4, 1981, p. 325-341.

/BON 84-a/ A. BONNET
L'intelligence artificielle. Promesses et réalités.
Interéditions. 1984.

/BON 84-b/ A. BONNET
Systèmes experts et cognitique.
Rapport INRIA n° 97, 1984, p. 29-31.

/BON 85/ A. BONNET
Etude critique et synthèse sur les langages orientés objets (LOOBS) pour l'intelligence artificielle.
Génie Logiciel, ADI, n° 2, mai 1985, p. 29-36.

/BRA 82/ Ronald J. BRACHMAN, Hector J. LEVESQUE
Competence in knowledge representation.
Proc. AAI, 1982, p. 189.

/BRU 79/ M. BRU, J.L. LAGOUANELLE, P.M. MARSILI, G. SOUBRY
Le guide du programmeur en matière de propagation et de contrôle de l'erreur d'arrondi.
Rapport de convention 76/CNES/1141.

/BUC 84/ B.G. BUCHANAN, E.H. SHORTLIFFE
Rule-based expert systems.
Ed. Addision Wesley, 1984.

/BUC 84-a/ Explanation as a topic of artificial intelligence research in [BUC 84]. Chap. 17, p. 331.

/CAR 82/ J.G. CARBONELL
Experiental learning in analogical problem solving.
Proc. AAI 82, p. 168.

/CHA 73/ C.C. CHANG, R.C.T. Lee
Symbolic logic and mechanical theorem proving.
Academic Press, New-York, 1973.

/CHO 87/ Laurence CHOLVY
Interrogation d'une base de règles.
PRC Bases de données 3ème génération.
Rapport CERT-DERI n° 2/3274, Février 1987.

/CLA 83/ W.J. CLANCEY
The epistemology of a rule-based expert system : a framework for explanation.
Artificial Intelligence, 20, p. 215-251.

/CLA 82/ K.L. CLARK, F.G. Mc CABE
PROLOG : a langage for implementing expert systems
in Machine Intelligence 10.
(Ed. J.E. HAYES, D. MICHIES, V.H. PAO).

/COR 84/ M.O. CORDIER
Les systèmes experts.
La recherche n° 151, Janvier 1984, p. 60-70.

/DAV 84/ Randall DAVIS, Bruce G. BUCHANAN
Meta-level knowledge
in /BUC 84/, Chap. 28.

/DAV 84-a/ Randall DAVIS
Interactive transfer of expertise
in [BUC 84], Chap 9.

/DEM 82/ Luc DEMIANS D'ARCHIMBAUD
Introduction aux systèmes experts.
Revue technique THOMSON-CSF
Vol 14, n° 4, Décembre 1982.

/DHA 87/ Varant DHAR, Harry E. POPLE
Rule-based versus structure-based models for explaining and generating expert behavior.
Communications of ACM, Juin 1987, Vol 30 n° 6, p. 542-554.

/DIN 83/ M. DINCBAS
Contribution à l'étude des systèmes experts.
Thèse de Docteur Ingénieur.
ENSAE, Janvier 1983, Toulouse.

/DRE 78/ M. DREYFUS
FORTRAN IV. DUNOD Informatique. 1978.

/DUC 86/ Mireille DUCASSE
OPIUM : a sophisticated tracing tool for PROLOG.
Journées de programmation en logique (CNET)
Trégastel, Mai 1986.

/ERI 85/ Agneta ERIKSSON, Anna-Lena JOHANSON
Neat explanation of proof trees.
Proc. IJCAI 85, Vol 1, p. 379-381.

/FAH 79/ Ali FAHMI
Contrôle de systèmes de déduction automatique fondés sur la logique.
Thèse de Docteur Ingénieur, ENSAE, Novembre 1979, Toulouse.

/FAR 80/ H. FARRENY
Un système pour l'expression et la résolution de problèmes orienté vers le contrôle des robots.
Thèse d'Etat, U.P.S. Toulouse, Septembre 1980.

/FAR 82/ H. FARRENY
Notes de cours ENSEEIHT. 1981.

/FIK 85/ Richard FIKES, Tom KEHLER
The role of frame-based representation in reasoning.
Communication of ACM, Septembre 1985, Vol 28, n° 9, p. 904-920.

/FIN 79/ Nicholas FINDLER
Associative networks - Representation and use of knowledge by computers.
Academic Press, 1979.

/FOI 80/ J. FOISSEAU, R. JACQUART, M. LEMAITRE, M. LEMOINE, J.C. VIGNAT, G. ZANON
Program development with or without coding.
IFIP Congress, Nelbourne, Octobre 1980.

/FOR 78/ Normes de programmation FORTRAN IV, CII, 1978.

/FOR 79/ B. FORD, J. BENTLEY, J.J. DU CROZ, J.J. HAGUE
The NAG library "machine".
Software practice and experience, Vol 9, p. 65-72. 1979.

/FRI 81/ P. FRIELAND
Acquisition of procedural knowledge from domain experts.
Proc. IJCAI 7 (1981) Vol 2, p. 256.

/GAL 79/ H. GALLAIRE, Cl. LASSERRE
Controlling knowledge deduction in a declarative approach.
Proc. IJCAI 79, Tokyo, p. 51-56.

/GEN 85/ Michael R. GENESERETH, Matthew L. GINSBERG
Logic programming.
Communication of ACM, Septembre 1985, Vol 28, n° 9.

/GIA 85/ F. GIANNESINI, H. KANOUI, R. PASERO, M. VAN CANEGHEM
PROLOG.
Interéditions, 1985.

/HAND/ Avron BAAR, Edward A. FEIGENBAUM
The handbook of artificial intelligence.

/HAS 84/ Diane W. HASLING, W.J. CLANCEY, Gleen RENNELS
Strategic explanations for a diagnostic consultation system in Developments in expert systems. p. 117-131.
Ed. M.J. Coombs.

/HAY 85/ Frederic HAYES-ROTH
Rule-based systems.
Communication of ACM, Vol 28, n°9, p.921-932, Septembre 85.

/JAN 79/ Jürgen JANAS
On the feasability of informative answers.
Workshop on data-bases, Toulouse, CERT, 1979.

/KAS 86/ G. KASSEL
Expliquer, c'est raisonner sur le raisonnement : le système CQFE.
6èmes Journées Internationales d'Avignon, 1986.

/KEO 85/ Kathleen R. Mc KEOWN
The need for text generation.
AFIPS Conference proceedings
1985, National Computer Conference
15-18 juillet 1985, Chicago.

/KOD 85/ Yves KODRATOFF
Apprentissage symbolique automatique : l'approche de l'intelligence artificielle.
Enjeux n° 61, Septembre 1985.

/KOL 82/ J.L. KOLODNER
The role of experience in development of expertise.
Proc. AAI 1982, p. 273.

/LAU 82-1/ J.L. LAURIERE
Représentation et utilisation des connaissances : les systèmes experts à règles de production (1ère partie).
TSI RAIRO. Janvier, février 1982. Vol 1.

/LAU 82-2/ J.L. LAURIERE
Représentation et utilisation des connaissances : représentation des connaissances (2ème partie).
TSI RAIRO, Avril 1982, Vol 2.

/LAU 84/ Jean-Pierre LAURENT
La structure de contrôle dans les systèmes experts.
TSI Vol 3, n° 3, 1984, p. 161-177.

/LAU 87/ Patrick LAUG
Pilotage d'un code modulaire d'éléments finis par un système Expert.
Rapport INRIA n° 653
mars 1987

/LEN 44/ Douglas B. LENAT
Automated theory formation in mathematics.
Proc. IJCAI 77, Vol 2, p. 833-841.

/MAC 80/ MACLISP, manuel d'utilisation.
BULL, 1980.

/MAN 84/ J.C. MANGIN
Construction d'un système expert vue par l'expert : rapports avec l'ingénieur cognitif.
INRIA, 1984.

/MAS 84/ Ph. MASSE
"Représentation des connaissances des techniques des systèmes experts dans un logiciel C.A.O. de type éléments finis : FLUX-EXPERT".
Journées systèmes experts d'Avignon, 1984.

