Un opérateur d'unification pour une machine base de connaissance PROLOG

Jean-Christophe Ianeselli

THESE

présentée à

l' Institut National Polytechnique de Grenoble

pour obtenir le grade de
DOCTEUR DE 3ème CYCLE
«Informatique»

par

Jean-Christophe IANESELLI

UN OPERATEUR D'UNIFICATION POUR UNE MACHINE BASE DE CONNAISSANCE PROLOG.

Thèse soutenue le 3 juin 1985 devant la commission d'examen.

P. JORRAND	Président
F. ANCEAU	
G. VEILLON	
G. BERGER SABBATEL	Examinateurs
L. DEMIANS D'ARCHIMBAUD	
C. ROCHE	

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

Année universitaire 1982-1983

Président de l'Université : D. BLOCH

Vice-Président : René CARRE
Hervé CHERADAME
Marcel IVANES

PROFESSEURS DES UNIVERSITES :

ANCEAU François	E.N.S.I.M.A.G.
BARRAUD Alain	E.N.S.I.E.G.
BAUDELET Bernard	E.N.S.I.E.G.
BESSON Jean	E.N.S.E.E.G.
BLIMAN Samuel	E.N.S.E.R.G.
BLOCH Daniel	E.N.S.I.E.G.
BOIS Philippe	E.N.S.H.G.
BONNETAIN Lucien	E.N.S.E.E.G.
BONNIER Etienne	E.N.S.E.E.G.
BOUVARD Maurice	E.N.S.H.G.
BRISSONNEAU Pierre	E.N.S.I.E.G.
BUYLE BODIN Maurice	E.N.S.E.R.G.
CAVAIGNAC Jean-François	E.N.S.I.E.G.
CHARTIER Germain	E.N.S.I.E.G.
CHENEVIER Pierre	E.N.S.E.R.G.
CHERADAME Hervé	U.E.R.M.C.P.P.
CHERUY Arlette	E.N.S.I.E.G.
CHIAVERINA Jean	U.E.R.M.C.P.P.
COHEN Joseph	E.N.S.E.R.G.
COUMES André	E.N.S.E.R.G.
DURAND Francis	E.N.S.E.E.G.
DURAND Jean-Louis	E.N.S.I.E.G.
FELICI Noël	E.N.S.I.E.G.
FOULARD Claude	E.N.S.I.E.G.
GENTIL Pierre	E.N.S.E.R.G.
GUERIN Bernard	E.N.S.E.R.G.
GUYOT Pierre	E.N.S.E.E.G.
IVANES Marcel	E.N.S.I.E.G.
JAUSSAUD Pierre	E.N.S.I.E.G.
JOUBERT Jean-Claude	E.N.S.I.E.G.
JOURDAIN Geneviève	E.N.S.I.E.G.
LACOUME Jean-Louis	E.N.S.I.E.G.
LATOMBE Jean-Claude	E.N.S.I.M.A.G.

.../...

LESSIEUR Marcel	E.N.S.H.G.
LESPINARD Georges	E.N.S.H.G.
LONGEQUEUE Jean-Pierre	E.N.S.I.E.G.
MAZARE Guy	E.N.S.I.M.A.G.
MOREAU René	E.N.S.H.G.
MORET Roger	E.N.S.I.E.G.
MOSSIERE Jacques	E.N.S.I.M.A.G.
PARIAUD Jean-Charles	E.N.S.E.E.G.
PAUTHENET René	E.N.S.I.E.G.
PERRET René	E.N.S.I.E.G.
PERRET Robert	E.N.S.I.E.G.
PIAU Jean-Michel	E.N.S.H.G.
POLOUJADOFF Michel	E.N.S.I.E.G.
POUPOT Christian	E.N.S.E.R.G.
RAMEAU Jean-Jacques	E.N.S.E.E.G.
RENAUD Maurice	U.E.R.M.C.P.P.
ROBERT André	U.E.R.M.C.P.P.
ROBERT François	E.N.S.I.M.A.G.
SABONNADIERE Jean-Claude	E.N.S.I.E.G.
SAUCIER Gabrielle	E.N.S.I.M.A.G.
SCHLENKER Claire	E.N.S.I.E.G.
SCHLENKER Michel	E.N.S.I.E.G.
SERMET Pierre	E.N.S.E.R.G.
SILVY Jacques	U.E.R.M.C.P.P.
SOHM Jean-Claude	E.N.S.E.E.G.
SOUQUET Jean-Louis	E.N.S.E.E.G.
VEILLON Gérard	E.N.S.I.M.A.G.
ZADWORNY François	E.N.S.E.R.G.

PROFESSEURS ASSOCIES

BASTIN Georges	E.N.S.H.G.
BERRIL John	E.N.S.H.G.
CARREAU Pierre	E.N.S.H.G.
GANDINI Alessandro	U.E.R.M.C.P.P.
HAYASHI Hirashi	E.N.S.I.E.G.

PROFESSEURS UNIVERSITE DES SCIENCES SOCIALES (Grenoble II)

BOLLIET Louis
Chatelin Françoise

PROFESSEURS E.N.S. Mines de Saint-Etienne

RIEU Jean
SOUSTELLE Michel

CHERCHEURS DU C.N.R.S.

FRUCHART Robert	Directeur de Recherche
VACHAUD Georges	Directeur de Recherche

.../...

ALLIBERT Michel	Maître de Recherche
ANSARA Ibrahim	Maître de Recherche
ARMAND Michel	Maître de Recherche
BINDER Gilbert	
CARRE René	Maître de Recherche
DAVID René	Maître de Recherche
DEPORTES Jacques	
DRIOLE Jean	Maître de Recherche
GIGNOUX Damien	
GIVORD Dominique	
GUELIN Pierre	
HOPFINGER Emil	Maître de Recherche
JOUD Jean-Charles	Maître de Recherche
KAMARINOS Georges	Maître de Recherche
KLEITZ Michel	Maître de Recherche
LANDAU Ioan-Dore	Maître de Recherche
LASJAUNIAS J.C.	
MERMET Jean	Maître de Recherche
MUNIER Jacques	Maître de Recherche
PIAU Monique	
PORTESEIL Jean-Louis	
THOLENCE Jean-Louis	
VERDILLON André	

CHERCHEURS du MINISTERE de la RECHERCHE et de la TECHNOLOGIRE (Directeurs et Maîtres de Recherches, ENS Mines de St. Etienne)

LESBATS Pierre	Directeur de Recherche
BISCONDI Michel	Maître de Recherche
KOBYLANSKI André	Maître de Recherche
LE COZE Jean	Maître de Recherche
LALAUZE René	Maître de Recherche
LANCELOT Francis	Maître de Recherche
THEVENOT François	Maître de Recherche
TRAN MINH Canh	Maître de Recherche

PERSONNALITES HABILITEES à DIRIGER des TRAVAUX de RECHERCHE (Décision du Conseil Scientifique)

ALLIBERT Colette	E.N.S.E.E.G.
BERNARD Claude	E.N.S.E.E.G.
BONNET Rolland	E.N.S.E.E.G.
CAILLET Marcel	E.N.S.E.E.G.
CHATILLON Catherine	E.N.S.E.E.G.
CHATILLON Christian	E.N.S.E.E.G.
COULON Michel	E.N.S.E.E.G.
DIARD Jean-Paul	E.N.S.E.E.G.
EUSTAPOPOULOS Nicolas	E.N.S.E.E.G.
FOSTER Panayotis	E.N.S.E.E.G.

.../...

GALERIE Alain	E.N.S.E.E.G.
HAMMOU Abdelkader	E.N.S.E.E.G.
MALMEJAC Yves	E.N.S.E.E.G. (CENG)
MARTIN GARIN Régina	E.N.S.E.E.G.
NGUYEN TRUONG Bernadette	E.N.S.E.E.G.
RAVAINE Denis	E.N.S.E.E.G.
SAINFORT	E.N.S.E.E.G. (CENG)
SARRAZIN Pierre	E.N.S.E.E.G.
SIMON Jean-Paul	E.N.S.E.E.G.
TOUZAIN Philippe	E.N.S.E.E.G.
URBAIN Georges	E.N.S.E.E.G. (Laboratoire des ultra-réfractaires ODEILLON)
GUILHOT Bernard	E.N.S. Mines Saint Etienne
THOMAS Gérard	E.N.S. Mines Saint Etienne
DRIVER Julien	E.N.S. Mines Saint Etienne
BARIBAUD Michel	E.N.S.E.R.G.
BOREL Joseph	E.N.S.E.R.G.
CHOVET Alain	E.N.S.E.R.G.
CHEHIKIAN Alain	E.N.S.E.R.G.
DOLMAZON Jean-Marc	E.N.S.E.R.G.
HERAULT Jeanny	E.N.S.E.R.G.
MONLLOR Christian	E.N.S.E.R.G.
BORNARD Guy	E.N.S.I.E.G.
DESCHIZEAU Pierre	E.N.S.I.E.G.
GLANGEAUD François	E.N.S.I.E.G.
KOFMAN Walter	E.N.S.I.E.G.
LEJEUNE Gérard	E.N.S.I.E.G.
MAZUER Jean	E.N.S.I.E.G.
PERARD Jacques	E.N.S.I.E.G.
REINISCH Raymond	E.N.S.I.E.G.
ALEMANY Antoine	E.N.S.H.G.
BOIS Daniel	E.N.S.H.G.
DARVE Félix	E.N.S.H.G.
MICHEL Jean-Marie	E.N.S.H.G.
OBLED Charles	E.N.S.H.G.
ROWE Alain	E.N.S.H.G.
VAUCLIN Michel	E.N.S.H.G.
WACK Bernard	E.N.S.H.G.
BERT Didier	E.N.S.I.M.A.G.
CALMET Jacques	E.N.S.I.M.A.G.
COURTIN Jacques	E.N.S.I.M.A.G.
COURTOIS Bernard	E.N.S.I.M.A.G.
DELLA DORA Jean	E.N.S.I.M.A.G.
FONLUPT Jean	E.N.S.I.M.A.G.
SIFAKIS Joseph	E.N.S.I.M.A.G.
CHARUEL Robert	U.E.R.M.C.P.P.
CADET Jean	C.E.N.G.
COEURE Philippe	C.E.N.G. (LETI)

.../...

DELHAYE Jean-Marc	C.E.N.G. (STT)
DUPUY Michel	C.E.N.G. (LETI)
JOUVE Hubert	C.E.N.G. (LETI)
NICOLAU Yvan	C.E.N.G. (LETI)
NIFENECKER Hervé	C.E.N.G.
PERROUD Paul	C.E.N.G.
PEUZIN Jean-Claude	C.E.N.G. (LETI)
TAIEB Maurice	C.E.N.G.
VINCENDON Marc	C.E.N.G.

LABORATOIRES EXTERIEURS

DEMOULIN Eric	C.N.E.T.
DEVINE	C.N.E.T. (R.A.B.)
GERBER Roland	C.N.E.T.
MERCKEL Gérard	C.N.E.T.
PAULEAU Yves	C.N.E.T.
GAUBERT C.	I.N.S.A. Lyon

Je tiens à remercier:

Monsieur F. Anceau, Professeur ENSIMAG actuellement en disponibilité à la société BULL, pour m'avoir accueilli dans son équipe, ainsi que J. Della Dora, actuel directeur du laboratoire TIM3 de l'IMAG, et B. Courtois, responsable de l'équipe de recherche en architecture des ordinateurs,

Tout spécialement Gilles Berger-sabbatel, chargé de recherche au CNRS, pour l'aide qu'il m'a fournie tout au long de cette thèse ainsi que pour les nombreuses collaborations qu'il m'a proposées,

Monsieur P. Jorrand, directeur de recherche au CNRS et responsable du laboratoire d'informatique et d'intelligence artificielle de l'IMAG, de m'avoir fait l'honneur de présider le jury de cette thèse,

Monsieur G. Veillon, Professeur ENSIMAG et directeur de cette école, de m'avoir fait l'honneur de participer au jury de cette thèse.

L. Demians d'Archimbaud, chercheur en intelligence artificielle à la société CIMSA, pour sa participation au jury de cette thèse,

C. Roche, chercheur en intelligence artificielle à la société CRIL, non seulement pour sa participation au jury mais également pour toutes les discussions fructueuses que nous avons eues au cours de ces dernières années.

Je ne voudrais pas oublier dans mes remerciements tous les membres de l'équipe de recherche en architecture en ordinateurs qui ont toujours amicalement et avec compétence répondu à mes questions.

Enfin, je remercie D. Igliesas et son équipe du service de reprographie de l'IMAG, pour la qualité de leur travail.

A mes parents,

A Véronique.

TABLE DES MATIERES

TABLE DES MATIERES

PARTIE A : SPECIFICATIONS FONCTIONNELLES

CHAPITRE I : PRESENTATIONS

CHAPITRE II : PROLOG IMPLANTATION ET STRATEGIE DE RECHERCHE

CHAPITRE III : SELECTION DES CONNAISSANCES

PARTIE B : IMPLANTATION LOGICIELLE

CHAPITRE IV : UNIFICATION PARALLELE

CHAPITRE V : IMPLANTATION LOGICIELLE

CHAPITRE VI : COOPERATION DES STRATEGIES

PARTIE C : SPECIFICATIONS MATERIELLES

CHAPITRE VII : ARCHITECTURE ORIENTEE MODULE

CHAPITRE VIII : DESCRIPTION MATERIELLE

CHAPITRE IX : SIMULATION

INTRODUCTION

I N T R O D U C T I O N

En ces premières années 80, on parle de la deuxième révolution informatique, ce qui tend à cacher l'évolution permanente du domaine, qui maintient une révolution perpétuelle. Ce phénomène est la conséquence d'une croissance non pas linéaire mais exponentielle: la puissance des ordinateurs double tous les deux à trois ans. Les ordinateurs des années 70 considérés comme des "monstres" sont aujourd'hui réduits à l'état de "puces"... Les raisons de cette croissance sont essentiellement technologiques, nous voulons parler de l'avénement de la technique d'intégration qui permet aujourd'hui de développer des circuits VLSI intégrant jusqu'à un million de transistors.

Au vu de cette croissance débridée, nous sommes en droit de nous demander ce que nous réserve la prochaine décennie. nous délaisserons les problèmes sociologiques, trop importants pour être traités en quelques lignes pour proposer seulement quelques éléments "scientifiques" de réponse:

- Si les limites physiques bornant l'évolution des VLSI sont en vue, des gains encore notables sont possibles. D'autre part, de nouveaux matériaux seraient susceptibles de repousser de plusieurs ordres de grandeur les limites du silicium (Arseniure de Gallium, pour rester dans le domaine des semi-conducteurs [REM 84], voire utilisation de circuits biologiques !? [REM 85]).

- Dans l'état actuel de la technologie, la maitrise de la complexité des circuits, autre que ceux intégrant uniquement des cellules répétitives telles les mémoires, est loin d'être acquise ce qui freine pour l'heure l'évolution et ce avant les limites physiques.

- Des besoins nouveaux apparaissent, que l'on regroupera autour des applications de l'intelligence artificielle, que les ordinateurs classiques ont de plus en plus de difficultés à satisfaire.

Délaissant le premier point qui relève plutôt de la physique (et peut-être bientôt de la biologie ?), voyons comment les deux suivants peuvent nous fournir quelques indications quant aux développements à venir.

La conception des circuits intégrés, domaine de la micro-électronique, demeure une opération très artisanale en ce sens que la plupart des étapes du processus sont faites à la main. Différents outils, regroupés sous le vocable de CAO, sont pourtant disponibles pour aider le concepteur mais aucun ne permet une conception entièrement automatique. C'est pourquoi les efforts actuels se tournent vers la réalisation de "compilateurs de silicium" [GRO 83] processus assurant, à partir d'une description algorithmique de haut niveau d'un circuit, la génération des masques nécessaires à sa réalisation physique. L'avènement de tels outils permettra d'effectuer un saut tant quantitatif que qualitatif, ce en banalisant la conception des circuits. Néanmoins il semble encore difficile d'estimer l'influence de leur utilisation sur les architectures des machines à venir, ce pour diverses raisons que nous essaierons de développer au chapitre IX.

Si les ordinateurs ont déjà largement dépassé le cadre des centres de calcul, les tâches non-numériques qu'ils effectuent aujourd'hui ne sont que les balbutiements de celles visées par les "machines de cinquième génération". Selon les Japonnais, instigateurs de la dénomination "cinquième génération", ces machines seront capables non seulement de voir, parler, entendre mais qui plus est seront "douées de raison". Si les premiers points sont "extraordinaires" pour un public non averti, c'est plutôt le deuxième qui retient l'attention de la communauté informatique, les bases sous-jacentes de ce raisonnement étant largement utilisées pour mettre en oeuvre les premiers. Désormais les Japonnais n'emploient plus le vocable d'ordinateurs, mais celui de "système informatique de traitement de la connaissance" (KIPS en anglais). Ces systèmes se singularisent par leur aptitude à traiter la connaissance, à raisonner, voire apprendre (pour un aperçu des motivations, des conséquences et enfin des différents travaux sur la cinquième génération, de par le monde, on propose la lecture de l'ouvrage référencé [FEI 82]).

L'idée maîtresse est de substituer au traitement numérique de base un processus d'inférence logique: la résolution. Aujourd'hui on évalue la puissance d'un ordinateur en nombre d'instructions (opérations essentiellement arithmétiques) par seconde (IPS). Pour les ordinateurs de cinquième génération on parle de nombre d'inférences logiques par seconde (LIPS), une inférence logique pouvant être schématisée par une séquence de raisonnement du genre SI/ALORS.

Il existe déjà un langage de programmation basé sur le principe de résolution: PROLOG. De ce fait ses concepts furent repris par les Japonnais comme bases du langage noyau de leurs systèmes [CHI 84]. De part le monde, PROLOG est actuellement reconnu comme une approche intéressante, et nombre de travaux s'efforcent soit d'améliorer ses performances soit de l'étendre.

En accord avec la plupart de ces réflexions, nous pensons que PROLOG est un outil précieux quant à la manipulation des connaissances. Nous sommes également convaincus que seules des machines adaptées supporteront efficacement des applications importantes, une des faiblesses actuelles du langage étant liée à un temps d'exécution rapidement prohibitif dès que l'application croît. Ainsi le projet OPALE [BER 82], cadre du sujet de cette thèse, a pour objectif d'apporter une contribution à l'effort actuellement développé dans ce sens et nous le présenterons plus en détail au chapitre 1.

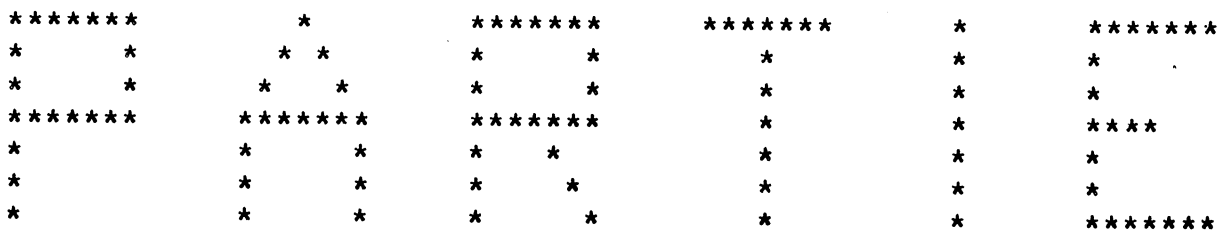
Nous présenterons les objectifs de cette thèse après la description du projet l'englobant (chapitre 1, paragraphe 4). D'un point de vue technique, elle se décompose en trois parties: chacune débute par un résumé étendu des points traités, chaque chapitre commençant par un résumé succint.

La première partie fournit les spécifications fonctionnelles en développant d'une part la présentation du projet et en apportant, d'autre part, les considérations et justifications qui sous-tendent ces spécifications.

Dans la deuxième, on présente tout d'abord les idées des algorithmes mis en oeuvre pour supporter ces spécifications, puis leur

implantation. Cette partie se termine par l'intégration des outils logiciels en une première maquette.

La troisième et dernière partie aborde le côté matériel en fournissant les spécifications, les idées de conception et de réalisation d'une première simulation d'une maquette matérielle.



****
*   *
* SPECIFICATIONS FONCTIONNELLES *
*   *
****

RESUME

Cette première partie a pour trait les spécifications des objectifs à atteindre et précise les premières orientations quant à l'élaboration des mécanismes les satisfaisant. Elle est composée de trois chapitres traitant respectivement:

- Des présentations:
  - du langage PROLOG,
  - des Bases de Données (que l'on généralisera en Bases de Connaissances),
  - du projet OPALE, cadre du sujet.

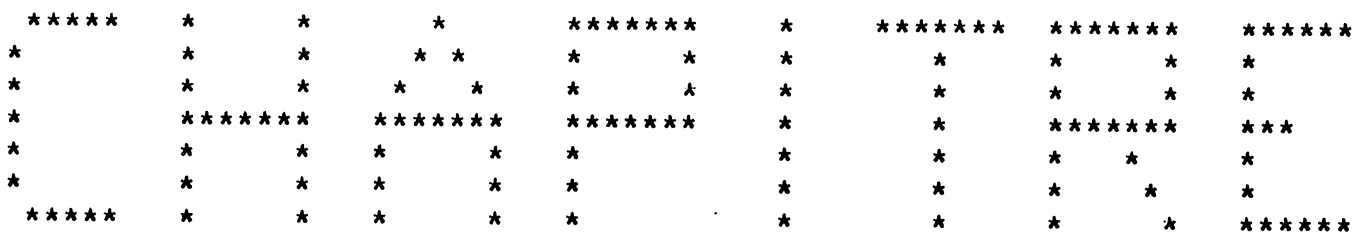
- De l'implantation du langage PROLOG et plus particulièrement de la stratégie de recherche, ainsi que les modifications expérimentées à travers divers langages dérivés.

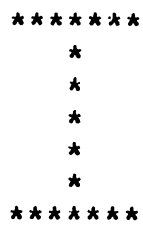
- Du problème de sélection dans les Bases de Connaissances, et des mécanismes sous-jacents, en particulier le filtrage.

Le premier chapitre présente le langage PROLOG, comme langage de programmation logique, et montre brièvement ses possibilités quant à l'expression des connaissances. Nous verrons alors les fondements théoriques du langage qui seront exploités pour la modélisation d'une Base de Connaissances. En effet, analysant l'évolution des Bases de Données "classiques" vers les Bases de Données déductives, nous proposons de réaliser des Bases de Connaissances en s'appuyant sur le langage PROLOG. Nous verrons alors les insuffisances d'une interprétation classique pour réaliser un "Système de Gestion de Bases de Connaissances", ces réflexions ayant abouti à l'élaboration du projet OPALE. Nous en exposerons alors les grandes lignes avant de situer la thèse dans ce contexte, puis de fournir une vue d'ensemble de cette dernière.

Le chapitre deux aborde le coeur du sujet: le langage PROLOG. Nous en détaillons les mécanismes de base tant du point de vue de la programmation, que du point de vue l'implantation, ce en nous appuyant sur ses aspects déclaratifs et procéduraux. Parmi ces mécanismes nous développons plus longuement la stratégie de recherche, reportant au chapitre quatre la description complète de l'algorithme d'unification. Cette stratégie n'est pas toujours satisfaisante, et de nombreux travaux en proposent des modifications, la plupart aboutissant à des langages voisins. Nous en tirerons certains enseignements quant à la stratégie à employer dans notre contexte. Le chapitre se termine par l'exposé de l'interpréteur "classique" que nous avons implanté et dont nous nous servirons tout au long ce cette étude, comme support de développement.

Le dernier chapitre porte sur le point essentiel, pour la réalisation d'un SGBC: la sélection des connaissances. Partant des idées mises en oeuvre dans les Bases de Données, d'une part quant à la restriction du domaine de recherche par la création de chemins d'accès, et d'autre part par l'utilisation de mécanismes de filtrage, nous aboutirons à la définition d'une stratégie de recherche ensembliste méthode également issue des Bases de Données. A la suite de quoi, nous préciserons les spécifications du filtre, assurant la deuxième phase de la sélection. Après une comparaison du filtrage et de l'unification dans un cas général, nous verrons les particularités et spécificités de notre contexte Bases de Connaissances, ainsi que leurs conséquences sur l'unification en termes de sélection. Nous terminons ce chapitre et cette partie par les premières orientations du projet, quant aux outils utilisés pour la mise en oeuvre du filtre: les automates, dont nous préciserons les limites, toujours vis à vis de notre contexte.





PRESENTATIONS

CHAPITRE I

RESUME

Après la présentation du langage PROLOG, sous ses aspects manipulations et représentations des connaissances, nous décrirons rapidement les fondements théoriques sous-jacents du langage. Nous verrons ensuite que les problèmes liés au stockage et à l'exploitation de volumes importants d'informations ont donné naissance à des "Systèmes de Gestion de Bases de Données" (SGBD). Nous proposons d'utiliser PROLOG pour réaliser des "SGBC" répondant aux insuffisances des SGBD, quant aux possibilités de raisonnement. Le sujet de la thèse sera alors précisé, ce dans le contexte du projet OPALE.

Chapitre I

PRESENTATIONS

Le projet de cinquième génération, comme tout projet d'envergure, a nécessité de déterminer un certain nombre de domaines dans lesquels les recherches avancent simultanément, les résultats pouvant être intégrés pour atteindre le but final. La communauté informatique s'accorde sur le fait que l'émergence d'une cinquième génération ne peut être réalisée que par une symbiose des résultats des recherches effectuées dans les trois domaines suivants:

- La microélectronique et plus particulièrement la technologie VLSI, apportant la puissance matérielle nécessaire à des applications toujours plus exigeantes.

- L'architecture des machines et des systèmes, supportant plus ou moins efficacement les spécificités nouvelles des applications.

- L'Intelligence Artificielle, fournissant la puissance logicielle. Curieusement ce domaine entraine l'apparition de besoins nouveaux que doivent supporter les deux autres, mais semble également apporter quelques éléments de réponse aux problèmes ainsi soulevés.

Le projet OPALE se situe dans le domaine de l'architecture des machines. A l'intérieur de ce domaine on peut grossièrement regrouper les fonctionnalités attendues des machines en trois catégories:

- Fonction de résolution de problèmes par inférences.

- Fonction de gestion de Bases de Connaissances.

- Fonction d'interfaces "intelligentes".

Au vu de cette classification le projet OPALE s'intéresse aux fonctions de gestion des connaissances (connaissances au sens large, c'est à dire incluant les règles générales et les faits élémentaires). Plus précisément il a pour objectif la conception d'une machine offrant toutes les fonctions nécessaires à la manipulation de volumes importants de connaissances. Le sujet de la thèse, quant à lui, porte sur la spécification d'un des éléments de la machine.

Dans le paragraphe suivant, nous présenterons le domaine concerné par le projet, à travers un premier aperçu du langage PROLOG, outil de base du sujet. Puis nous définirons le projet OPALE, et enfin le sujet lui-même.

I.1. PROLOG

A la question: "Pourquoi choisir le langage PROLOG comme point de départ d'un projet tel OPALE ?" on répondra: "Pour ses aptitudes intrinsèques à représenter et manipuler des connaissances" [AVRb 81], [LAU 82]. Pour ceux qui ne connaitraient pas le langage une telle réponse peut paraître gratuite. C'est pourquoi nous le présenterons ici sous cet aspect, les "initiés" voulant bien excuser cette vision unique, tant il est vrai que l'on peut parler de PROLOG de bien différentes manières (avec ou sans 's' ...). A la suite de cette approche pragmatique nous verrons également les bases théoriques sous-jacentes qui permettont, dans la suite, de justifier les orientations du projet.

I.1.1. Le langage

Rappelons tout d'abord que le langage PROLOG a été élaboré par A. Colmerauer dans le contexte de travaux sur la compréhension des langues naturelles [ROU 75], [COL 79].
De nombreux articles ont maintenant été consacrés à sa présentation [CLO 81], [COL 83], [LEG 84] ... Ceci est bien entendu lié à sa célébrité nouvelle, mais aussi aux différentes façons de l'aborder. Voici rapidement comment on peut l'approcher dans un esprit de manipulation et représentation des connaissances:

- L'unité de base du langage est l'expression d'une connaissance sous forme d'un théorème de la logique du premier ordre (nous expliciterons en détails ce point théorique au chapitre suivant). Ainsi il n'existe pas de différence entre ce que l'on appelle, dans un langage "classique" du type ALGOL, "donnée" et "programme"; le premier concept est matérialisé par l'expression d'une connaissance élémentaire, appelé fait ou assertion, alors que le deuxième est représenté par une connaissance conditionnelle que l'on appelle règle. Dans les deux cas la représentation utilise un symbolisme unique (la clause) ce qui permettra de manipuler sans distinction connaissances élémentaires et conditionnelles.

- L'utilisation de connaissances exprimées sous formes de clauses (pour des raisons de performances PROLOG n'utilise que des clauses dites de HORN que nous définirons un peu plus loin) consiste à exprimer sa requête également sous la forme d'une clause. Dès lors l'interpréteur tentera de démontrer le ou les théorémes exprimés dans cette clause, en fonction de ceux énoncés par l'utilisateur.