/MAZ 85/ P. MAZAS
LKJ-ex : système expert d'aide à la modification d'une base de connaissances.
5èmes Journées Internationales d'Avignon, 1985.

/MIN 74/ Marvin MINSKY
A framework for representing knowledge.
Massachussets institute of technology. Juin 1974.

/MIS 84/ Prateek MISHRA
Towards a theory of types in PROLOG.
Proc. IEEE. International symposium of logic programming.
p. 289-298. Atlantic City.

/MOR 75/ Charles G. MORGAN
Automated hypothesis generation using extended inductive resolution.
Proc. IJCAI 75, p. 351-356.

/MOR 79/ Alfred H. MORRIS
Development of mathematical software and mathematical software libraries.
Strategic systems department.
Dahlgren, Avril 1979.

/MOU 86/ Gérald MOULIS
Etude du langage mathématique en vue de sa manipulation informatique.
Rapport de DEA. ENSAE Toulouse. Septembre 1986.

/MYC 84/ Alan MYCROFT, Richard A. O'KEEFE
A polymorphic type system for PROLOG.
Artificial Intelligence n° 23 (1984) p. 295-307.

/NIL 80/ N. NILSSON
Principles of artificial intelligence.
Tioga Pub. Co., 1980.

/PAS 87/ Elsa PASCUAL
Interrogation d'une base de règles dans le cadre de base de données déductives.
Rapport de DEA, ENSAE Toulouse, Juin 1987.

/PIN 81/ S. PINSON
Représentation des connaissances dans les systèmes experts.
RAIRO Informatique. Vol 15, n° 4, 1981 p. 341-367.

/POL 82/ M.E. POLLACK, J. HIRSCHBERG, B. NEBBER
User participation in the reasoning process of expert systems.
Proc. AAI 82, Vol 2, p. 358.

/PRO 83/ PROTRAN evaluation guide - CDC
IMSL, Inc., Avril 1983.

/PUG 87/ J.M. PUGIN
Using integrity constraints to control hypothesis formation.
Journées de programmation en logique (CNET).
Trégastel, Mai 1987.

/REC 84/ F. RECHENMANN, A. BENSAID, D. GRANIER
SHIRKA : des systèmes experts centrés objets.
Journées systèmes experts d'Avignon, Mai 1984.

/RIC 83/ J.R. RICE
IMSL reference edition 1983.
Numerical methods, software and analysis.
Purdue University, 1983.

/ROB 65/ J.A. ROBINSON
A machine-oriented logic based on the resolution principle.
Journal of the association for computing machinery, Vol 12, n° 1, p. 23-41. January 1965.

/ROU 87/ M.C. ROUSSET, B. SAFAR
Negative and positive explanations in expert systems.
Applied Artificial Intelligence, n° 1, p. 25-28, 1987.

/SAF 85/ B. SAFAR
Explications dans les systèmes experts.
5èmes Journées Internationales d'Avignon, 1985.

/SAU 84-1/ Cl. SAUREL
Rapport Intermédiaire n° 1/322200/DERI. CERT, Toulouse.

/SAU 84-2/ Cl. SAUREL
Rapport Intermédiaire n° 2/322200/DERI. CERT, Toulouse.

/SAU 85/ Cl. SAUREL
Système expert d'assistance aux réalisateurs de logiciels scientifiques : présentation de la maquette PELAGIE.
Rapport final n° 4/3222/DERI, CERT, Toulouse.

/SAU 85-1/ Cl. SAUREL
Système expert d'assistance aux réalisateurs de logiciels scientifiques.
Journées Internationales d'Avignon, 1985.

/SAU 87/ Cl. SAUREL
EXPLINEG1 : une méthode de génération d'explications négatives dans les systèmes à bases de connaissances formalisées en logique des prédicats.
Journées de programmation en logique (CNET)
Trégastel, Mai 1987.

/SHA 82/ Ehud Y. SHAPIRO
Algorithm program debugging.
MIT Press. 1982.

/SCO 84/ A.C. SCOTT, W.J. CLANCEY, R. DAVIS, E.H. SHORTLIFFE
Methods for generating explanations.
in /BUC 84/, Chap. 18, p. 338.

/SWA 81/ W.R. SWARTOUT
Explaining and justifying expert consulting programs.
Proc. IJCAI 81, Vol 2, p. 815.

/SWA 83/ W.R. SWARTOUT
System for creating and explaining expert programs.
Artificial Intelligence 21, 1983.

/TES 85/ Cl. TESTEMALE
Un système de traitement d'informations incomplètes ou incertaines dans une base de données relationnelles.
Thèse de 3ème cycle. UPS Toulouse. Juin 1984.

/TUR 86/ R. TURNER
Logiques pour l'intelligence artificielle.
Ed. Masson, 1986.

/UTG 82/ Paul E. UTGOFF, Tom M. MITCHELL
Acquisition of appropriate bias for inductive concept learning.
Proc. AAI 82, p. 414.

/VIV 77/ Martial VIVET
Use of knowledge bases in automatic theorem proving and symbolic manipulations.
4th International Colloquium on advanced computing methods in theoretical physics.
St Maximin, Mars 1977.

/VIV 80/ Martial VIVET
Calcul formel et intelligence artificielle.
Colloque Intelligence Artificielle.
Caen 1980.

/VIV 83/ Martial VIVET
Un exemple d'utilisation de méta-règles : la sélection des plans dans CAMELIA.
Colloque d'Intelligence Artificielle.
Chambéry, 19-23 septembre 1983.

/VIV 84/ Martial VIVET
Expertise mathématique et informatique : CAMELIA, un logiciel pour raisonner et calculer.
Thèse d'Etat, Paris VI, Juin 1984.

/VIV 87/ Martial VIVET
Systèmes experts pour enseigner : méta-connaissances et explications.
Cognitiva 1987.

/WAL 83/ Adrian WALKER
Prolog / EX1, an inference engine which explains both Yes et No answers.
Proc. IJCAI 83. Vol 1, p. 526-528.

/WEI 82/ J.L. WEINER
BLAH, a system which explains its reasoning
Artificial Intelligence 15 (1980), p. 19-48.

/ZAN 84/ Guy ZANON
PROLISP : manuel d'utilisation.
Rapport 1/3617/DERI, CERT, Toulouse.

ANNEXES

ANNEXE 1

Tableaux récapitulatifs des principales informations contenues dans la base de connaissances de PELAGIE.

1 - Types de méthodes susceptibles de résoudre les problèmes traités dans PELAGIE.

2 à 5 Règles inter-objets (par type d'outil)

6 - Règles de comparaison entre objets de même nature suivant les contraintes sur performances.

7 - Règles de déduction entre propriétés.

Problème posé	Types de méthode permettant a priori de résoudre ce problème
résolution de système linéaire avec un second membre (une colonne)	directe - composite - itérative par points ou par blocs
résolution de système linéaire avec plusieurs seconds membres et / ou inversion de matrice	directe - composite - itérative par points ou par blocs
calcul de déterminant d'une matrice	directe - composite

Figure 1

Figure 2 : Règles inter-objets de choix de types de méthodes.