Voici un exemple sur la représentation et l'exploitation d'un arbre généalogique:

- On choisit de définir "naturellement" l'arbre à l'aide des relations "Père" et "Mère".

- Le fait que Colette soit la mère de Véronique est noté:

```
Mère (Colette,Véronique) -> ;
```

- Exprimons un arbre généalogique:

```
Père (Luc,Mathieu) -> ;          Mère (Nadine,Mathieu) -> ;
Père (Michel,Luc) -> ;           Mère (Colette,Luc) -> ;
Père (Michel,Véronique) -> ;     Mère (Colette,Véronique) -> ;
Père (Roger,Michel) -> ;         Mère (Adrienne,Michel) -> ;
Père (Roger,Jean-Claude) -> ;    Mère (Adrienne,Jean-Claude) -> ;
```

- Enonçons également diverses règles de parenté:

```
Grand-Père (X,Y) -> Père (X,Z) Père (Z,Y) -> ;
Grand-Père (X,Z) -> Père (X,Z) Mère (Z,Y) -> ;
/* X est le grand-père de Y si X est le père d'un Z tel que Z
soit le père ou la mère de Y (X, Y, Z étant des "variables") */
```

- Plus généralement on exprime la règle ancêtre par:

```
Ancêtre (X,Y) -> Parent (X,Y) ;
Ancêtre (X,Y) -> Parent (X,Z) ancêtre (Z,Y) ;
Avec:
Parent (X,Y) -> Père (X,Y) -> ;
Parent (X,Y) -> Mère (X,Y) -> ;
```

L'exploitation de cet arbre revient alors à exprimer ses requêtes sous la forme de clauses comme le montrent les exemples suivants.

Exemple: "Quelle est la descendance de Michel ?"

-> Ancêtre (Michel,X) ;

l'interpréteur fournit les réponses:

X = Luc ; X = Mathieu

Exemple: "Quels sont les ancêtres féminins de Mathieu ?"

-> Ancêtre (X,Mathieu) Femme (X) ;

avec: Femme (X) -> Mère (X,Y) / ;

l'interpréteur fournit les réponses:

X = Nadine ; X = Colette ; X = Adrienne

L'exemple ci-dessus nous montre les principales caractéristiques du langage:

- Les structures de contrôle "classique" n'existent pas. Les seules possibles sont celles exprimées dans l'énnoncé des règles (de type SI/ALORS) et celles issues de l'utilisation de la récursivité.

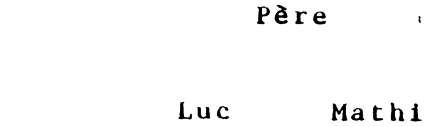
- Les variables ont diverses utilisations et servent notamment:
  . dans l'expression des règles,
  . pour préciser les éléments recherchés,
  . comme connaisance incomplète.

- Il n'y a pas de distinction entre les variables d'entrée et de sortie. De ce fait la question: -> Père (X,Mathieu), peut-être soit une interrogation si X n'est pas connue (libre) au moment de l'appel, soit une vérification en cas contraire (X liée).

- L'évaluation d'une requête est non déterministe en ce sens qu'elle retourne toutes les solutions possibles.

Insistons sur une caractéristique fondamentale du langage, non clairement exprimée dans ces exemples: les connaissances peuvent être structurées sous forme d'arbre, une forme dégénérée étant la liste.

En effet "Père(Luc,Mathieu)" a une structure d'arbre:



Cet exposé rapide des possibilités de PROLOG en tant que langage de représentation et de manipulation des connaissances ne veut pas conclure que c'est le seul langage ayant de telles capacités. Il se trouve que la puissance qu'il offre pour réaliser certaines de ces applications est de loin supérieure à des langages classiques (pour s'en convaincre il suffit d'essayer de programmer en PASCAL l'exemple ci-dessus). Pour l'heure si ces capacités ne se discutent pas, PROLOG souffre de temps d'exécution trop élevés. Ceci est principalement du à l'incapacité des machines actuelles à supporter les mécanismes de base sous-jacents à l'interprétation. Ceci n'est d'ailleurs nullement une surprise si l'on compare ces mécanismes de base à ceux des langages classiques, collant à l'architecture des machines. L'idée est alors de discerner les mécanismes fondamentaux sous-jacents du langage et de concevoir un matériel adéquat. Pour ce faire nous allons étudier les fondements théoriques de PROLOG.

I.1.2. Les fondements théoriques

D'un point de vue théorique, un programme PROLOG est constitué d'un ensemble de théorèmes de la logique du premier ordre, que l'on appellera clauses. Ces clauses sont formées de prédicats (encore appelés littéraux) dont les arguments sont considérés être quantifiés universellement (les prédicats ne pouvant l'être en logique du premier ordre sinon on passe dans des logiques d'ordre 2, 3 ...). Dans ce domaine de la logique, PROLOG a pour caractéristique de ne manipuler que des clauses, qui mises sous leur forme normale conjonctive, ne possèdent qu'un seul littéral positif (cette restriction étant liée à des considérations d'efficacité). Ces clauses sont dites "de HORN".

Exemple:

(non h1) ou (non h2) ... ou (non hn) ou c

Ce qui s'interprète naturellement par:

h1 et h2 et ... et hn => c

Ceci posé, un interpréteur PROLOG est en fait un démonstrateur de théorèmes sur cet espace des clauses de HORN, basé sur le principe de la résolution [ROB 65]. Ce principe repose sur le fait que

démontrer:

p => q

est équivalent à prouver que la proposition:

p ou (non q)

est contradictoire.

A partir du théorème à démontrer, transformé en sa négation, l'interpréteur tente d'unifier le premier littéral du théorème avec l'un des littéraux positifs des règles constituant le programme (l'unification consiste à égaler deux littéraux, nous définirons cette opération au cours des chapitres suivants).
Si une unification réussit, l'interpréteur remplace le littéral négatif unifié par la conjonction de littéraux négatifs liés au littéral positif unifié. L'opération est alors itérée sur la nouvelle liste de littéraux.

Exemple:

Soit la question: p et q
transformée en : (non p) ou (non q)
et soit la règle: p1 et p2 => p
transformée en : (non p1) ou (non p2) ou p
Le premier pas de l'interpréteur "résout" non p de la question avec p de la règle, si l'unification des deux littéraux est possible. La liste de littéraux à contredire devient alors:

(non p1) ou (non p2) ou (non q)

Deux issues sont possibles lors de la tentative d'unification d'un littéral négatif:

- Soit il existe un littéral s'unifiant et l'interpréteur avance dans la résolution, ce jusqu'à aboutir à une liste de contradictions vide. Dans ce cas une solution est trouvée.

- Soit aucune unification du littéral courant n'est possible. Il y a donc un échec dans cette branche de la résolution.

Dans les deux cas l'interpréteur va revenir en arrière d'un pas. Le retour consiste à restituer le contexte précédant la dernière unification et rechercher si d'autres unifications sont possibles pour le littéral de la liste de contradictions, soit

antérieurà celui ayant provoqué l'échec, soit ayant provoqué l'apparition de la liste vide.

Ce parcours de l'arbre de solution (la stratégie de recherche), connu sous le nom de "depth-first, left-to-right" (parcours préfixé), permet un parcours exhaustif de l'arbre et autorise ainsi la génération de toutes les solutions, le principe de résolution étant une méthode de déduction logique complète.

Dans la pratique, tout interpréteur PROLOG, permet de couper certaines branches de l'arbre, à l'aide d'un "prédicat évaluable" couramment appelé "cut" (que l'on note "/"). Un prédicat évaluable est un prédicat prédéfini dans l'interprète lui-même et non par l'utilisateur. Si cette possibilité est contestable d'un point de vue théorique (elle n'assure plus la complétude de la résolution), en pratique elle est nécessaire pour faire de PROLOG un langage de programmation utilisable (notamment pour éviter les boucles infinies ainsi que pour des considérations de performances).

Notons que ce raisonnement par l'absurde est connu en Intelligence Artificielle sous le nom de "chaînage arrière" [AVRb 81], c'est à dire que les inférences sont guidées par le but. A. Colmerauer, quant à lui, définit PROLOG comme un "système d'effacement" (réécriture) [COL 82]. Nous verrons au chapitre suivant que cette vision se rapproche plus de l'implantation de l'interprétation, ainsi que de l'utilisation en tant que langage de programmation.

I.2. Stockage et manipulation d'informations

Si PROLOG est sans conteste un outil puissant pour représenter et manipuler les connaissances, il ne résout pas pour autant les problèmes liés à la gestion de volumes importants d'informations. Ces problèmes ont donné naissance à ce que l'on appelle des Systèmes de Gestion de Bases de Données (SGBD). Aprés un survol de ces systèmes, nous verrons les orientations actuelles des recherches s'efforçant d'accroître leurs fonctionnalités, ce en s'appuyant sur des considérations issues de la logique mathématique. Nous préciserons également la terminologie et plus particulièrement ce que l'on entend par Bases de Connaissances.

I.2.1. Bases de Données classiques

Pour résumer la définition du concept de Base de Données (BD) on dira: une BD est l'instanciation de la modélisation d'une application, modélisation elle-même définie à l'aide d'un modèle de données. Cette BD est alors manipulée par un SGBD (qui fournit le modèle de données) offrant différentes fonctions: interrogation, modification, ... De plus le SGBD assure la vérification de diverses contraintes: intégrité des données, sécurité, confidentialité ...

Les SGBD actuels se décomposent en trois grandes familles, ce en fonction du modèle de données qu'il propose:

- Modèle réseau: il permet de représenter et relier les informations sous forme de graphe, les noeuds stockant les informations élémentaires, et les arcs entre les noeuds les associations entre informations .
  (SGBD: SOCRATE, CODASYL)

- Modèle hiérarchique: il suit les mêmes principes et conventions que le modèle réseau, mais le graphe est réduit à une arborescence. En d'autre termes il existe un noeud racine n'ayant aucun arc entrant, et tous les autres noeuds ont au plus un arc entrant. Par contre tout noeud peut avoir plusieurs arcs sortants, donc plusieurs liens vers d'autres informations.

(SGBD: IMS, SYSTEME 2000)

- Modèle relationnel (n-aire): les informations sont structurées sous forme de relations n-aires, au sens mathématique du terme, une information élémentaire étant matérialisée par un n-uplet. (SGBD: SYSTEM R, INGRES, QBE, SQL/DS)

Discuter des particularités, qualités et défauts de chacun nous entraînerait trop loin et nous renvoyons le lecteur à l'ouvrage référencé [DEL 82] pour une présentation complète du modèle relationnel avec en intoduction (étendue) une présentation et comparaison de ces différents modèles.

Pour l'heure la constatation qui nous intéresse est la suivante: aucun de ces modèles ne permet de définir une connaissance générale, les informations représentées étant toutes des faits élémentaires. Qui plus est, ces faits n'ont pas de structure propre en ce sens que ce sont des objets atomiques (par exemple il est impossible de fournir un arbre pour valeur d'un des constituants d'une relation).

Ces limitations étant apparues bien vite comme trop contraignantes, les recherches se sont efforcées de définir des mécanismes offrant la possibilité de représenter et d'utiliser ce que l'on appelle en Base de Données des lois générales.

I.2.2. Bases de Données Déductives

Si des lois générales sont déjà utilisées par les SGBD, leur rôle est d'exprimer des contraintes d'intégrité. En ce sens elles ont une fonctionnalité bien spécifique et n'autorisent que l'expression d'une connaissance de vérification et non de déduction. D'autre part elles sont représentées et manipulées différemment des données, nous reviendrons sur ces points pour expliciter leurs conséquences.

Une idée est d'utiliser ces lois aussi bien en vérification d'intégrité qu'en déduction [REI 78], [MIN 78], [GAL 83]. Les travaux en ce sens s'appuient sur le formalisme mathématique développé autour du modèle relationnel (dans la suite nous nous intéresserons

uniquement à ce modèle, des recherches étant également en cours sur les autres [JAC 82]). Dans une première étape, ces travaux [COD 70] eurent pour objectif d'aider à la conception d'un "bon" schéma de BD relationnelle, et ils ont abouti à un processus de "normalisation" [COD 71], [DEL 78]. Depuis on s'efforce d'élargir les possibilités du modèle en s'appuyant essentiellement sur la logique du premier ordre.

Plus précisément on utilise les résultats obtenus dans le domaine du calcul des prédicats du premier ordre. Un tel système formel possède:
- un langage du même ordre,
- un ensemble de schémas d'axiomes,
- deux règles d'inférences qui sont le modus ponens et la généralisation.

Plusieurs auteurs ([NIC 78], [GAL 78] ...) ont montré et discuté le fait qu'une BD "classique" pouvait être formalisée, dans le cadre de la logique du premier ordre, de deux manières:
- Soit comme une interprétation d'une théorie du premier ordre.
- Soit comme une théorie du premier ordre.

Les deux modélisations représentent respectivement les approches BD classiques et déductives. Dans l'une et l'autre, il est nécessaire de poser trois axiomes traduisant les conventions habituelles des BD:
- Représentation des infos négatives (si une information n'est pas dans la base, son contraire est considéré l'être).
- Unicité de nom.
- Fermeture des domaines.

Ces points communs exposés explicitons succintement en quoi consiste l'une et l'autre des deux approches.

I.2.2.1. Interprétation ou BD classiques

Cette modèlisation considère tous les faits de la BD comme des interprétations des formules du langage du premier ordre de la théorie sous-jacente. Dès lors les lois générales, réduites aux contraintes d'intégrité dans les BD classiques, sont exprimées en

tant que formules. Prenant en compte les axiomes cités ci-dessus, on dira que la BD est valide si ces formules sont évaluées à vrai pour l'état courant de la Base. Théoriquement celà revient à prouver que la BD, exprimée dans ce formalisme, est un modèle pour les lois générales (les contraintes d'intégrité).

I.2.2.2. Théorie ou BD déductives

Cette approche consiste à considérer tous les faits de la BD, non plus comme des interprétations de formules, mais comme les axiomes d'une théorie du premier ordre. Les lois générales également exprimées sous cette forme, sont alors à ajouter à cette théorie, ainsi que les axiomes assurant les conventions habituelles.

Dans ce contexte les lois générales ne sont plus restreintes à des contraintes d'intégrité. En effet les questions posées à la Base seront considérées comme des théorèmes à démontrer, et la démonstration pourra utiliser tous les axiomes présents dans la Base donc entre autres des axiomes traduisant des lois générales de déduction.

L'intégrité de la Base est assurée par le déclenchement d'un mécanisme de vérification de validité de la théorie, ce dès qu'un axiome est ajouté, supprimé ou modifié. On conviendra qu'une telle mise en oeuvre va être sujette à difficultés, principalement quant au temps d'exécution de par l'explosion combinatoire de la preuve à construire à chaque modification de la Base.