Caractéristiques de la matrice du problème	Ordre de la matrice	Conditionnement de la matrice	Problème(s) à résoudre	Type(s) de méthode adapté(s)	% de réussite espéré
Quelconque	⩽ 50	bon	*	directe - composite	95
		moyen	*	directe - composite	65
		mauvais	RSL - Inv	itérative/pts + contrôle qualité	10
			CD - DM	composite directe + contrôle qualité	10
		inconnu	*	directe-composite + contrôle qualité	10
	[50,100]	bon	*	directe-composite	90
		moyen	*	directe-composite	65
		mauvais	*	directe-composite + préconditionnement	50
		inconnu	*	directe-composite + contrôle qualité	50
	[100,250]	bon	*	directe-composite	90
		moyen	CD - DM	composite-directe	30
		moyen	RSL - Inv	itérative/pts	40
		mauvais	RSL - Inv	itérative+préconditionnement	30
			CD - DM	directe-composite + préconditionnement	10
		inconnu	RSL - Inv	itérative + contrôle qualité	30
			CD - DM	directe-composite + contrôle qualité	10
	[250,1000]	bon	*	composite	90
			RSL - Inv	itératives	90
		moyen	CD - DM	composite	20
			RSL - Inv	itératives	40
		mauvais	CD - DM	composite+préconditionnement	10
			RSL - Inv	itérative+préconditionnement	30
		inconnu	CD - DM	composite + contrôle qualité	10
			RSL - Inv	itérative + contrôle qualité	30

Caractéristiques de la matrice du problème	Ordre de la matrice	Conditionnement de la matrice	Problème(s) à résoudre	Type(s) de méthode adapté(s)	% de réussite espéré
"					
	> 1000	bon	RSL - Inv	itérative	70
			CD - DM	composite	30
		moyen	RSL - Inv	itérative	40
			CD - DM	composite	10
		mauvais	RSL - Inv	itératives + préconditionnement	30
			CD - DM	composite + préconditionnement	5
		inconnu	RSL - Inv	itératives+contrôle	30
			CD - DM	composite+contrôle	5
Symétrique	< 100	bon	*	directe - composite	95
		moyen	*	idem	70
		mauvais	RSL - Inv	composite-itérative	10
			CD - DM	composite	10
		inconnu	*	directe - composite + contrôle qualité	10
	100-500	bon	*	directe - composite	90
		moyen	*	idem	70
		mauvais	*	idem + préconditionnement	65
		inconnu	*	idem + contrôle qualité	65
	500-2000	bon	*	directe - composite	90
		moyen	RSL - Inv	itérative	50
			*	composite	50
		mauvais	RSL - Inv	- itérative + préconditionnement	40
			*	composite+préconditionnement	
		inconnu	RSL - Inv	- itérative + contrôle qualité	40
				composite+contrôle qualité	

Caractéristiques de la matrice du problème	Ordre de la matrice	Conditionnement de la matrice	Problème(s) à résoudre	Type(s) de méthode adapté(s)	% de réussite espéré
"	> 2000	bon	RSL - Inv	itératives	75
			CD - DM	composite	45
		moyen	RSL - Inv	itératives	50
			CD - DM	composite	30
		mauvais	RSL - Inv	itérative+préconditionnement	35
			CD - DM	composite+préconditionnement	10
		inconnu	RSL - Inv	itérative+contrôle qualité	35
			CD - DM	composite+contrôle	10
Définie-positive	< 50	bon	*	directe - composite	95
		moyen		idem	75
		mauvais,inconnu	*	idem+contrôle qualité	10
		mauvais	RSL - Inv	itérative+contrôle qualité	10
	100-500	bon		directe - composite	95
		moyen		"	75
		mauvais		idem+préconditionnement	60
		inconnu		idem+contrôle qualité	60
	500-2000	bon	*	directe - composite	95
		moyen	CD - DM	composite-itératives	55
			RSL - Inv	itératives	55
		mauvais	CD - DM	composite-itératives	50
			RSL - Inv	itératives + préconditionnement	50
		inconnu	CD - DM	composite-itératives	50
			RSL - Inv	itératives + contrôle qualité	50
	> 2000	bon	RSL - Inv	itératives	85
			CD - DM	directe - composite	50
		moyen	RSL - Inv	itérative	55
			CD - DM	directe - composite	35
		mauvais	RSL - Inv	itérative+précondi-	40
			CD - DM	directe - composite + préconditionnement	15
		inconnu	RSL - Inv	itérative+contrôle qualité	40
			CD - DM	directe - composite + contrôle qualité	15

Caractéristiques de la matrice du problème		Ordre de la matrice	Conditionnement de la matrice	Problème(s) à résoudre	Type(s) de méthode adapté(s)	% de réussite espéré
définie positive, creuse, non structurée		> 2000		RSL - Inv	Itératives+contrôle qualité	90
				CD - DM	directe - composite + contrôle qualité	20
Bande	txr ⩽ 10 %	< 50	bon	CD - DM	directe - composite	90
	txr ⩽ 10			RSL - Inv	directe - composite	90
	txr ⩾ 5			RSL - Inv	itérative	90
	txr > 10			CD - DM	directe - composite	70
	txr ⩽ 5		moyen	*	directe - composite	70
	txr > 5			RSL - Inv	itérative	70
	txr > 5			CD - DM	composite	50
	txr ⩽ 5		mauvais	*	directe - composite + préconditionnement	50
	txr > 5			RSL - Inv	itérative+précondi-	50
				CD - DM	directe-composite + préconditionnement	30
	txr ⩽ 5		inconnu	RSL - Inv	directe - composite + contrôle qualité	50
	txr > 5			RSL - Inv	itérative+contrôle qualité	50
	txr > 5			CD - DM	directe - composite + contrôle qualité	30
	txr > 10	50-500	bon	CD - DM	directe composite	70
	txr ⩾ 5			RSL - Inv	itérative	90
	txr ⩽ 10			*	directe - composite	90
	txr ⩽ 5		moyen	*	directe - composite	70
	txr ⩾ 5			CD - DM	directe - composite	30
	txr ⩾ 5			RSL - Inv	itérative	50
			mauvais	RSL - Inv	itérative+précondi-	30
				CD - DM	composite	15
	txr ⩽ 2	> 500	bon	*	directe - composite	90
	2 ⩽ txr ⩽ 5			*	directe - composite	80
	2 ⩽ txr ⩽ 5			RSL - Inv	itérative	80
	txr > 5			CD - DM	directe - composite	55
	txr > 5			RSL - Inv	itérative	75

Caractéristiques de la matrice du problème	Ordre de la matrice	Conditionnement de la matrice	Problème(s) à résoudre	Type(s) de méthode adapté(s)	% de réussite espéré
Bande txr ⩽ 2	> 500	moyen	RSL - Inv	itératives	40
txr ⩽ 2			CD - DM	composite	65
2 ⩽ txr ⩽ 5			RSL - Inv	itératives	30
2 ⩽ txr ⩽ 5			CD - DM	composite	60
5 ⩽ txr			RSL - Inv	itératives	25
5 ⩽ txr			CD - DM	composite	50
txr ⩽ 2		mauvais	RSL - Inv	itératives + préconditionnement	30
txr ⩽ 2			CD - DM	composite+précondi-	15
2 ⩽ txr ⩽ 5			RSL - Inv	itératives+précondi-	20
2 ⩽ txr ⩽ 5			CD - DM	composite+précondi-	10
5 ⩽ txr			RSL - Inv	itératives+précondi-	10
5 ⩽ txr			CD - DM	composite+précondi-	5
txr ⩽ 2		inconnu	RSL - Inv	itératives+contrôle qualité	30
txr ⩽ 2			CD - DM	composite+contrôle qualité	15
2 ⩽ txr ⩽ 5			RSL - Inv	itératives+contrôle qualité	20
2 ⩽ txr ⩽ 5			CD - DM	composite+contrôle qualité	10
5 ⩽ txr			RSL - Inv	itératives+contrôle qualité	10
5 ⩽ txr			CD - DM	composite+contrôle qualité	5

Figure 3 : Règles inter-objects de choix de méthodes

Type de méthode retenu	Caractéristiques de la matrice du problème	Ordre de la matrice	Problème(s) à résoudre	Méthodes adaptées	% de réussite
directe/composite	définie-positive	< 1000	*	décomposition-symétrique	95
	tridiagonale symétrique				
	txr > 5 %			décomp. symétrique	90
	txr < 5 %			décomp. triangulaire	65
				diagonalisation	10
	symétrique			décomposition symétrique	90
	bande symétrique				
	txr > 5 %			décomposition symétrique	90
	txr < 5 %			décomposition triangulaire	65
				diagonalisation	10
	bande (quelconque)				
	txr > 5			décomposition triangulaire	85
	txr < 5			décomposition triangulaire	
				diagonalisation	10
	* txr			décomposition triangulaire	85
	quelconque				
				diagonalisation	10
itérative/blocs	symétrique	> 1000		décomposition sym. /it.	50
	bande			décomposition trian-/it	50
	quelconque			Gauss-Seidel par bloc	50
itérative/pts				Jacobi	20
				Gradient conjugué	80
				Gauss-Seidel	70
				relaxation	80

Figure 4 : Règles inter-objets de choix d'algorithmes

Méthode retenue	Caractéristiques de la matrice du problème	Algorithmes retenus	% de réussite
Décomposition symétrique	définie - positive	LL^T	95
		LDL^T	95
	symétrique	LDT^T	90
	bande symétrique		
	txr ≥ 5	LDL^T	90
	bande définie - positive	LL^T	90
		LDL^T	90
Décomposition triangulaire	quelconque	Gauss pivot max.	90
		Crout pivot et équilibrage	90
		Gauss pivot partiel	
		Crout pivot sans équilibrage	85
		Gauss pivot direct	80
		Crout sans pivot	
		LDU	85
diagonalisation		diagonalisation	
Décomposition triangulaire	bande, txr < 5	Gauss pivot max	60
		Gauss pivot partiel	65
		Gauss pivot direct	70
		Crout pivot et équilibrage	60
		Crout pivot sans équilibrage	65
		Crout sans pivot	70
		LDU	65
		diagonalisation	50

Figure 5 : Règles inter-objets de choix de modules.