Plusieurs problèmes, non encore résolus, sont donc posés par les deux approches précédentes, l'un (et non des moindres) étant l'accroissement du temps de traitement impliqué par l'utilisation des lois générales que ce soient en intégrité ou en génération. Pour ce dernier point l'utilisation d'un matériel spécialisé devrait apporter une amélioration sensible.

I.2.3. Terminologie

Nous voulons préciser ce que l'on entend dans cet ouvrage par la terminologie "Base de Connaissances" (notée BC), ce terme étant déjà employé par ailleurs, dans un sens quelque peu différent.

Ainsi dans le domaine des Systèmes Experts une BC est constituée uniquement de règles (ce que l'on a appelé pour une BD les lois générales), les faits élémentaires étant séparés et regroupés dans une Base de Faits.
L'exploitation de ces deux Bases se fait à l'aide d'un "moteur d'inférence" qui ne s'appuie pas forcément sur une logique du premier ordre, ni même sur une logique complète [ROCa 84].

Sans se démarquer aussi nettement du contexte de la logique du premier ordre certains auteurs, parlant d'applications Bases de données réalisées en PROLOG, séparent la Base de Données "Intensionnelle" (BDI) représentant les lois générales, et la Base de Données "Extensionnelle" (BDE) représentant les faits [GAL 78] (PROLOG étant alors considéré comme le "moteur d'inférence" manipulant ces Bases).

Pour notre part on regroupera BDE et BDI en un même vocable BC dans le sens où les mécanismes développés au cours de cet ouvrage manipulent aussi bien les uns que les autres. Ceci dit nous n'écartons pas, loin s'en faut, la possibilité de séparer BDE et BDI, pour des raisons de performances voire même d'exploitation, mais nous restons réservées quant au choix d'utiliser un interpréteur PROLOG classique pour manipuler ces Bases. Par contre nous exploiterons le fait que raisonnablement la BDE aura un nombre d'éléments nettement plus important que la BDI, fait sur lequel tout le monde s'accorde. On fournit en annexe 1 un exemple de BC, développant l'exemple généalogique présenté au paragraphe 1 de ce même chapitre.

Précisons qu'en ce qui concerne le projet OPALE, l'idée d'utiliser PROLOG pour des applications base de données nous a été suggérée par J. Rhomer. De plus nous verrons au chapitre III l'apport, pour le projet, de ses travaux sur les automates [RHO 80], et plus particulièrement dans leur application au filtrage.

I.3. Le sujet

Le sujet de cette thèse est à situer par rapport au projet OPALE en tant qu'élément de ce dernier. Nous présenterons tout d'abord les grandes lignes de ce projet en renvoyant le lecteur désireux de compléments à plusieurs ouvrages, notamment à la thèse d'état de G. Berger Sabbatel, [BER 85], [BER 82]. Cet aperçu fourni, nous pourrons alors spécifier les objectifs et donner le plan de cet ouvrage.

I.3.1. Le projet OPALE

En regard des présentations précédentes, le projet OPALE (Organisation PArallèle et LogiquE) porte sur la conception d'une machine supportant efficacement le langage PROLOG, ce dans un contexte d'utilisations de grandes bases de connaissances (représentées bien sur en PROLOG). Pour ce faire on envisage de réunir et d'approfondir plusieurs idées développées jusqu'à présent séparément:

- Choix de PROLOG en tant que langage de représentation et de manipulation de la base de connaissances.
- Introduction de parallélisme quant à l'exécution des tâches manipulant la BC, et ce en plusieurs niveaux.
- Utilisation de mécanismes de filtrage.
- Exploitation des possibilités issues de la technologie VLSI.

Ces suggestions sont les conclusions des considérations suivantes:

- PROLOG peut-être considéré comme un sur-ensemble de l'algébre relationnelle et permet, en ce sens, d'exprimer aisément tous les opérateurs de cette algèbre. Qui plus est, il offre des possibilités nouvelles quant:
  . aux déductions.
  . à la représentation de structures complexes telles les arborescences (un cas particulier mais fréquent étant celui des listes).

  Dès lors utiliser PROLOG à la fois pour représenter et manipuler les connaissances permettra d'une part d'éviter les goulots d'étranglement que provoquerait une interface entre PROLOG et une

Base Relationnelle, et d'autre part d'accroître la puissance du système.

- Dans un contexte d'importantes Bases de Connaissances deux particularités apparaissent:
  . les ensemble de faits, les faits étant équivalents en première approche aux n-uplets des relations, auront des cardinalités élevées.
  . les règles quant à elles devraient, d'une part avoir une complexité relativement faible, et d'autre part l'orde d'essai de leurs alternatives sera le plus souvent indifférent.

  Ces particularités permettront, d'utiliser des mécanismes d'interprétation efficaces, notamment de par l'introduction de parallélisme important, point que nous développerons en détail au chapitre III.

- Toujours en regard du contexte, la plupart des unifications seront, d'une part triviales (la plupart des faits ne contenant pas de variable), et d'autre part se solderont dans la plupart des cas (>90%) par un échec. Là encore l'utilisation de mécanismes appropriés (le filtrage [RHO 80]) réduira le coût des opérations fondamentales (entre autres l'unification).

- Enfin on compte sur le fait que la technologie VLSI apportera les moyens de concrétiser ces idées.

Dès lors, les premiers objectifs seront de définir une méthode d'interprétation par ensemble de solutions, à l'inverse d'une interprétation classique manipulant une solution à la fois, et d'intégrer l'utilisation d'un filtre permettant de réaliser l'unification sur le flux de données issu du disque.

L'architecture d'OPALE est visualisée à la figure I.1. Elle découle directement des deux objectifs décrits ci-dessus. Pour ce faire la machine est répartie en plusieurs opérateurs, connectés par un système de communication permettant de construire dynamiquement des réseaux de résolution. La répartition des informations sur plusieurs unités de disque de taille moyenne augmentera d'autant les possibilités de parallélisme et permettra, en cas de panne d'une des unités, d'assurer un fonctionnement dégradé.

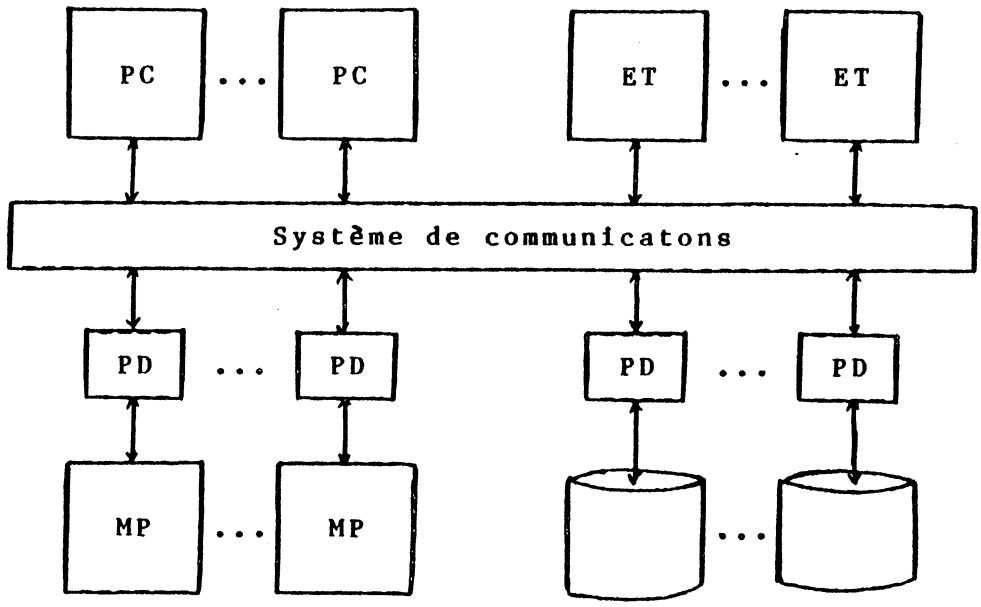


Figure I.1

Voici rapidement les fonctions des différents types d'éléments:

- Les processeurs de contrôle (PC) assurent d'une part l'interface avec l'extérieur (utilisateur ou autre machine), et d'autre part assurent dynamiquement la répartition des traitements à effectuer.

- Les éléments de traitement (ET) mconstruisent les requêtes vers les éléments de mémorisation en fonction des sollicitations des éléments de contrôle et gèrent les ensembles de solutions retournées permettant ainsi une vérification en pipe-line des règles.

- On accède aux mémoires primaires (MP) et aux unités de disque par des processeurs disques (PD) assurant le filtrage de données en cas d'interrogation. Les premières servent à stocker les résultats intermédiaires des différentes stations de pipe-line. Les unités de disque, quant à elles, mémorisent les informations permanentes.

Nous en resterons là pour la présentation de la machine, cet aperçu étant suffisant pour fixer les objectifs nous concernant pour l'heure (pour plus de détails voir [BER 85]).

I.3.2. La thèse

Le sujet de la thèse s'appuie sur plusieurs des idées directrices du projet:

- Interprétation par ensemble de solutions.
- Utilisation d'unificateurs câblés au niveau des canaux d'échange, mélant en un seul mécanisme filtrage et unification.
- Intégration en VLSI de ces unificateurs.

Plus précisément les objectifs de cette thèse sont d'une part l'élaboration des algorithmes permettant d'effectuer l'unification dans le contexte d'une interprétation par ensemble de solutions, et d'autre part la spécification d'un opérateur matériel pouvant supporter ces algorithmes. Brossons rapidement les différentes étapes parcourues pour parvenir à nos fins.

Partant des travaux effectués quant au filtrage dans les bases de données relationnelles, nous avons progressivement adopté idées et outils à nos besoins. Au chapitre III nous mettons en évidence certaines similitudes, dans notre contexte, entre filtrage et unification.

Si ces réflexions sur le filtrage ont fourni le point de départ, très vite elles n'ont plus permis de respecter les contraintes liées à la philosophie de la machine. En accord avec la stratégie de recherche par ensemble de solutions, il a fallu définir un algorithme permettant d'unifier non pas un, mais plusieurs buts en une seule activation: ce sera l'objet du chapitre IV, où l'on définit la préunification comme une condition nécessaire de l'unification.

Le chapitre V fournit la validation des algorithmes par leur programmation. D'ores et déjà on s'est attaché à prendre en compte l'objectif matériel en extrayant plusieurs primitives de l'algorithme de préunification. Différentes mesures confirment les espérances de gains quant aux temps d'exécution.

Ces gains semblant prometteurs, on a poussé un peu plus loin la

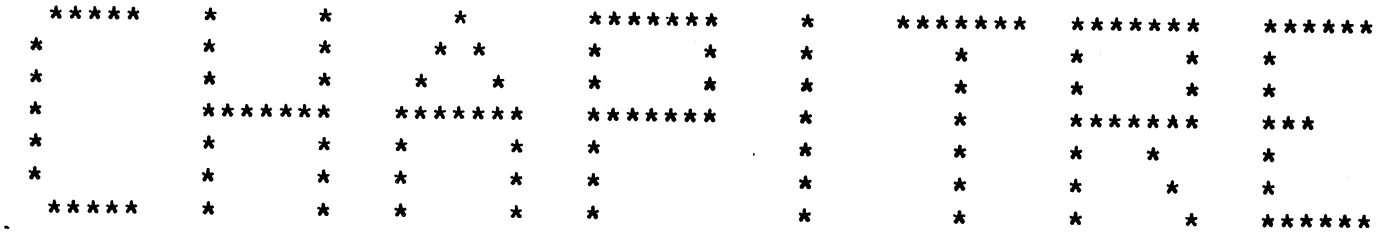
partie logicielle pour aboutir à l'intégration d'une maquette logicielle du filtre à un interpréteur classique. Nous explicitons au chapitre VI cette mise en oeuvre.

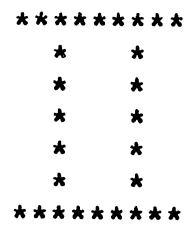
Nous proposons au chapitre VII une ébauche de méthodologie pour la conception d'une architecture modulaire, capable d'exploiter au mieux le parallélisme exprimé dans un algorithme. Cette méthodologie s'appuie en fait sur les concepts issus de la programmation orientée objet, que nous présenterons.

Nous appliquerons cette méthodologie pour assembler les différentes primitives de l'algorithme de préunification, le chapitre VIII abordant l'aspect matériel de ces primitives et de l'unificateur en général.

Enfin avec le chapitre IX nous esquissons les premiers pas de la réalisation matérielle en proposant un aperçu des outils utilisables pour aboutir à la conception d'une maquette: langage de description de circuits, compilateur de silicium ...

Avant de se lancer dans ces recherches, il était nécessaire d'avoir une connaissance approfondie du constituant essentiel du projet: PROLOG. C'est pourquoi nous débuterons par la présentation détaillée de l'interprétation de ce langage et ferons le point quant aux dérivés ayant vu le jour en essayant soit de l'étendre soit d'améliorer son efficacité. Au terme de cette présentation nous exposerons l'interpréteur que nous avons mis en oeuvre, outil qui nous servira tout au long de l'étude.





PROLOG: IMPLANTATION ET STRATEGIE

CHAPITRE II

RESUME

Après un rappel de la terminologie, nous exposons PROLOG sous son aspect langage de programmation: mécanisme d'unification, sémantique procédurale. Nous abordons alors un des points essentiels vis à vis de la mise en oeuvre du langage: la stratégie de recherche. Nous étudions ensuite les travaux entrepis, quant à une amélioration de la stratégie "depth-first, left-to-right" de PROLOG, ayant donné naissance à des langages "cousins". Le chapitre se termine par l'exposé de l'implantation d'un interpréteur classique, utilisé comme outil de développement pour le projet, et les problèmes liés à un contexte Base de Connaissances.

Chapitre II

PROLOG: IMPLANTATION ET STRATEGIE DE RECHERCHE

Pour l'heure l'utilisation de la logique du premier ordre semble la voie la plus prometteuse quant aux recherches sur les BD. Nous avons vu qu'une BD pouvait être formalisée comme une **interprétation** d'une théorie du premier ordre ou comme une **théorie** du premier ordre. Si la première vision fournit un formalisme aisé à mettre en oeuvre elle n'offre pas autant de perspectives que la deuxième, qui à l'inverse s'avère plus délicate à implanter.

Formaliser une BD sous la forme d'une théorie du premier ordre permet d'envisager d'utiliser PROLOG comme langage de cette théorie. Un programme PROLOG peut alors être considéré comme une BD déductive que l'on appellera Base de Connaissances. Rappelons, point important pour la suite, que ces programmes auront les particularités, d'une part de posséder un nombre d'assertions faits élémentaires) élevées, et d'autre part de n'utiliser que des règles d'une complexité limitée.