Algorithmes retenus	Caractéristiques de la matrice du problème	Problèmes résolubles	Modules adaptés	% de réussite
LL^T (dir.)	définie-positive,stockée	RSL	LEQT1P	95
LL^T bis (comp.)	"	RSL	LEQT2P	95
	définie-positive,stockée	RSL		
	en mode bande symétrique		LEQ1PB	95
LL^T-bis	"	RSL	LEQ2PB	95
Crout pivotage (dir.)	quelconque réelle	RSL	LEQT1F	85
équilibrage (comp.)	"	RSL	LEQT2F	85
LDL^T (dir.)	symétrique	RSL-Inv	LEQ1S	
(comp.)	"	RSL-Inv	LEQ2S	
LU équilibré (dir.)	bande stockée en mode	RSL-Inv	LEQT1B	65
et pivotage (comp.)	"	RSL-Inv	LEQT2B	65
LU pivotage (dir.)	quelconque complexe	RSL-Inv	LEQT1C	85
et équilibrage (comp.)	"	RSL-Inv	LEQT2C	85
Crout pivotage	quelconque réelle	Inv	LINV1F	85
(dir. et équili-	"	Inv	LINV2F	85
brage (comp.)				
LL^T (dir.)	définie-positive	Inv	LINV2P	95
(comp.)		Inv	LIN1PB	95
LL^T (dir.)	bande définie-positive	Inv	LIN1PB	95
	stockée en mode	Inv	LIN2PB	95
(comp.)	bande symétrique	Inv	LIN2PB	95
Crout pivotage et équilibrage (dir.)	quelconque réelle	Inv - RSL - CD	LINV3F	85
	"			
LL^T (dir.)	définie-positive stockée en mode symétrique	RSL - Inv - CD	LINV3P	95

Algorithmes retenus		Caractéristiques de la matrice du problème	Problèmes résolubles	Modules adaptés	% de réussite
Gauss pivot max	it/ blocs	N ⩽1500	RSL	LFQOF	50
"		"	"	LEQ1F	50
Crout pivotage équilibrage		quelconque	DM - CD	LUDATF	95
LL^T (direct)		définie-positive, stockée en mode symétrique	DM - CD	LUDECP	95
LL^T (direct		définie-positive, bande stockée en mode bande symétrique définie-positive		LUDAPB	95

Figure 6 - Règles de comparaison entre objets de même nature suivant les contraintes sur performances

Règle de comparaison entre types de méthodes suivant les contraintes sur performances.

Au point de vue précision, composite est préférable à Directe
Itérative(s) " Directe
Itérative par points est préférable à Itérative par blocs
(si convergence)

Au point de vue place mémoire Directe est préférable aux autres
Itérative par blocs est plus coûteux que Itérative par points

Au point de vue temps exécution Directe est préférable aux autres
Itérative par blocs est préférable à itérative par points

Règles de comparaison entre Méthodes sur contraintes performances

Aux points de vue - précision , la décomposition triangulaire est meilleure que la diagonalisation
- place mémoire
- temps exécution

Au point de vue temps exécution, , Jacobi est la pire des méthodes itératives
" , Gradient-conjugué est préférable à Gauss-Seidel

Au point de vue précision, Gauss-Seidel est préférable à Jacobi
" Gradient-conjugué est préférable à Gauss-Seidel

Au point de vue place mémoire, Gauss-Seidel est préférable à Jacobi

" Relaxation est préférable à Jacobi

" Relaxation est préférable à Gradient Conjugué

" Gauss-Seidel est préférable à Gradient conjugué

Au point de vue temps exécution, si ω(paramètre de relaxation)optimal connu, alors préférable à Gauss-Seidel (si les 2 convergent)

Au point de vue précision, si ω optimal connu, alors Relaxation préférable à Gauss-Seidel

si ω optimal inconnu, alors Gauss-Seidel préférable à Relaxation

Autres règles sur les méthodes (savoir-faire)

Si la matrice est à diagonale-dominante alors Jacobi et Gauss-Seidel convergent- Gauss-Seidel par blocs converge si Gauss-Seidel converge.

Si le rayon spectral d'une matrice spécifique à une méthode itérative est inférieur à 1, alors la méthode en question converge.

Règles de comparaison entre algorithme sur des contraintes performances

Crout sans pivot plus rapide que Crout avec pivot sans équilibrage

Crout avec pivot sans équilibrage plus rapide que Crout avec pivot avec équilibrage

Gauss pivot direct plus rapide que Gauss pivot partiel

Gauss pivot partiel plus rapide que Gauss pivot max

Au point de vue précision Gauss pivot total préférable à Gauss pivot-partiel

Gauss pivot partiel préférable à Gauss pivot direct

Crout pivot et équilibrage préférable à

Crout pivot sans équilibrage

Crout pivot sans équilibrage préférable à

Crout sans pivot

LDLT plus rapide que LLT

LDLT plus précis que LLT

Relaxation de comparaison entre modules suivant les contraintes sur performances

Au point de vue place mémoire :

LEQT1F	préférable à	LINV1F
LEQT2F	"	à LINV2F
LEQT1F	"	à LINV3F
LINV1F	"	à LINV3F
LEQT1F	"	à LINV1P
LINV2P	"	à LEQT2P
LINV3P	"	à LINV1P
LEQ1PB	"	à LIN1PB
LEQ2PB	"	à LIN2PB

Règles de déduction de propriétés

Une matrice tridiagonale est nécessairement bande (et inverse niée)

Si la largeur de bande d'une matrice, bande est inférieure à $\alpha(N)$
alors cette matrice est considérée comme creuse.

Si une matrice n'est ni symétrique
ni définie-positive
ni bande
ni creuse
alors on la considère comme quelconque

Si une matrice n'est pas creuse
alors on la considère comme pleine.

FIGURE 7

ANNEXE 2

Projet de fichier de mémorisation des sessions usagers

Enregistrement n° 1 - "identification" usager

- indicatif INDIC permettant de repérer la session (ex : n° de session, date) ex : quantième du mois, mois, année, n° de consultation dans le jour.

- caractaristiques de l'usager : nom, division, département, code projet ou laboratoire, etc...

Enregistrement n° 2 - description du problème

- indicatif
- code classe mathématique abordée (couvrant l'ensemble des domaines de l'analyse numérique)

  ex : arithmétique 01
  algèbre linéaire 02
  interpolation 03
  etc...

- code finalité de l'utilisation du Système Expert

  ex : enseignement 01
  consultation pour ces tests 02
  recherche méthode 03
  recherche SP (sous programme) 04
  etc...

- code détail du problème posé (s'inspirer de la classification SHARE). ex dans l'algèbre linéaire

  calcul déterminant 01
  résolution systlinéaire 02
  inversion de matrice 03
  etc...