Ceci dit, un interpréteur PROLOG classique ne saurait en aucun cas être considéré comme un "SGBC" pour au moins deux raisons:

- Son manque de possibilités quant à la définition de chemin d'accès. Plus précisément, il est impossible de créer des index permettant d'accéder efficacement aux alternatives d'un paquet.

- L'inefficacité de sa méthode d'interprétation, dans ce contexte de Bases de Connaissances.

En revanche PROLOG permet d'implanter diverses fonctions d'un "SGBC" telles le maintien de la cohérence, la vérification des contraintes d'intégrité, la définition de la structure de la base...

De plus il peut être utilisé à la fois comme langage de représentation et de manipulation de la base.

Une des solutions pour résoudre les problèmes d'accès est d'interfacer PROLOG avec un SGBD [CHA 82], [GAL 83]. L'idée est attrayante mais on peut se demander si le goulot d'étranglement résidant en l'interface "PROLOG <-> SGBD" pourra être supprimé. D'autre part on perd de la souplesse, voire des possibilités, quant à la représentation des connaissances.

Ces réflexions nous ont amenés plutôt à envisager la réalisation d'un "SGBC" réel. Dans une première étape nous avons délaissé les problèmes de chemins d'accès ne posant pas de difficultés nouvelles et pouvant être résolus à l'aide des techniques classiques, pour nous concentrer sur la définition d'une méthode d'interprétation prenant en compte le contexte BC.

II.1. PROLOG "classique"

Au paragraphe 1.1. nous avons présenté le langage PROLOG sous son aspect déclaratif, puis les bases théoriques sous-jacentes. D'un point de vue programmation, PROLOG peut également être présenté sous un aspect procédural, comme un langage classique. L'implantation d'un interpréteur se base plutôt sur cette vue du langage. Aprés un rappel de la terminologie utilisée, nous décrirons successivement: l'unification, l'aspect procédural du principe de résolution, la stratégie d'un interpréteur classique.

II.1.1. Terminologie et notations

De nombreuses variantes d'un point de vue terminologie et notations sont en usage; nous présenterons celles que nous utiliserons par la suite.
Un programme PROLOG est constitué d'un ensemble de **règles.** Une règle est formée d'une **tete de règle** et d'une **queue de règle.** Nous utiliserons "->" pour séparer la tête de la queue de la règle et terminerons les règles par ";".
Une tête de règle ne comporte qu'un seul **littéral** à l'inverse de la queue constituée d'une suite de littéraux (éventuellement vide). Un littéral est un **prédicat:** un **symbole fonctionnel** suivi de ses **arguments.** Le symbole fonctionnel peut se réduire à un atome (l'arité d'un atome est nulle car il n'a pas d'arguments). Tous ces éléments sont représentés sous forme d'arborescences appelées **termes** .

Exemples:

```
P(X,Y) -> P1(X) P2(Y) ;
P(a,b) -> ;
atome -> ;
```

où les lettres majuscules représentent les variables (les notations classiques consistent à préfixer les identificateurs des variables: "*" , "_", "première lettre en majuscule", ...).

On appelle **paquet de règles** l'ensemble des règles définissant un prédicat, c'est à dire ayant même littéral de tête. Deux littéraux sont semblables s'ils possèdent même symbole fonctionnel

et même nombre d'arguments.
Un prédicat **évaluable** est non pas défini par l'utilisateur mais par l'interpréteur lui-même (par exemple les prédicats d'entrée sortie).

Ceci posé voyons une des opérations de base du langage: l'unification.

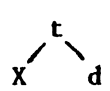
II.1.2. Unification

Nous présenterons ici l'unification de façon informelle, un algorithme étant proposé au chapitre IV.

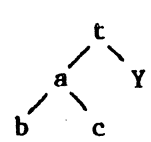
L'unification est une opération portant sur deux termes. Elle consiste à comparer les deux arbres sous-jacents, noeud à noeud. Plus précisément, il s'agit de comparer les noeuds de même position dans l'un et l'autre des deux arbres en substituant, éventuellement, les parties variables de l'un ou de l'autre par le sous-arbre correspondant (noeud de même position). On appelle substitution les couples (variables,valeurs) ainsi générés. Si les comparaisons effectuées sur chaque noeud réussissent alors l'unification se termine par un succès sinon par un échec.

Exemple:

t(X,d)



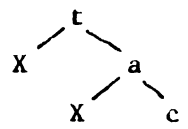
t(a(b,c),Y)



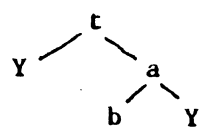
s'unifient en substituant: X par a(b,c) et Y par d

Exemple:

t(X,a(X,c))



t(Y,a(b,Y))



ne s'unifient pas car:
- il faut substituer X par Y (ou vice versa)
- ce qui entraîne qu'il faille par la suite substituer:
  Y par b et Y par c ce qui est évidemment impossible.

Signalons à l'occasion de ce deuxième exemple, une caractéristique importante de l'unification: substituer une variable par un terme signifie qu'il faut remplacer **toutes** les occurences de la variable par ce terme. Nous verrons par la suite les problèmes que cela pose quant à l'implantation logicielle, et plus encore vis à vis d'une implantation matérielle.

II.1.3. Aspect procédural et interprétation

Percevoir PROLOG sous un aspect procédural revient à interpréter chaque règle comme une procédure:

- La tête de la règle représente, par le symbole fonctionnel et les arguments du littéral, le nom et les arguments formels de la procédure.

- La queue de la règle constitue le corps de la procédure (essentiellement des appels à d'autres procédures).
  Une règle ayant une queue vide est interprétée comme un fait.

On peut considérer qu'un programme PROLOG définit un arbre abstrait (l'arbre de résolution) qu'il faut parcourir dans son ensemble pour obtenir toutes les solutions possibles. La mise en oeuvre du mécanisme de résolution se concrétise alors par la construction de la "résolvante" (une partie de l'arbre de résolution) qui se compose:

- De la liste d'appels de procédures en attente, que l'on appelle plutôt liste des buts à satisfaire lorsque l'on considère PROLOG comme un démonstrateur de théorèmes.

- D'un contexte permettant de mémoriser les substitutions générées lors des unifications effectuées.

Une fois la liste des buts à satisfaire initialisée par la requête, et le contexte initialisé à vide, l'avancement dans la résolution s'effectue en trois étapes (on parle ici des choix à effectuer pour implanter le principe de résolution en général):
- Choix d'un des buts dans la liste de buts.
- Tentative d'unification de ce but avec l'une des têtes de règle du paquet concerné (définissant le prédicat que le but désigne).
- Si une unification se termine par un succès:
  . stockage dans le contexte des substitutions générées.
  . remplacement, dans la liste de buts, du but unifié par la queue de la règle unifiant.

Un retour arrière dans la résolution est provoqué par l'un des deux événements suivants:
- Soit la liste de buts devient vide: une solution est trouvée, il faut rechercher si il en existe d'autres.
- Soit aucune unification n'est possible pour le but courant: il faut remonter dans l'arbre pour prendre une autre branche.

Ce retour arrière nécessite deux opérations:
- Remonter dans la résolution jusqu'à un point de choix: en d'autres termes revenir sur un des buts à satisfaire précédemment choisis qui pouvait s'unifier avec une autre en-tête de règle.
- Restituer le contexte tel qu'il était avant l'unification du point de choix: annuler les substitutions (et leurs effets) effectuées à partir de ce choix.

La mise en oeuvre d'un interpréteur PROLOG consiste alors à compléter le mécanisme décrit ci dessus en définissant les fonctions de choix permettant de sélectionner le but courant ainsi que la règle à unifier. On regroupe ces fonctions sous une même dénomination: la stratégie de recherche.

II.1.4. Stratégie de recherche

La stratégie "classique" consiste à parcourir tout l'arbre de résolution, issu du programme, en "dept-first, left-to-right" c'est à dire en préfixé. Ainsi le but choisi dans la liste de buts de la résolvante sera le premier et il en sera de même pour les différentes alternatives d'un paquet de règles (l'implantation est détaillée au paragraphe 3 de ce même chapitre).

Cette stratégie pose différents problèmes quant aux facilités (voire possibilités) de programmation et plus encore vis à vis des temps d'exécution:

- elle n'autorise pas le programmeur à supprimer certaines branches de l'arbre (même à ses risques et périls ...). Nous avons vu au chapitre I que si d'un point de vue théorique cette possibilité est incorrecte, elle est nécessaire à une utilisation algorithmique et est offerte par les interpréteurs sous forme d'un prédicat évaluable, que l'on notera "/".
  Pour exemple voici comment programmer le non conventionnel des BD:

```
non (P) -> P / échec ; /* Si P est vrai alors non P est faux */
non (P) -> ;           /* Sinon non P est vrai */
```

- Elle est figée ce qui entraîne que l'on parcourt l'arbre toujours dans le même ordre. Ceci est pénalisant au moins dans les deux cas de figure suivant:
  . si on ne s'intéresse pas à toutes les solutions la recherche pour arriver à une des solutions peut, dans certains cas, être guidée par le contexte et ainsi être optimisée.
  . même si toutes les solutions sont recherchées elle peut de même entraîner un surcroît de travail inaceptable surtout dans un contexte BC.
  Certains interpréteurs, comme nous le verrons par la suite, autorise un contrôle sur l'ordonnancement des buts en attente de résolution.

Avant de fournir un aperçu des différentes propositions quant aux modifications possibles de cette stratégie, nous voulons

fournir un exemple montrant l'impossibilité de l'utiliser dans notre contexte BC.

Exemple:

- Soit le prédicat Père (X,Y) définit par un certain nombre d'assertions.
- Soit la question: "de qui Jean est-il le grand-père ?"

Si les deux expressions suivantes sont équivalentes d'un point de vue logique (elles fournissent la même réponse), il n'en sera pas de même quant à leur temps d'exécution, l'évaluation de la deuxième forme étant beaucoup plus coûteuse:

i) R1 -> Père(Jean,Y) Père(Y,Z) ecrit(Z) ;
ii) R2 -> Père(Y,Z) Père(Jean,Y) ecrit(Z) ;

Il ne faut pas s'arrêter dans cet exemple au seul problème d'optimisation de la requête, mais plutôt au fait que le mécanisme de base va chercher les solutions les unes après les autres (même pour la forme la plus optimale de la requête) et exécuter pour chacune des deux, un nombre de tentatives d'unification égal à:

$$Ns \times Cp$$

avec: Ns = Nombre de solutions du premier appel
et: Cp = Cardinalité de la relation père

Même en choisissant la meilleure implantation, en l'occurence R1, le fait de parcourir une seule branche à la fois n'est pas une stratégie acceptable dans un contexte BC de par la redondance du travail effectué.
Nous allons voir les différentes stratégies qui ont été envisagées (et implantées) pour pallier aux déficiences de la stratégie "depth-first, left-to-right" vis à vis des spécificités de certaines applications.

II.2. Langages dérivés: stratégies de recherche

En résumant ce que l'on a vu au paragraphe II.1.3., le contrôle d'une résolution implantée séquentiellement, c'est à dire ne parcourant qu'une seule branche de résolution à la fois, peut se faire soit lors de l'avancement, soit lors du retour arrière. Il consiste essentiellement dans les choix:

- Lors de l'avancement:
  - . du but, à résoudre.
  - . de la règle, à utiliser.
  - . de la sélection ou non des points de reprise pour le but courant.
- Lors du retour arrière:
  - . du point de reprise.

Une autre forme de contrôle consiste à parcourir en parallèle plusieurs branches. La solution extrême, inverse à la stratégie de PROLOG, est un parcours "breadth-first", c'est à dire toutes les branches en parallèle. L'explosion combinatoire dûe à cette approche nécessite de la modérer, voire de la conjuguer avec une stratégie séquentielle.

La suite propose une rapide revue des différents langages, hybrides de PROLOG, issus de cette volonté de contrôler la résolution et offrant différentes facilités au programmeur pour le faire.

II.2.1. IC-PROLOG

IC_PROLOG [CLA 82], est certainement le langage à la fois le plus proche de PROLOG et offrant le plus de possibilités de contrôle:

- Le plus proche car ses options par défaut sont celles de PROLOG.
- Le plus étendu quant aux possibilités de contrôle car permettant de le spécifier sous toutes ses formes: lors de l'avancement, lors du retour arrière, par spécification d'évaluation parallèle.

La plupart de ces particularités ont d'ailleurs été reprises séparément par les autres langages "cousins" de PROLOG.

La première particularité de IC-PROLOG, induisant un contrôle implicite, est de fournir un prédicat évaluable permettant de construire la liste de toutes les solutions d'une requête (possibilité reprise par la suite pour diverses implantations de PROLOG, par exemple le prédicat évaluable "setof" de C-PROLOG [PER 84]).

Exemple:

[(X,Y): Père(X,Z) Parent(Z,Y)]

fournit tous les couples tels que X soit le grand-père de Y.

Ceci dit la première forme explicite de contrôle réside dans la possibilité de poser des contraintes quant à l'état des variables. Ainsi la définition d'un prédicat n'est plus nécessairement constituée d'un ensemble **d'alternatives logiques** exprimant **l'aspect déclaratif** de la connaissance, mais peut-être également un ensemble **d'alternatives de contrôle** exprimant **l'aspect procédural** de la connaissance (dans ce deuxième cas, la vue d'ensemble du prédicat assure un aspect déclaratif).

Exemple: en reprenant la définition du prédicat grand-père, on le formulera comme suit à l'aide des deux règles suivantes:

```
gp(X?,Y) -> Père(X,Z) Parent(Z,Y) ;
gp(X,Y)  -> Parent(Z,Y) Père(X,Z) ;
```

- où "?" indique que le premier argument du littéral tentant d'unifier la première règle doit être lié lors de l'appel. Si ce n'est le cas, l'unification échoue.
- où "^" indiquera qu'un argument doit être une variable libre.

Cette expression du contrôle assure d'obtenir la résolution la plus efficace que l'appel soit de l'une ou de l'autre des deux formes suivantes:

```
-> gp(X,Jean) ;
-> gp(Jean,X) ;
```

Une autre forme du contrôle consiste à contraindre une variable d'un des prédicats d'une queue de règle, à être liée avant de lancer la vérification. Si ce n'est le cas l'appel est suspendu

jusqu'à l'instanciation de cette variable.

Exemple:

```
p(X,Y) -> r1(X!) r2(X,Y) ;
```

Si X est liée lors de l'appel de p(X,Y) alors l'interpréteur évaluera tout d'abord r1(X) sinon il évaluera en premier r2(X,Y), la vérification de r1 étant différée (suspendue) jusqu'à ce que X soit instanciée par un terme (différent d'une variable). Cette possibilité se trouve dans d'autres intrerpréteurs, notamment dans PROLOG II [CAN 82] sous la forme du prédicat évaluable "Geler".

Les autres possibilités de contrôle sont liées à une interprétation non séquentielle. On distingue trois formes agisant lors de l'avancement dans la résolution.

La première consiste à progresser en parallèle et sans contrainte dans un certain nombre de branches de l'arbre de résolution. Dans ce cas l'interpréteur partage équitablement son temps entre les différentes branches actives (le quantum de temps alloué à chaque branche permettant d'effectuer au moins une inférence). Ce mécanisme assure que toutes les variables communes ne sont manipulées que par une seule opération indivisible (unification) à la fois, et évite ainsi les problèmes posés par des résolutions effectuées indépendamment les unes des autres.

Exemple:

```
p(X,Y) -> p1(X,Z) // p2(Y,Z) ;
```

La notation "//" stipulant que p1 et p2 sont à évaluer en parallèle (la queue de la règle étant toujours d'un point de vue sémantique une conjonction de buts).

La deuxième possibilité est une extension de la première, construite de la même façon mais pour laquelle on autorise à contraindre les variables à l'aide des notations "?" et "^".

Exemple:

```
p(X,Y) -> p1(X,Z) // p2(Y,Z^) ;
```

indique que p2 est le producteur de Z. Ainsi p1 sera suspendu s'il essaie de lire Z et ne sera réactivé que lorsque p2 aura instancié Z (un seul producteur n'étant admis pour une variable, p1 sera donc considéré comme consommateur de Z).

Une troisième forme consiste à entremêler l'évaluation des différentes branches, sans que deux soient actives simultanément. Pour ce faire il suffit de supprimer le "//"

Exemple

p(X,Y) -> p1(X,Z) p2(Y,Z^)

L'évaluation alternera entre p1 et p2. Ce processus est connu sous le vocable de "lazy producer" (p2) et "eager consumer" (p1).

Notons que la combinaison de ces différentes formes de contrôle permet de guider en partie le retour arrière.

Pour la suite nous retiendrons essentiellement deux caractéristiques:

- la construction d'un ensemble de solutions, opération précieuse dans un contexte BC.
- La notion de producteurs-consommateurs au cours d'une évaluation parallèle, entraînant naturellement celle de pipe-line, aisée à implanter en matériel.

Pour conclure si IC-PROLOG offre des possibilités de contrôle très sophistiquées, leur utilisation risque de s'avérer délicate de par son imbrication avec le programme lui-même. Nous allons voir dans la suite comment d'autres langages proposent une séparation entre contrôle et programme.

II.2.2. METALOG

Une autre approche du contrôle consiste à utiliser des "méta-règles" indiquant, en fonction du contexte, l'ordre de parcours des différentes branches de l'arbre de résolution. Le terme "méta-règle" signifie que ces règles expriment des "méta-connaissances": connaissance sur la connaissance, ou comment utiliser la

connaissance. L'usage de méta-règles offre un contrôle total du déroulement de la résolution [FAH 79]. Notons que l'introduction de méta-connaissances peut-être étendue et dépasser l'utilisation particulière de contrôle, ce en définissant un "méta-langage" que l'on imbrique dans le langage [BOW 82].

Dans une optique de contrôle, règles et méta-règles sont généralement séparées les unes des autres. Des différents travaux effectués sur la question [FAH 79], [BOW 82], [GAL 82] nous décrirons rapidement ceux ayant abouti au langage METALOG [DIN 80].

METALOG est le moteur d'inférence du système expert PEACE. Construit sur les bases de PROLOG, il offre un mécanisme de contrôle par l'utilisation de méta-règles lors de l'avancement: choix du but à résoudre, choix de la règle pour le résoudre.

En plus de ces possibilités offertes au programmeur, le système inclut un mécanisme de détection d'échec, implanté par un test de préunification en largeur. Pour chaque règle sélectionnée le système vérifie que les substitutions générées lors de son unification n'entraînent pas un échec quant aux autres buts en attente de résolution.
A rajouter à ses capacités de détection d'échec notons encore: la détection de boucle, la mémorisation de certaines causes d'échec.

Le test de préunification (cette idée ayant déjà été exploitée par D. Warren [WAR 77] en vue de la compilation de PROLOG) distingue deux types d'échec. Là encore les méta-règles sont utilisées pour déterminer le point de reprise, ce en tenant compte du type de l'échec; si aucune méta-règle n'est applicable, la stratégie par défaut sera celle utilisée par PROLOG.
On trouvera dans [PER 79] une discussion sur le retour "intelligent" non guidé par les méta-règles et dans [BEK 83] une discussion sur la comparaison des deux méthodes.

Pour notre part nous retiendrons deux caractéristiques du langage METALOG:
- La séparation des méta-règles de contrôle.
- La détection d'échec par un test de préunification.

II.2.3. PARLOG

PARLOG [CLA 81], [CLA 83] est un langage de programmation logique quasiment semblable à PROLOG de part son aspect déclaratif, ainsi que vis à vis de ce qu'il calcule. Par contre son évaluation est guidée par une exécution parallèle exprimée par le programmeur, sous une forme dérivée de celle vue dans IC-PROLOG. Il utilise également la notion de "clause gardée" [DIJ 76], (notion que nous détaillerons au paragraphe suivant) qui est imbriquée dans l'expression du parallélisme.

En PARLOG les prédicats sont séparés en deux classes appelées "et-relation" et "ou-relation" permettant d'exploiter respectivement le "ou-parallélisme" et le "et-parallélisme" sous-jacents:

- ET-RELATION: une séquence d'appels de prédicats définis comme des "et-relations" sera évaluée en parallèle. Les variables communes agissent alors comme des canaux de communication, producteurs et consommateurs étant spécifiés suivant les conventions de IC-PROLOG. Pour cette classe de prédicats, une seule solution sera retournée, le mécanisme de "clause gardée" supprimant tous les choix possibles en cas de retour arrière.

- OU-RELATION: contrairement à la démarche précédente, l'évaluation d'une séquence d'appels de ce type de prédicats se fera séquentiellement et toutes les solutions seront recherchées. Par contre l'exploration des différentes branches liées à un appel se fera en parallèle.

L'interface entre les deux types de relation se fait à l'aide d'un "constructeur d'ensemble", manipulant les solutions retournées par les différents processus. De ce langage nous exploiterons essentiellement les idées:
- de construction d'un ensemble de solutions.
- d'exploitation parallèle de diverses branches.

II.2.4. Concurrent PROLOG

"Concurrent PROLOG" est un langage où sont généralisés les mécanismes vus dans PARLOG. L'expression du contrôle est assurée par:

- La possibilté de spécifier qu'une variable n'est accessible qu'en lecture (notée "?").
- L'utilisation de clauses gardées comme unité de programmation (éventuellement la garde peut être vide).

Un programme écrit en Concurrent PROLOG est un ensemble de clauses de la forme:

A -> G1 ... Gm | B1 ... Bn

où les Gi constituent la garde, et les Bj le corps de la clause, la séparation entre les deux étant indiquée par "|".

L'évaluation d'une requête s'effectue en parallèle; en voici une description informelle, des précisions pouvant être trouvées dans [SHA 83], [SHA 84] (dans le cadre du projet, [DAN 85] a réalisé un "mini-interpréteur" écrit en C-PROLOG).

La vérification d'un but s'appelle **réduction** et s'effectue par l'intermédiaire d'un **processus** . Un **système** est l'ensemble des processus créés pour la vérification d'un corps ou d'une garde de clause. Le système initial est composé d'autant de processus qu'il existe de buts dans la requête.

L'interpréteur va essayer de réduire en parallèle (et-parallélisme) chaque processus d'un système. Pour chaque processus il va créer autant de sous-systèmes qu'il existe de clauses dont l'en-tête s'unifie avec le but lié à ce processus (ou-parallélisme). Ces sous-systèmes ont pour objectif de réduire les gardes des clauses unifiées. Cette interprétation donne naissance à une hiérarchie de systèmes, communiquant par le protocole suivant:

- Tous les sous-systèmes descendants d'un processus n'ont accès qu'aux variables de ce processus, et ce uniquement en lecture. Toutes les instanciations à effectuer sur des variables du processus le seront sur des variables locales. Si un processus contient une variable spécifiée en lecture seule, non instanciée, il est suspendu jusqu'à l'instanciation de cette variable.

- Si un des sous-systèmes est réduit à vide, alors les variables locales sont unifiées aux variables globales correspondantes. Si l'unification réussit, le processus père du sous-système est réduit aux processus liés au corps de la clause dont la garde vient d'être unifiée. Tous les autres sous-systèmes activés par ce père sont abandonnés (généralisation du "cut" appelé "commitment").

Dans un environnement où le parallélisme est réellement mis en oeuvre, il suffit d'assurer une mutuelle exclusion des différents systèmes tentant d'unifier leurs variables locales aux variables globales de leur processus père. Les variables servent alors de canaux de communication et de moyens de synchronisation par le biais de l'annotation "en lecture seule".

Notons encore que si une variable spécifiée en "lecture seule", est unifiée avec un terme comportant des variables, celles-ci ne sont pas en "lecture seule". Ceci permet de mettre en oeuvre un mécanisme utilisé, pour l'heure, que dans des techniques de programmation orientée objet: la création de messages partiellement déterminés.

Nous retiendrons que Concurrent PROLOG est une implantation du langage tournée vers l'exploitation du parallélisme à tous les niveaux ("et-parallélisme", "ou-parallélisme"). Il en résulte que seul un matériel adapté pourra satisfaire aux besoins de son interprétation. Choisi par les Japonais comme langage de leur machine de cinquième génération, il sera intéressant de voir quelles solutions matérielles ils proposeront.

II.2.5. Application aux BC

Les mécanismes mis à jour par les différents travaux exposés ci-dessus peuvent devenir ceux sous-jacents à l'élaboration d'un SGBC. En effet on distinguera deux phases dans l'évaluation d'une requête émise vers le SGBC, où ils seront utilisés avec profit:

- La phase d'optimisation de la requête: on pense plus précisément aux possibilités de réordonner les buts, l'évaluation des coûts, etc...

- La phase d'évaluation proprement dite: utilisation des différents parallélismes, tant par la construction d'ensembles de solutions que par la création d'un flux de données à travers les canaux de communications offrant une autre forme de parallélisme: le pipeline.

Nous verrons au paragraphe trois du chapitre suivant comment nous exploiterons le deuxième point, délaissant pour l'heure le premier. Mais nous allons tout d'abord exposer l'interpréteur séquentiel classique que nous avons implanté, outil que nous utiliserons comme support de développement pour les mécanismes présentés par la suite.

II.3. YAAP

La première étape du projet OPALE consiste en la spécification d'un algorithme d'unification permettant de filtrer les données issues du disque. Dès lors la réalisation d'un interpréteur PROLOG s'avère une étape préliminaire indispensable pour fournir le support de la validation de cet algorithme, mise en oeuvre dont nous allons préciser quelques aspects.

Cette mise en oeuvre repose sur un mécanisme de traduction du texte source en une représentation interne manipulée par l'interpréteur. L'algorithme de ce dernier est basé sur la description fournie en II.1.3. Pour préciser l'implantation nous allons détailler les structures de données utilisées que nous séparons en deux catégories:

- Les espaces statiques.

- Les espaces dynamiques.

II.3.1. Espaces statiques

Le rôle de la traduction est d'analyser le texte source pour générer la représentation interne du programme. Ce code est stocké dans trois espaces statiques différents. Précisons que le qualificatif "statique" ne doit pas être pris en son sens strict, de par la possibilité d'ajouter ou de supprimer des clauses en cours d'interprétation. On l'utilise néanoins par oppposition aux espaces beaucoup plus "dynamiques", utilisés par la résolution.

II.3.1.1. Dictionnaire

Chaque symbole du programme est codé par l'intermédiaire d'un dictionnaire. Un symbole est décrit à l'aide de trois informations:

- La première fournit le type du symbole:
  - atome,
  - prédicat défini par l'utilisateur et en ce cas l'adresse du début du paquet de clauses le définissant,
  - prédicat évaluable et en ce cas le code de ce prédicat,
  - variable globale.

- La deuxième est un pointeur vers un modèle de la structure du terme, stocké dans l'espace des clauses (éventuellement nul si le terme n'a pas de structure).
- La dernière est un pointeur sur le nom externe dans l'espace des chaînes.

Le nom interne du symbole est fourni par l'adresse, dans le dictionnaire, de l'item le représentant.

II.3.1.2. Table des clauses

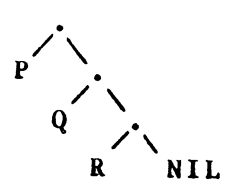
La table des clauses contient d'une part les clauses du programme, sous leurs formes compilées et d'autre part les modèles des termes complexes. Un paquet de clauses définissant un prédicat est stocké sous forme d'une liste chaînée, chaque élément représentant une alternative. On accède à la tête de cette liste par l'intermédiaire du dictionnaire.

Une clause, ainsi que les modèles des termes, sont représentés sous forme d'une liste à l'aide du constructeur de liste ".", considéré comme un prédicat à deux arguments.

Exemple:

La clause: P -> Q R ;

est codée ainsi:



II.3.1.3. Espace des chaînes

L'espace des chaînes est un tableau de caractères où l'on stocke tous les noms externes et toutes les chaînes de caractères issues du programme source.

On trouve dans l'annexe 2 un exemple de programme codé dans la représentation explicitée ci-dessus.

II.3.2. Espaces dynamiques

Rappelons que l'interprétation consiste à parcourir l'arbre de résolution implicitement défini par le programme et dont nous avons vu les grandes lignes en II.1.3. La construction de l'arbre utilise trois espaces:

- Un pour la construction de l'arbre lui-même: la pile de résolution.
- Un autre pour les contextes associés, par le biais de l'unification, à chacun des noeuds: la pile des instances.
- Un dernier pour mémoriser les instanciations non locales: la trace.

II.3.2.1. Pile de résolution

On peut voir un item de la pile de résolution comme la matérialisation d'un noeud de l'arbre. Trois champs permettent d'indiquer pour un noeud:
- Son identification: un pointeur vers la table des clauses sur son nom interne.
- Un de ses fils: un pointeur vers la table des clauses sur une des têtes de règle définissant le prédicat. A partir de la première alternative, obtenue par l'intermédiaire du dictionnaire, on accède à toutes les autres grâce à leur chainage. Ce pointeur est éventuellement nul, soit que le noeud est une feuille, soit que toutes les aternatives aient été prises en compte.
- Son père: pointe vers un autre item de la pile de résolution et fournit ainsi les buts laissés en attente (nul pour le noeud racine).