Enregistrements suivants : n° 3 à n

- la résolution d'un problème donné se décompose en un certain nombre d'étapes (une ou plusieurs suivant le cas)
- la séquence des enregistrements n° 3 à n décrira chacune de ces étapes ; ces enregistrements sont multiples pour respecter ainsi les diverses séquences du dialogue entre le SE et l'usager
  ex : cas de la recherche d'un SP

étape n° 1 : enregistrements 3 multiples

- de code secondaire n° 1 à n : contiennent les éléments codifiés du dialogue avec l'usager permettant le choix d'une méthode possible. Ce seront par exemple les caractéristiques principales du problème.
- de code secondaire n° 10 : contient les éléments codifiés du résultat.
  ex : diverses méthodes retenues
- de code secondaire n° 90 et plus (éventuellement) : si demande d'informations de la part de l'usager (enseignement sur un point particulier du problème)

étape n° 2 : enregistrements 4 multiples

- de code secondaire n° 1 à n : contiennent les caractéristiques secondaires du problème (par ex) ayant permis le choix d'un algorithme (ou plusieurs)
- de code n° 10 : contient le résultat : algorithme(s) retenu(s)
- de code n° 90 : si enseignement à la demande de l'usager pour garder trace de points particuliers pour lesquels il a réclamé des compléments d'informations

étape n° 3 : enregistrements 5 multiples

- de code secondaire n° 1 à n : éléments du dialogue pour aider au choix du SP
- de code n° 10 : le ou les SP retenus
- de code n° 90 : enseignement

enregistrement n° 90

- INDIC
- indices traduisant les impressions de l'usager à l'issue de la consultation, et commentaires (texte)

enregistrements n° 50 et plus (multiples)

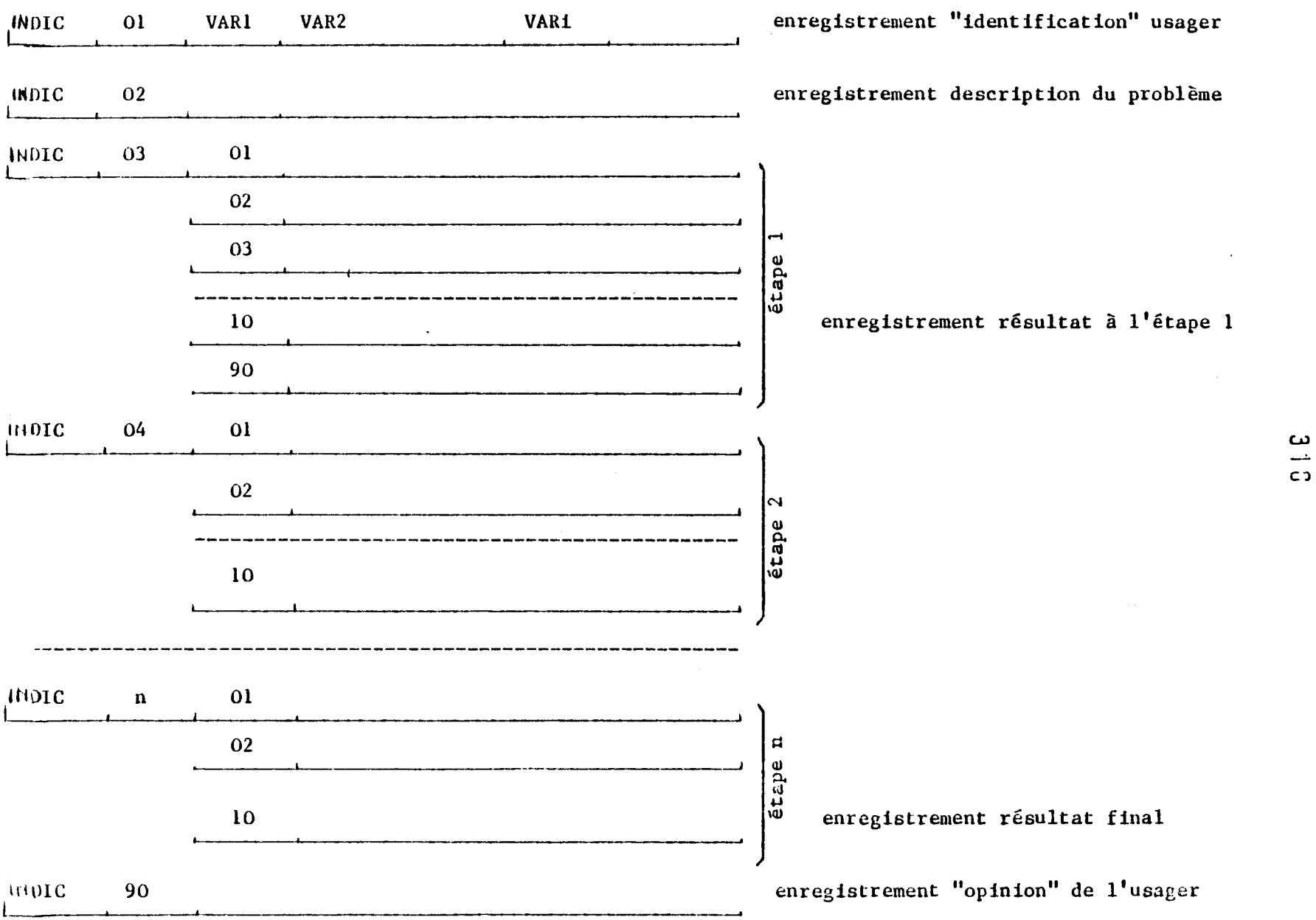
- INDIC

etc...

si poursuite d'une session déjà commencée.

Un schéma global de ce projet figure page suivante :

Pour une session : (consultation complète)



ANNEXE 3

Composition du code de PELAGIE

103 fonctions LISP

145 prédicats

936 clauses avec variables

en tout 156 Koctets (y compris l'interpréteur PROLISP)

environ 5000 lignes de code

ANNEXE 4

Exemples de sessions avec PELAGIE

NB : Dans les extraits de sessions suivants, nous avons entouré les réponses de l'usager à PELAGIE. Le reste est du texte généré par PELAGIE ; nous avons souligné les principales requêtes de l'usager au cours de la session.

Essai des différentes fonctions documentaires

(Pelagie)
Bonjour! Je m'appelle Pelagie, et je suis une toute petite maquette d'aide a la conception de logiciels scientifiques.
Au travail et bon courage !

Attention, je ne suis competente qu'en algebre_lineaire.
C'est bien ce qui vous interesse ? (voui)

****** Que desirez-vous :
-1- modifier une variable globale de Pelagie
0 - connaitre les domaines dans lesquels je peux vous aider
1 - avoir une definition d'un concept mathematique
2 - avoir des informations sur des outils mathematiques (domaine, probleme, type_methode, methode, algorithme)
3 - resoudre un probleme
4 - avoir acces aux donnees sur les variables du dernier probleme que vous avez traite depuis que Pelagie est chargee
5 - utiliser un module FORTRAN
6 - avoir des informations sur des outils de controle ou d'amelioration de la solution d'un probleme
7 - aller vous reposer ? (0)

Veuillez patienter, je charge mes connaissances de type : domaine ...
Je possede des connaissances en :
algebre_lineaire -

*** Que desirez-vous faire ensuite ? (si vous ne savez pas, tapez: ?) (1)

Patientez, je charge mes connaissances en algebre_lineaire ...
Je peux vous expliquer les termes suivants:
controle_qualite - preconditionnement - vecteur_propre - valeur_propre - unitaire - taux_de_remplissage_de_matrice - mode_de_stockage_plein - mode_de_stockage_symetrique - mode_de_stockage_bande_symetrique - mode_de_stockage_bande - semblable - conditionnement_inconnu - mal_conditionnee - moyennement_conditionnee - bien_conditionnee - creuse - pleine - structuree - diagonale_dominante - transposee - rayon_spectral - triangulaire - complexe - reelle - tridiagonale - anti_symetrique - symetrique - definie_positive - ordre - diagonale - determinant - deflation - conditionnement - colonne - conjonction - congruence - conjugue - cofacteur - base - bande - adjointe - addition -

*** Sur quoi voulez_vous que je vous renseigne ? mode_de_stockage_bande

Une matrice bande de dimensions (n,n) avec k sous-diagonales et j sur-diagonales stockee en mode bande se reduit a une matrice de dimensions n * (k+j+1).

ex.
a11	a12	a13	0	0		0	a11	a12	a13
a21	a22	a23	a24	0		a21	a22	a23	a24
0	a32	a33	a34	a35	donnera	a32	a33	a34	a35
0	0	a43	a44	a45		a43	a44	a45	0
0	0	0	a54	a55		a54	a55	0	0

*** Que desirez-vous faire ensuite ? (si vous ne savez pas, tapez: ?) ?

****** Que desirez-vous :
-1- modifier une variable globale de Pelagie
0 - connaitre les domaines dans lesquels je peux vous aider
1 - avoir une definition d'un concept mathematique
2 - avoir des informations sur des outils mathematiques (domaine, probleme, type_methode, methode, algorithme)
3 - resoudre un probleme
4 - avoir acces aux donnees sur les variables du dernier probleme que vous avez traite depuis que Pelagie est chargee
5 - utiliser un module FORTRAN
6 - avoir des informations sur des outils de controle ou d'amelioration de la solution d'un probleme
7 - aller vous reposer ? 2

*** Sur quoi voulez_vous etre informe : D(omaine mathematique), PB(probleme), TM(type de methode), M(ethode), A(lgorithme), P(module) ? PB

Veuillez patienter, je charge mes connaissances de type : probleme ...