Deux autres pointeurs achèvent la définition d'un noeud:
- L'un pointe vers la pile des instances, et fournit le contexte associé à la clause dont fait partie le but.
- L'autre pointe vers la trace, et mémorise d'éventuelles instanciations non locales.

Ceci étant l'avancement dans l'arbre consiste à créer une nouvelle entrée dans la pile, correspondant au nouveau noeud; le

retour arrière se fait alors en dépilant le noeud de sommet de pile ce qui assure que le nouveau sommet permet d'atteindre le premier point de choix laissé en attente.

II.3.2.2. Pile des instances

La pile des instances mémorise les éléments de contexte associés à chaque noeud. Un nouvel élément de contexte est créé lors de la tentative d'unification entre un but et une alternative. Il est associé au noeud représentant cette alternative.
Un élément de contexte est un ensemble de doublets. Le nombre de doublets est égal au nombre de variables de l'alternative. Ces doublets permettent de représenter les substitutions par une technique dite de "structure partagée". Pour ce faire un doublet est constitué de deux pointeurs:

- Un pointant vers la table des clauses. Si la variable est liée ce pointeur référence le terme instanciant la variable, sinon il est nul.
- Un pointant sur l'élément de contexte du terme référencé par le précédent pointeur. Il permet, le cas échéant, d'accéder aux instanciations des variables apparaissant dans ce terme, sinon il est nul.

Dans le cas où deux variables sont liées entre elles, la plus récente pointe sur la plus ancienne. Cette convention permet, de diminuer le nombre d'instanciations non locales.

Lors d'un retour arrière, la restitution du contexte consiste alors en:

- Dépiler l'élément de contexte du noeud courant.
- Eventuellement annuler les instanciations non locales, mémorisées à l'aide de la trace.

II.3.2.3. La trace

La trace est un ensemble de pointeurs, gérés en pile, adressant la pile des instances. Le pointeur trace de chaque nouveau noeud crée est affecté à la valeur du sommet de pile. Lors de l'unification entre l'alternative et le but courant toutes les instanciations non locales sont mémorisées.

Une instanciation est non locale si elle ne porte pas sur une variable de l'alternative, mais sur une variable appartenant à un but ancêtre.

Lors du retour arrière l'annulation des instanciations non locales consiste à remettre à zéro les doublets adressés par les item de la trace compris entre le sommet de la trace et le pointeur trace du noeud courant (si sommet et pointeur ont même valeur cela signifie qu'aucune instanciation non locale a été effectuée).

II.3.3. Contexte BC

Les limitations amenées par cette représentation sont liées à la taille des espaces utilisés. En effet il faut pouvoir charger tout le programme dans la table des clauses, ce par l'intermédiaire du dictionnaire. Dans un contexte BC cette hypothèse n'est pas réaliste. Pour supprimer ces limites les solutions ont été:

- De réaliser une pagination du dictionnaire permettant ainsi de mettre en oeuvre un dictionnaire virtuel stocké sur disque.
- De réserver un espace tampon dans la table des clauses permettant d'ajouter et de supprimer les connaissances nécessaires à la construction de la résolution.

On trouve des précisions sur l'implantation dans [DAN 84], on détaillera le fonctionnement de ces mécanismes au chapitre VI.

Notons que les possibilités des interpréteurs classiques, quant à la manipulation de clauses "externes" sont trés restreintes, seul D-PROLOG [DON 84] offrant certaines facilités un peu plus sophistiquées que le simple ajout.

II.4. Conclusions

Le langage PROLOG a permis de "banaliser" une nouvelle forme de programmation, jusqu'ici réservée aux domaine et applications de "l'intelligence artificielle". Deux de ces principaux mécanismes, l'unification et le raisonnement par inférence, sont en passe de se substituer aux unités de base de la programmation classique tels, l'affectation, les tests ou autres sauts.

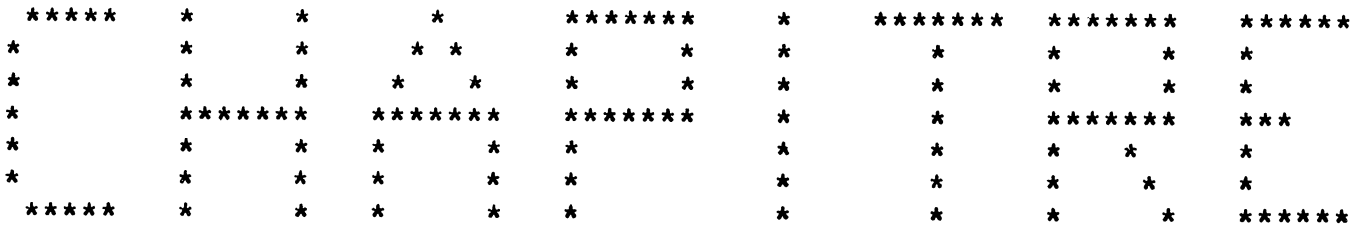
Si la puissance de ces mécanismes n'est plus à démontrer, leur efficacité dépend de leur implantation, ce particulièrement vis à vis des inférences. Gérer et guider les inférences sont les deux problèmes à résoudre pour obtenir un outil efficace dans le cadre d'applications soit complexes, soit importantes. Sur ce plan, la stratégie séquentielle "depth-first" offerte par PROLOG ne saurait suffire.

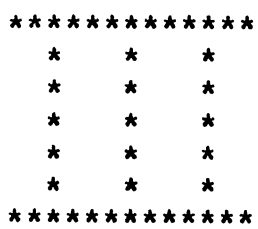
Nous avons donné un rapide aperçu des différents travaux portant sur ces problèmes et ayant donné le jour à des langages "cousins" de PROLOG. Les différentes idées proposent de:

- Guider les inférences: emploi de méta-règles, annotation des variables en "lecture seule" ou en "producteur-consommateur", utilisation des clauses gardées.

- Gérer les inférences: mise en oeuvre de parallélisme, "ou-parallélisme" et "et-parallélisme".

Dans notre contexte BC, on utilisera avec profit les premiers mécanismes, ce pour la phase d'optimisation de la requête, car elle ne demande qu'une seule solution: un des ordonnancements favorables de la requête.
Par contre la phase d'évaluation nécessite généralement la recherche de toutes les solutions. Aussi nous nous tournerons, pour cette phase, plutôt vers la mise en oeuvre des différents parallélismes, avec pour objectif final de les reporter sur le matériel.





SELECTION DES CONNAISSANCES

CHAPITRE III

RESUME

Ce chapitre offre tout d'abord un aperçu des techniques utilisées pour resteindre le domaine des recherches dans une BD: optimisation des requêtes, utilisation des chemins d'accès, avant de s'intéresser à l'évaluation elle-même. Nous verrons alors ce qu'apporte la notion de filtrage en terme de sélection, ses différences et ressemblances vis à vis de l'unification, puis les spécificités de l'unification dues au contexte BC. Pour finir, nous répertorions les outils développés pour supporter le filtrage, et plus spécialement les automates.

Chapitre III

SELECTION DES CONNNAISSANCES

Nous venons de voir que PROLOG offre des mécanismes puissants quant à l'exploitation des connaissances. Malheureusement leur efficacité diminue en raison inverse du volume des connaissances manipulées. Dans un contexte BC il est hors de question de traiter toutes les informations, ce pour répondre à une requête n'en nécessitant qu'une infime partie. On appelle cette réduction du champ d'investigations des recherches, la phase de sélection des connaissances, phase qui ne doit pas pour autant écarter d'informations utiles à la suite de l'évaluation de la requête. Si on ne fait état que de processus de recherche, c'est que toutes les opérations nécessitent une recherche, phase qui s'avère la plus longue quelque soit l'opération effectuée.

Si dans les SGBD conventionnels cette étape est déterminante quant aux performances, son impact sera encore plus grand pour un SGBC, ceci en raison des mécanismes plus complexes, donc plus coûteux, opérant sur les connaissances sélectionnées. C'est pourquoi nous considérons cette étape comme un des points clés de la réalisation d'un SGBC. Nous verrons que nous proposons même d'aller plus loin en effectuant, en même temps que la sélection, un premier traitement.

La définition de la stratégie de recherche se fera alors en fonction des considérations émises, d'une part au chapitre précédent, et d'autre part ci-dessus. Nous verrons comment adapter au mieux stratégie et "sélection pré-traitement" en fonction l'une de l'autre, adaptation devant permettre un gain notable de performances.

III.1. Sélection BD

Dans une BD classique l'évaluation d'une requête se fait en deux phases:

- Une phase d'optimisation de la requête et de recherche des chemins d'accès.
- Une phase d'évaluation proprement dite

Nous allons rapidement voir en quoi consiste la première étape, ce qui nous amènera à parler de l'organisation physique des données et permettra ainsi d'aboutir à un mécanisme mis en oeuvre par certaines Machines Bases de Données: le filtrage.

III.1.1. Optimisation et chemins d'accès

Dans une BD relationnelle l'optimisation des requêtes consiste à réordonner les éléments d'une requête, en fonction des paramètres suivants (leur ordre d'exposé n'étant pas significatif de leur importance):

- Exécution le plus tôt possible des opérations de sélection.

  Exemple:

  on évaluera : Père(Jean,X) Père(X,Y)
  plutôt que : Père(X,Y) Père(Jean,X)

- Evaluation de la cardinalité des réponses (l'évaluation s'effectue généralement en fonction de la cardinalité de la relation et de la sélectivité de la requête).

  Exemple:

  . si R1(a,X) est supposé retourner n1 < 10 solutions
  . si R2(b,X) est supposé retourner n2 < 100 solutions
  on évaluera : R1(a,X) R2(b,X)
  plutôt que : R2(b,X) R1(a,X)

- Prise en compte de l'organisation physique des données, plus précisément utilisation des chemins d'accès.

  Exemple: S'il existe un index sur le premier constituant de

R1(X,Y) mais pas sur la relation R2(X,Y) alors
on évaluera : R1(a,X) R2(b,X)
plutôt que : R2(b,X) R1(a,X)

Nous n'entrerons pas dans les détails d'une phase qui peut s'avérer fort complexe, et qui reste le sujet de nombreuses recherches [DELb 82]. Nous voulons seulement montrer, au travers de ces exemples triviaux, que cette étape a pour but une première et importante réduction du domaine d'évaluation des requêtes, soit en exploitant leur sélectivité, soit en utilisant les chemins d'accès disponibles.

Nous allons maintenant préciser ce que l'on entend par chemin d'accès, notion directement lié à l'organisation physique des données.

III.1.2. Chemin d'accès - Organisation physique

Si d'un point de vue logique une relation peut être vue comme un tableau, son implantation physique n'est pas aussi simple. Les n-uplets d'une relation sont stockés dans des fichiers découpés en page, la page étant l'unité physique d'entrée-sortie. On distingue trois types d'organisation physique:

- La plus triviale consiste à stocker les n-uplets "en vrac", c'est à dire sans ordre ni relation entre eux. Cette méthode n'offre aucun moyen intrinsèque de récupérer un n-uplet, dont on connaît la valeur de l'un de ses constituants, autre qu'un balayage séquentiel de toutes les pages du fichier.

- Une idée pour atteindre plus directement un n-uplet, toujours en fonction de la valeur de l'un de ses constituants, est de calculer un h-code à l'aide d'une fonction de "hachage". Ce h-code fournit l'adresse d'un bloc, constitué d'une ou de plusieurs pages, dans lequel sera stocké le n-uplet. A l'intérieur d'un bloc la recherche est séquentielle. De nombreuses variantes de cette technique sont encore du domaine de la recherche notamment en ce qui concerne la définition de la fonction de "hachage".

- Enfin une dernière possibilité est de trier suivant une clé les enregistrements du fichier. Ce tri effectué, on crée alors un fichier d'index groupant les valeurs des clés de tous les n-uplets stockés en début de page. Plus précisément un enregistrement du fichier d'index est un couple composé de:
  . la valeur de la clé du premier enregistrement de chaque page.
  . l'adresse physique de la page.
  Ce fichier d'index peut à son tour être vu comme une relation triée sur le constituant clé. Dès lors on peut itérer le processus en créant un index sur cet index. On aboutit ainsi à ce que l'on appelle un "B-arbre" qui matérialise cette hiérarchie d'index.

Ceci dit, ces organisations physiques ne peuvent s'appliquer que sur un (ou un ensemble de) constituant de la relation, ce au détriment des autres. Dans le cas de l'indexation on qualifie l'index de "primaire". Pour un accès associatif sur d'autres constituants il est nécessaire de créer des index "secondaires". A l'inverse des index primaires ne contenant que les valeurs des n-uplets de début de page (index dit "creux"), les index secondaires nécessitent de référencer tous les n-uplets de la relation, aucun ordre physique ne pouvant être exploité. Par contre on pourra créer une hiérarchie d'index primaires sur tout fichier d'index secondaire. Des organisations vues ci-dessus dérivent les chemins d'accès permettant d'obtenir les pages à consulter pour satisfaire la requête.

Un paramètre important dans l'organisation en index est la taille des pages physiques. De leur dimension va dépendre la taille des index primaires, voire de leur hiérarchisation en "B-arbre". En effet les tailles des pages et index varient en sens opposé l'un de l'autre. Du compromis adopté résultera les bonnes ou mauvaises performances du système:

- Si les pages sont trop petites:
  . soit les index ne sont pas hiérarchisés et leur taille réduit d'autant leur efficacité.
  . soit les index sont hiérarchisés, mais le nombre de niveaux risquent de dégrader les performances, chaque accès à un niveau nécessitant en général un accès disque.

- si les pages sont trop grandes:
  . l'indexation perd de son efficacité car un traitement important devra être effectué au sein d'une page.
  . le temps de transfert d'une page pénalise également les temps de réponse, aucun traitement n'étant effectué en parallèle.

Dans la pratique la taille des pages va également être liée à des paramètres physiques tels la capacité d'un secteur, d'une piste etc ... Pour certaines machines Bases de Données, la taille des pages sera aussi fonction de l'existence ou non d'un mécanisme de filtrage (que nous définirons ci-aprés), permettant de réduire le coût de traitement à effectuer en mémoire centrale, au sein d'une page.

On trouve dans [DELc 82] une discussion complète sur les coûts des modifications et recherches à l'aide de ces différentes organisations. Pour la suite on retiendra qu'elles ont pour objectif:

- De sélectionner les pages physiques contenant des informations utiles à la requête en fonction des paramètres connus et ainsi de réduire le nombre de pages transférées en mémoire centrale, opérations très coûteuses.

- De réduire les temps de traitement puisque seules les pages utiles seront consultées.