Nom de probleme - en abrege svp - (rep. ? si vous ne savez pas) ? ?

Voici ce que je connais comme probleme :
Decomposition d'une matrice carree -abrege: decomp
Calcul du determinant d'une matrice carree -abrege: caldet
Inversion d'une matrice carree -abrege: invmat
Resolution d'un systeme lineaire carre avec plusieurs vecteurs seconds membres -abrege: resolsl2
Resolution d'un systeme lineaire carre -abrege: resolsl

Nom de probleme - en abrege svp - (rep. ? si vous ne savez pas) ?
resolsl

*** Desirez-vous connaitre :
1- le domaine traitant de ce probleme
2- la definition du probleme
3- les contraintes que doivent verifier les variables
4- des cas_tests associes a ce probleme
5- des conseils de controle associes a ce probleme
6- les types de methodes permettant de resoudre ce probleme
7- rien d'autre sur ce probleme ? 2

Nom du probleme : Resolution d'un systeme lineaire carre
Le probleme consiste a resoudre un systeme lineaire AX=B ,ou A est une matrice carree, B un vecteur colonne et X la solution du systeme. Probleme en relation directe avec celui d'inversion de matrice.

*** Que desirez-vous faire ensuite ? (si vous ne savez pas, tapez: ?) 3

Le determinant de la matrice du systeme est non nul.

*** Que desirez-vous faire ensuite ? (si vous ne savez pas,

tapez: ?) 6

Patientez, je rassemble mes connaissances de type : type_methode ...
semi_iterative - iterative_par_blocs - iterative_par_points -
composite - directe -

*** Que desirez-vous faire ensuite ? (si vous ne savez pas,
tapez: ?) 5

Un controle de la solution obtenue peut s'averer necessaire dans
certains cas
(cf John.R.Rice Numerical methodes software and analysis- IMSL 83
Gene.H.GOLUB. Charles.F Van LOAN Matrix computation)

*** Que desirez-vous faire ensuite ? (si vous ne savez pas,
tapez: ?) ?

*** Que desirez-vous faire ensuite ? (si vous ne savez pas,
tapez: ?) ?

Nous allons essayer de résoudre simultanément deux problèmes ne portant pas sur les mêmes variables (l'utilisateur s'est peut-être trompé !) : résolution d'un système linéaire et calcul de déterminant.

```
****** Que desirez-vous :
-1- modifier une variable globale de Pelagie
0 - connaitre les domaines dans lesquels je peux vous aider
1 - avoir une definition d'un concept mathematique
2 - avoir des informations sur des outils mathematiques (domaine,
    probleme, type_methode,methode,algorithme)
3 - resoudre un probleme
4 - avoir acces aux donnees sur les variables du dernier probleme
    que vous avez traite depuis que Pelagie est chargee
5 - utiliser un module FORTRAN
6 - avoir des informations sur des outils de controle ou d'ameliо-
    ration de la solution d'un probleme
7 - aller vous reposer                    ? (3)

*** Desirez-vous
1 - verifier l 'adequation d'un outil particulier a un probleme
2 - trouver les meilleurs outils adaptes a la resolution d 'un pro-
    bleme  ? (1)

*** Probleme a resoudre? (nom abrege svp)
(Si vous ne savez pas:?(explications),i(nfo sur un pb),a(ucun))
     resolsl
resolsl :ok? (o)

*** Voulez_vous resoudre simultanement un autre probleme? caldet
Vous avez mal repondu.
*** Voulez_vous resoudre simultanement un autre probleme? (oui)

*** Probleme a resoudre? (nom abrege svp)
(Si vous ne savez pas:?(explications),i(nfo sur un pb),a(ucun))
     caldet
caldet :ok? (o)

*** Voulez_vous resoudre simultanement un autre probleme? (non)

*** Vous allez maintenant nommer vos variables .

***          Pour le probleme Resolution d'un systeme lineaire carre
* Nom de la matrice du systeme? (A)
* Nom de la solution du systeme? (X)
* Nom de l'ordre de la matrice du systeme? (N)
* Nom de le second membre du systeme? (B)

***          Pour le probleme
Calcul du determinant d'une matrice carree
* Nom de la matrice donnee? (A)
* Nom de l'ordre de la matrice? (H)
* Nom de le determinant de la matrice? (K)
Je ne peux pas resoudre ces problemes simultanement puisque ils ne
portent pas sur les memes variables.
nil
```

Nous allons maintenant tenter de résoudre le problème : résolution de système linéaire avec un seul second membre, avec un module d'abord précis, puis occupant le moins de place mémoire possible. La matrice du système sera symétrique, mal conditionnée et d'ordre 600.

```
****** Que desirez-vous :
-1- modifier une variable globale de Pelagie
0 - connaitre les domaines dans lesquels je peux vous aider
1 - avoir une definition d'un concept mathematique
2 - avoir des informations sur des outils mathematiques (domaine,
    probleme,type_methode,methode,algorithme)
3 - resoudre un probleme
4 - avoir acces aux donnees sur les variables du cernier probleme
    que vous avez traite depuis que Pelagie est chargee
5 - utiliser un module FORTRAN
6 - avoir des informations sur des outils de controle ou d'
    amelioration de la solution d'un probleme
7 - aller vous reposer                    ?(3)

*** Desirez-vous
1 - verifier l 'adequation d'un outil particulier a un probleme
2 - trouver les meilleurs outils adaptes a la resolution d'un
    probleme?                (2)   ; définition grossière du problème

*** Probleme a resoudre? (nom abrege svp)
(Si vous ne savez pas:?(explications),i(nfo sur un pb),a(ucun))
     resolsl
resolsl :ok? (o)

*** Voulez_vous resoudre simultanement un autre probleme?(n)

*** Desirez_vous nommer les variables du probleme ?(o)

***          Pour le probleme Resolution d'un systeme lineaire carre
* Nom de la matrice du systeme?(A)
* Nom de la solution du systeme?(X)
* Nom de l'ordre de la matrice du systeme?(N)
* Nom de le second membre du systeme?(B)

*** Donnez-moi vos criteres un par un, classes par ordre de priorite
    decroissante  (rep. possibles: precision , placemem -(oire),
    tempsexec -(ution)    (si pas (ou plus) de critere, rep. nil))    :
* Critere :? precision
* Critere :? placemem
* Critere :? nil

* Vos criteres sont donc (precision placemem) :ok ?(o)

*** Quelle sorte d'outil cherchez-vous :TM(type_methode), M(methode),
    A(algorithme) ou P(module Fortran) ?(P)
 P :ok ?(o)

*** Desirez_vous me guider dans ma recherche ?(n)    ; résolution automatique
```

; Acquisition de données supplémentaires sur le problème

* A est_elle une matrice tridiagonale?
(rep:o-n-p(ourquoi?)-e(xplic.)-v(oir une regle))(e)
Une matrice tridiagonale est une matrice bande telle que Aij=0 si
|i-j|>1.

* A est_elle une matrice tridiagonale?
(rep:o-n-p(ourquoi?)-e(xplic.)-v(oir une regle))(n)

* A est_elle une matrice bande?
(rep:o-n-p(ourquoi?)-e(xplic.)-v(oir une regle))(n)

* A est_elle une matrice definie_positive?
(rep:o-n-p(ourquoi?)-e(xplic.)-v(oir une regle))(n)

* A est_elle une matrice symetrique?
(rep:o-n-p(ourquoi?)-e(xplic.)-v(oir une regle))(o)

* Combien vaut N (meme approximativement) ?
(rep. possibles: ?(si vous l'ignorez) ou e(xplic.) ou
p(ourquoi) ou v(oir une regle))(600)
N =600 :ok ?(o)

* A est_elle bien conditionnee?
(rep:o-n-p(ourquoi?)-e(xplic.)-v(oir une regle))(n)

* A est_elle moyennement conditionnee?
(rep:o-n-p(ourquoi?)-e(xplic.)-v(oir une regle))(n)

* A est_elle mal conditionnee?
(rep:o-n-p(ourquoi?)-e(xplic.)-v(oir une regle))(o)

* A est_elle une matrice creuse?
(rep:o-n-p(ourquoi?)-e(xplic.)-v(oir une regle))(e)
Une matrice A est dite creuse si elle a beaucoup de
coefficients nuls.

* A est_elle une matrice creuse?
(rep:o-n-p(ourquoi?)-e(xplic.)-v(oir une regle))(n)

* A est_elle une matrice a diagonale dominante?
(rep:o-n-p(ourquoi?)-e(xplic.)-v(oir une regle))(e)
Une matrice carree A est a diagonale dominante si les
coefficients de sa diagonale principale sont grands en valeur
absolue par rapport aux autres coefficients.

* A est_elle une matrice a diagonale dominante?
(rep:o-n-p(ourquoi?)-e(xplic.)-v(oir une regle))(n)

* Combien vaut le rayon spectral de A (meme approximativement) ?
(rep. possibles: ?(si vous l'ignorez) ou e(xplic.) ou
p(ourquoi) ou v(oir une regle))(?)

; PELAGIE, "après réflexion", propose une solution

(RECAPITULONS: voici donc ce que je vous propose comme module
pour resoudre le probleme [resolsl] :
LEQ2S avec environ 0.2916 pour coefficient de validite)

 ; PELAGIE propose aussi des "outils de contrôle"

Je vous conseille aussi avant d'appliquer LEQ2S :
de preconditionner la matrice par exemple avec methode :
mise_a_echelle_par_ligne_ou_colonne
de preconditionner la matrice par exemple avec methode :
normalisation
de preconditionner la matrice par exemple avec methode :
scaling_optimal
de preconditionner la matrice par exemple avec methode :
preconditionnement_geometrique

de preconditionner la matrice par exemple avec methode :
preconditionnement_additif

Je vous conseille aussi apres avoir applique LEQ2S :
de controler la qualite numerique de la solution par exemple
avec conseil : - John.R. Rice, Numerical methods software and
analysis IMSL 83
- Gene.H.Golub.Charles.F Van Loan, Matrix computations
de controler la qualite numerique de la solution par exemple
avec methode : adjonction_par_ligne_ou_colonne
de controler la qualite numerique de la solution par exemple
avec methode : perturbation_permutation_de_Vignes_Laporte
de controler la qualite numerique de la solution par exemple
avec methode : gradient_conjugue
de controler la qualite numerique de la solution par exemple
avec methode : relaxation
de controler la qualite numerique de la solution par exemple
avec methode : calcul_en_double_precision
de controler la qualite numerique de la solution par exemple
avec methode : estimation_du_nombre_de_condition_de_la_matrice
si vous choisissez d'utiliser LEQ2S

*** Voulez-vous que je vous explique comment j'aboutis a cette conclusion ? (o)

(SELECTION MATHEMATIQUE de type: type_methode pour resoudre [resolsl]:

A est symetrique et non definie_positive et non [bande avec
son taux de remplissage <5] et N est compris entre 500 et 2000
et A est mal conditionnee donc [iterative_par_blocs] avec 40 %
de chances de reussite < regle 152 >
A est symetrique et non definie_positive et non [bande avec
son taux de remplissage <5] et N est compris entre 500 et 2000
et A est mal conditionnee donc [iterative_par_points] avec 40 %
de chances de reussite < regle 151 >
A est symetrique et non definie_positive et non [bande avec
son taux de remplissage <5] et N est compris entre 500 et 2000 et
A est mal conditionnee donc [composite] avec 40 % de chances de
reussite < regle 149 >
A est symetrique et non definie_positive et non [bande avec son
taux de remplissage <5] et N est compris entre 500 et 2000 et
A est mal conditionnee donc [directe] avec 40 % de chances de
reussite < regle 150 >

PRISE EN COMPTE DES CRITERES: precision placemem

iterative_par_points mieux que iterative_par_blocs au point de vue precision <regle 245 >
iterative_par_points mieux que directe au point de vue precision < regle 233 >
composite mieux que directe au point de vue precision <regle 239>
donc selon le critere precision je ne retiendrai que :
iterative_par_points
composite

le critere placemem est contradictoire car
iterative_par_points mieux que iterative_par_blocs au point de vue place memoire < regle 244 >
directe mieux que iterative_par_points au point de vue place memoire < regle 243 >
directe mieux que composite au point de vue place memoire <regle 242>

donc je lance la 2de strategie a partir de [composite iterative_par_points]
Pas de meilleur selon le critere placemem

donc a partir de [iterative_par_blocs iterative_par_points composite directe] :
la selection de type: type_methode sur les CRITERES precision placemem ne retient que :
iterative_par_points avec 40 % de chances de reussite
composite avec 40 % de chances de reussite

SELECTION MATHEMATIQUE de type: methode a partir de iterative_par_points :

aucune autre methode iterative_par_points convergente donc [gradient_conjugue] avec 80 % de chances de reussite <regle 253>

Une seule solution donc je ne tiens pas compte des criteres Usager

Desolee mais je ne connais pas de algorithme descendant de gradient_conjugue

Donc finalement aucun succes a partir de type_methode iterative_par_points !

SELECTION MATHEMATIQUE de type: methode a partir de composite :

A est symetrique et A est non definie_positive et non [bande avec son taux de remplissage < 5] et N est inferieur a 2000 donc [decomposition_symetrique_bis] avec 90 % de chances de reussite < regle 249 >

Une seule solution donc je ne tiens pas compte des criteres Usager

SELECTION MATHEMATIQUE de type: algorithme a partir de decomposition_symetrique_bis
:

A est symetrique et A est non definie_positive et non [bande avec son taux de remplissage < 5] donc [LDLt_bis] avec 90 % de chances de reussite < regle 359 >

Une seule solution donc je ne tiens pas compte des criteres Usager

SELECTION MATHEMATIQUE de type: module a partir de LDLt_bis :

A est symetrique donc [LEQ2S] avec 90 % de chances de reussite
< regle 321 >

Une seule solution donc je ne tiens pas compte des criteres Usager

donc SOLUTION: LEQ2S

RECAPITULONS: voici donc ce que je vous propose comme module pour resoudre le probleme [resolsl] :
LEQ2S avec environ 0.2916 pour coefficient de validite
)

; la résolution du problème précédent est terminée

*** Que desirez-vous faire ensuite?(si vous ne savez pas, tapez:?) (?)

****** Que desirez-vous :
-1- modifier une variable globale de Pelagie
0 - connaitre les domaines dans lesquels je peux vous aider
1 - avoir une definition d'un concept mathematique
2 - avoir des informations sur des outils mathematiques (domaine, probleme,type_methode,methode,algorithme)
3 - resoudre un probleme
4 - avoir acces aux donnees sur les variables du dernier probleme que vous avez traite depuis que Pelagie est chargee
5 - utiliser un module FORTRAN
6 - avoir des informations sur des outils de controle ou d'amelioration de la solution d'un probleme
7 - aller vous reposer ? (4)

*** Veuillez taper svp:
1- Pour connaitre tous les faits que vous m'avez affirmes
2- Pour connaitre les faits que vous m'avez affirmes sur une variable particuliere
3- Pour annuler tous les faits que vous m'avez affirmes
4- Pour annuler les connaissances que j'ai sur une variable particuliere
(Attention, si vous voulez maintenant resoudre un nouveau probleme avec des variables de nom identique au probleme precedent, mais ayant des contraintes differentes, vous devez passer par les options 3 ou 4)

```
     Vous vouliez resoudre le probleme :
     (resolsl sur les variables : A X N B )
     Criteres vous interessant : (precision placemem)

     A est symetrique
     A est mal conditionnee
     N vaut 600
     A n'est pas definie_positive
     A n'est pas tridiagonale
     A n'est pas bande
     A n'est pas creuse
     A n'est pas bien conditionnee
     A n'est pas moyennement conditionnee
     La valeur de (le rayon spectral de A) est inconnue
```

; Utilisation de fonctions documentaires, relativement à la solution proposée

```
****** Que desirez-vous :
-1- modifier une variable globale de Pelagie
0 - connaitre les domaines dans lesquels je peux vous aider
1 - avoir une definition d'un concept mathematique
2 - avoir des informations sur des outils mathematiques (domaine,
    probleme, type_methode,methode,algorithme)
3 - resoudre un probleme
4 - avoir acces aux donnees sur les variables du dernier probleme
    que vous avez traite depuis que Pelagie est chargee
5 - utiliser un module FORTRAN
6 - avoir des informations sur des outils de controle ou d'amelio-
    ration de la solution d'un probleme
7 - aller vous reposer                    ? (5)

 Veuillez patienter, je charge mes connaissances de type : module ...

Nom de module  (reo. ? si vous ne savez pas) ? (LEQ2S)

*** Desirez-vous connaitre :
1- la nature de LEQ2S
2- le role de LEQ2S
3- le mode d'appel de LEQ2S
4- la description d'une ou des variables d'appel du module LEQ2S
5- la description d'une ou des variables globales du module LEQ2S
6- les contraintes associees aux variables du module LEQ2S
7- les proprietes associees aux variables du module LEQ2S
8- les modules appeles par LEQ2S

9- la fonction du module LEQ2S decrite de facon formelle
10- les caracteristiques machine (CDC) de LEQ2S
11- les caracteristiques de LEQ2S vis a vis des criteres usager
12- des references sur de la documentation sur LEQ2S
13- un cas test pour LEQ2S
14- methode,algorithme...mis en oeuvre dans LEQ2S
15- rien d'autre sur ce module          ? (1)
```

subroutine

```
*** Que desirez-vous faire ensuite ? (si vous ne savez pas, tapez :?) 2

Resolution de systemes lineaires avec matrice symetrique. Donne une
solution precise (sous reserve du bon conditionnement de la matrice).


*** Que desirez-vous faire ensuite ? (si vous ne savez pas, tapez :?) 3

CALL LEQ2S (A,N,B,M,IB,IJOB,ICHNG,DET,IER)
INTEGER ICHNG(1)
REAL A(1),DET(1),B(IB,M),ZERO,XNORM,DXNORM
DOUBLE PRECISION SUM
DATA ZERO/0.0/,IMAX/50/


*** Que desirez-vous faire ensuite ? (si vous ne savez pas, tapez :?) 4

De toutes les variables (tapez 1) ou de certaines seulement (tapez 2) ? 1
variable : A
tableau N*(N+1)/2 reel
en entree , matrice symetrique du systeme

variable : N
scalaire entier
en entree , nombre de lignes de B et ordre de A

variable : B
tableau N*M reel
en entree , second(s) membre(s) de la solution
en sortie , solution (si IJOB=0 ou 2)
Si B=I le pb se ramene a celui de l'inversion de A pour IJOB=0 ou 2

variable : M
scalaire entier
en entree , nombre de second(s) membre(s)

variable : IB
scalaire entier
en entree , nombre de lignes de B

variable : IJOB
scalaire entier
en entree , parametre pour option

variable : ICHNG
tableau 2*N reel
en sortie , zone de travail

variable : DET

tableau N*(N+1)/(2+3*N) reel
en sortie , zone de travail
en sortie , mise en facteurs de A (si IJOB=0 ou 1)
les N*(N+1)/2 premieres positions de DET contiennent la mise en
facteurs de A
```

variable : IER
scalaire entier
en sortie , indice erreur
IER=129 signifie que A est numeriquement singuliere
IER=130 signifie que A est trop mal conditionnee d'ou echec de la procedure iterative d'amelioration de la qualite numerique de la solution

*** Que desirez-vous faire ensuite ? (si vous ne savez pas, tapez :?) 6

A stockee en mode symetrique
IB>=N
IJOB=0,1,ou 2

*** Que desirez-vous faire ensuite ? (si vous ne savez pas, tapez :?) 11

LEQ2S plus precis, plus encombrant, et plus long que LEQ1S

*** Que desirez-vous faire ensuite ? (si vous ne savez pas, tapez :?) ?

*** Desirez-vous connaitre :
1- la nature de LEQ2S
2- le role de LEQ2S
3- le mode d'appel de LEQ2S
4- la description d'une ou des variables d'appel du module LEQ2S
5- la description d'une ou des variables globales du module LEQ2S
6- les contraintes associees aux variables du module LEQ2S
7- les proprietes associees aux variables du module LEQ2S
8- les modules appeles par LEQ2S
9- la fonction du module LEQ2S decrite de facon formelle
10- les caracteristiques machine (CDC) de LEQ2S
11- les caracteristiques de LEQ2S vis a vis des criteres usager
12- des references sur de la documentation sur LEQ2S
13- un cas test pour LEQ2S
14- methode,algorithme...mis en oeuvre dans LEQ2S
15- rien d'autre sur ce module ? 12

1. BUNCH,J.R., and PARLETT, B.N., Direct methods for solving symmetric indefinite systems of linear equations
SIAM.J Numer.Anal.,8(1971),639-655
2. BUNCH,J.R., Analysis of the diagonal pivoting methods
SIAM J. Numer.Anal., 8(1971), 656-680

*** Que desirez-vous faire ensuite ? (si vous ne savez pas, tapez: ?) 15

```
****** Que desirez-vous :
-1- modifier une variable globale de Pelagie
0 - connaitre les domaines dans lesquels je peux vous aider
1 - avoir une definition d'un concept mathematique
2 - avoir des informations sur des outils mathematiques (domaine,
    probleme, type_methode,methode,algorithme)
3 - resoudre un probleme
4 - avoir acces aux donnees sur les variables du dernier probleme
    que vous avez traite depuis que Pelagie est chargee
5 - utiliser un module FORTRAN
6 - avoir des informations sur des outils de controle ou d'amelio-
    ration de la solution d'un probleme
7 - aller vous reposer                    ? 6

Je connais plusieurs categories d'outils:
1 - de preconditionnement
2 - de controle de qualite numerique de solution
    Donnez-moi le numero de la categorie qui vous interesse : 1
Nom de la methode (si vous ne savez pas, repondez: ?) ? ?
Voici ce que je connais :
mise_a_echelle_par_ligne_ou_colonne - normalisation -
scaling_optimal - preconditionnement_geometrique -
preconditionnement_multiplicatif -

Nom de la methode (si vous ne savez pas, repondez: ?) ? scaling_optimal
conditions d'utilisation : non restreintes
references :
GOLUB et Vander SMUIS   Numerische Mathematik 69/70
M.V. GOLITSCHEK  An algorithm for scaling Numerische Mathematik 80
commentaires :
Elle consiste a choisir simultanement des normalisations par ligne
et par colonne; il s'agit la plutot d'un probleme d'optimisation,
ce qui en limite l'utilisation.

Etes-vous interesse par une autre methode ayant le meme but ? o
Nom de la methode (si vous ne savez pas, repondez: ?) ? ?
Voici ce que je connais :
mise_a_echelle_par_ligne_ou_colonne - normalisation -
scaling_optimal - preconditionnement_geometrique -
preconditionnement_multiplicatif -

Nom de la methode (si vous ne savez pas, repondez: ?) ? normalisation
conditions d'utilisation : non restreintes
commentaires :
Moyen frequemment utilise.

Etes-vous interesse par une autre methode ayant le meme but ? non
```