III.1.3. Situation du filtrage

Une des pertes de performance provient de l'indivisibilité du transfert d'une page, ce qui entraîne qu'aucun traitement n'est effectué sur les premières données lues, tant que toute la page n'est pas transférée. L'idée du filtrage est précisément de pallier en partie la lenteur des unités de disque en effectuant "au vol", c'est à dire pendant le transfert et sans ralentir le débit de l'unité de disque, une ultime sélection, voire un premier traitement. A l'inverse des techniques d'indexation entièrement logicielles, les mécanismes du filtrage nécessitent un support matériel, s'intercalant entre la mémoire primaire et secondaire.

Généralement le filtrage est un processus tout à fait

élémentaire qui repose sur la comparaison des données lues avec un ou plusieurs "modèles". Les modèles proviennent des paramètres connus de la requête, soit directement, soit par une première évaluation, et concernent un ou plusieurs constituants. Dès lors le filtre élimine les données ne pouvant être mises en corespondance avec l'un des modèles.

Exemple:

Soient les modèles

R1(a,_)
R1(b,_)
R1(c,_)

où "_" indique que le constituant n'est pas connu.

alors: R1(a,a) sera accepté.

et : R1(d,d) sera rejeté.

Plusieurs projets ont tenté d'intégrer ces outils, connus sous le nom de filtre, au sein des machines bases de données [BAN 80], [BAN 81], [GAR 80]. D'autres travaux [RHO 80], d'un aspect plus théorique, proposent outre des solutions de mise en oeuvre, des techniques plus sophistiquées permettant des traitements numériques. Nous verrons au paragraphe III.3 les similitudes entre le filtrage et l'unification, et discutons des gains espérés au paragraphe III.4. après avoir vu différents outils.

III.2. Stratégie de recherche

Nous avons vu au chapitre II les faiblesses de la stratégie "depth-first", les solutions proposées et les conclusions quant à notre contexte se résument ainsi:

- Utilisation de mécanismes de contrôle pour la phase d'optimisation (et éventuellement en cours d'évaluation pour l'interprétation des règles issues de la BC).

- Introduction de parallélisme et de techniques de filtrage lors de l'évaluation, principalement pour les faits élémentaires.

Nous ne nous intéresserons qu'au deuxième problème, le premier étant intimement lié à des questions d'organisations physiques, et pourra se résoudre, dans un premier temps, à l'aide des solutions mises en oeuvre dans les BD.

La stratégie de recherche envisagée est en cours d'étude. On trouve dans [BER 84] et [BER 85a] les idées actuellement en développement et en présentons ici les grandes lignes en nous attachant plus particulièrement aux particularités liées à l'exploitation du filtrage.

III.2.1. Evaluation ensembliste

L'idée de base est de transformer l'évaluation séquentielle de PROLOG (solution par solution) en une évaluation ensembliste (toutes les solutions à la fois). Deux considérations justifient cette idée:

- La lenteur relative des unités de disque nécessite que lors de chaque accès, un maximum d'informations utiles soit retiré du bloc consulté, pour éviter, dans la mesure du possible, plusieurs accès au même bloc lors de l'évaluation d'une même requête.

- Dans un contexte BC on utilisera les mécanismes de filtrage, pour réaliser l'unification, mécanismes d'autant plus rentables qu'ils opèrent sur un ensemble de modèles, point que nous justifierons par la suite.

Notons que les SGBD utilisent avec profit cette recherche simultanée de toutes les solutions. De plus, cette stratégie supprime, dans l'interprétation de PROLOG, le mécanisme de retour arrière ("back-tracking"), toujours lourd à mettre en oeuvre. Par contre elle n'autorise plus le contrôle sur l'évaluation ("cut"), mais dans un contexte BC ce ne devrait pas être trop pénalisant, les recherches étant dans la plupart des cas exhaustives.

Voici la description informelle du déroulement de la résolution, des précisions se trouvant dans [BER 85b]. A partir de la requête initiale, un "ou-processus" crée un pipe-line comportant autant de stations qu'il y a de littéraux dans cette requête. Les vérifications de ces littéraux seront assurées par des "et-processus" acceptant pour entrée les environnements, ensemble de substitutions représentées sous la forme de couple "variable-valeur", du littéral précédent. Les "et-processus" activent alors des "processus de recherche" qui génèrent les buts correspondants, en appliquant les substitutions, et lancent leur vérification. Celle-ci consiste en la sélection de tous les en-têtes de clause s'unifiant avec cet ensemble de buts, sélection effectuée par un mécanisme de filtrage dont la définition et mise en oeuvre seront développées au cours de l'étude. Pour chaque clause lue sur disque, le processus de filtrage aboutit à l'une des issues suivantes:

- Aucune unification n'est possible; c'est un échec, le traitement de la clause lue est terminé en l'abandonnant.

- Un ou plusieurs buts sont unifiés; pour chacun les actions suivantes sont entreprises:

  . soit la clause traitée est une assertion. En ce cas la vérification est terminée et le processus de recherche retourne à son père les substitutions générées, construisant ainsi l'ensemble de solutions qui sera traité par le littéral suivant.
  . soit la clause possède un corps. En ce cas le processus de recherche génère un "ou-processus" qui crée un nouveau pipe-line pour évaluer le corps de la clause. Les solutions retournées par le processus de recherche seront celles issues de la vérification du dernier littéral de ce pipe-line

(éventuellement un échec ou un succès sans génération de nouvelles substitutions).

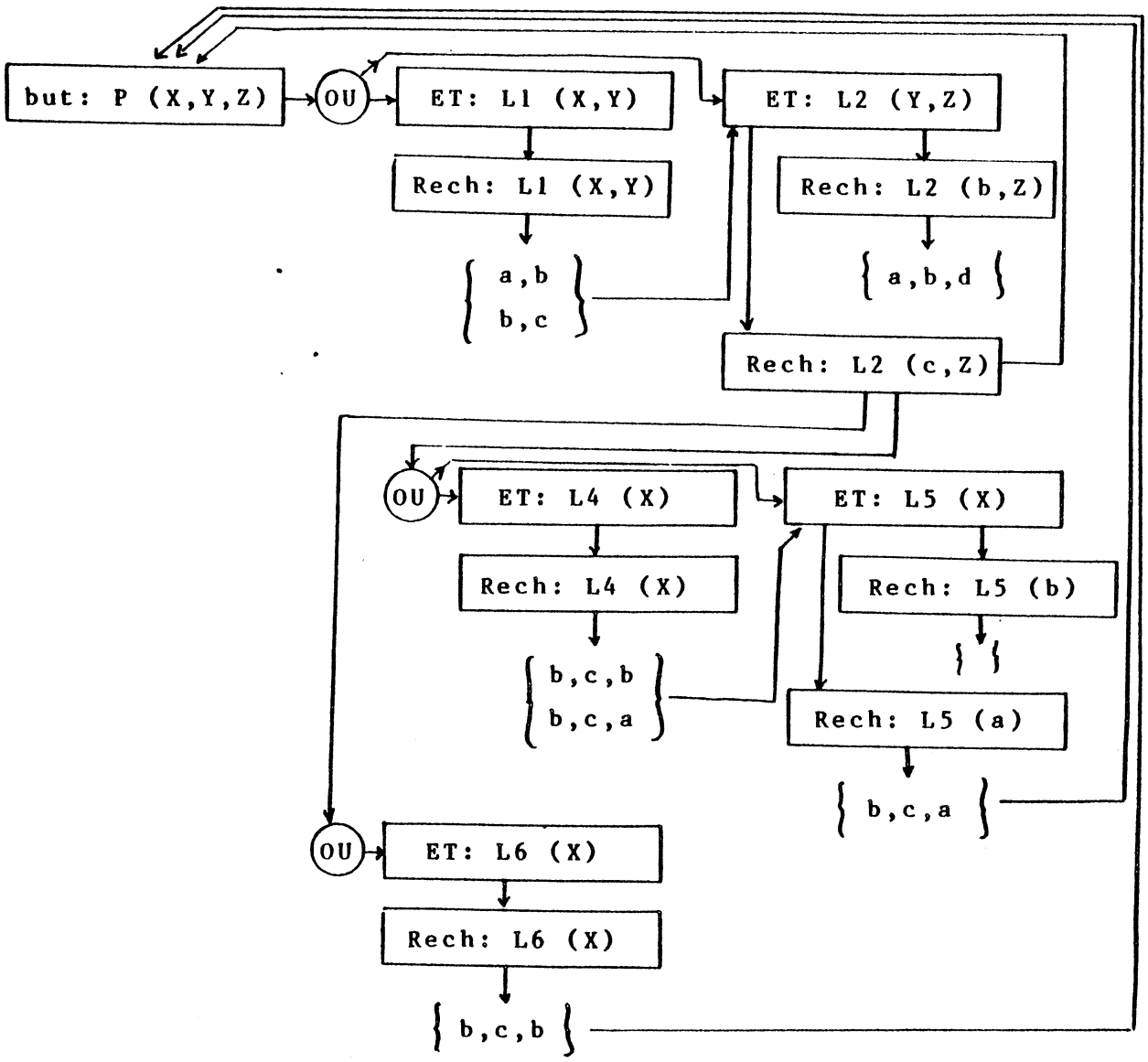
Nous reprenons ci-dessous un exemple exposé par Gilles Berger-Sabbatel dans [BERb 85] (chapitre VIII). Soit le programme:

```
P (X,Y,Z) -> L1 (X,Y) L2 (Y,Z) ;
L1 (a,b) -> ;
L1 (b,c) -> ;
L2 (c,X) -> L4 (X) L5 (X) ;
L2 (c,X) -> L6 (X) ;
L2 (b,d) -> ;
L4 (b) -> ;
L4 (a) -> ;
L5 (a) -> ;
L6 (b) -> ;
```

Considérons la question: P (X,Y,Z). Son évaluation entraînera le réseau de processus schématisé sur la figure III.1. Une telle mise en oeuvre est non seulement envisageable, mais qui plus est efficace, de par les spécificités du contexte BC:

- La faible complexité des règles associée à un nombre relativement peu élevé d'alternatives, évitera l'explosion combinatoire du nombre de pipe-lines créés, explosion qui ne manquerait pas de se produire dans tout autre contexte.

- L'organisation pipe-line permet d'introduire un parallélisme important d'une part dans la vérification d'une règle et d'autre part vis à vis de la vérification de plusieurs règles, les pipe-lines pouvant s'exécuter en total parallélisme. Ces mécanismes assurent de plus l'efficacité de la recherche par ensemble de solutions. En effet de par les accès disques les ensembles de solutions seront générés paquet par paquet, tout nouveau paquet pouvant être immédiatement traité par la station suivante, sans attendre la génération de l'ensemble des solutions. Qui plus est, il est possible que des vérifications issues de pipe-lines différents nécessitent les mêmes accès disques qui pourront alors être regroupés.



Après cette synthèse de la stratégie de recherche envisagée, nous allons fournir plus précisément les spécifications des mécanismes et accès aux disques supportant le parallélisme issu de la recherche d'un ensemble d'en-têtes de clauses.

III.2.2. Spécifications du filtre

La stratégie élaborée au paragraphe précédent repose sur un processus d'accès au disque réalisant l'unification des en-têtes de clauses lues sur disque non pas avec un, mais avec un ensemble de buts. Par la suite on appellera ce processus filtre ou unificateur, ce en fonction du contexte de la discussion.

Les spécifications du filtre d'un point de vue fonctionnel se résument comme suit. En fonction d'un ensemble de buts, fournis

sous forme de termes et repérés par des indices, le filtre retourne pour chaque en-tête de clause unifiée:

- Le ou les indices des buts unifiables.
- Les substitutions générées, le cas échéant.
- La queue de clause, si elle existe.

Exemple:

Soit l'ensemble de buts

```
1:   R1 ( X , t(b,c) ) -> ;
2:   R1 ( t(i,j) , Y ) -> ;
3:   R1 ( t(X,Y) , Z ) -> ;
```

Soit la clause lue sur disque:

```
R1( t(u,v) , t(b,c)) -> R2(u,X)  R3(v,Y) ;
```

Le filtre retourne:

```
1:   X = t(u,v)
2:   X = u   ;   Y = v   ;   Z = t(b,c)

queue:  R2(u,X)  R3(v,Y) ;
```

Nous préciserons par la suite comment résoudre (et si la tâche incombe au filtre) les problèmes liés à l'apparition de plusieurs occurences de la même variable au sein d'un même littéral, que ce soit un but ou une en-tête de clause lue sur disque.

Rappelons qu'à ces spécifications fonctionnelles s'ajoute la contrainte temporelle de suivre la cadence de transfert de l'unité de disque. La suite propose les idées des solutions, et nous commencerons par fournir la base de ces travaux: le filtrage et sa mise en oeuvre par des automates d'états finis.

III.3. Filtrage et unification

A priori on est tenté d'affirmer que le filtrage tel qu'on l'a défini en fin du paragraphe III.1.3. est un mécanisme inclus dans celui de l'unification, en tant que sous-mécanisme. En effet on peut, à partir d'un algorithme de filtrage, construire progressivement des algorithmes de plus en plus sophistiqués aboutissant à l'unification. Notre idée est d'apdopter la démarche symétrique, et de dire que si une unification complète n'est pas nécessaire, il est intéressant de ne programmer que la partie filtrage satisfaisante. Ceci devrait permettre de simplifier l'opération et ainsi apporter des gains de performance appréciables.

Notre démarche résultera de la réponse à la question suivante: "A partir de l'algorithme complet d'unification, jusqu'où peut on le "dégrader", donc le simplifier et en réduire ainsi le coût, pour qu'il soit encore efficace en terme de sélection des connaissances?"

Pour pouvoir répondre, nous allons tout d'abord mettre en évidence les ressemblances du filtrage et de l'unification, puis les particularités de l'unification. Nous évaluerons ensuite les spécificités d'un contexte BC.

III.3.1. Inclusion

Rappelons brièvement que l'unification consiste à comparer deux arbres en substituant, si nécessaire, les variables de l'un par les sous-arbres corespondants de l'autre ou vice-versa (un algorithme est fourni au chapitre suivant). Cette succinte définition met en évidence l'inclusion du filtrage dans celui de l'unification en ceci que le filtrage se compose uniquement de la comparaison de deux données, sans structure donc à fortiori, sans variables.

III.3.2. Différences

La première différence est une conséquence de la représentation des informations. Dans un contexte BD, la structure des données est uniforme pour l'ensemble d'une relation. Par contre dans une BC, la structure des termes est fournie par le terme lui-même et varie de

l'un à l'autre. Dès lors l'algorithme d'unification doit être capable d'analyser la structure des termes unifiés, ce au fur et à mesure de l'avancement de l'unification, alors que l'algorithme de filtrage pourra se baser sur une structure figée.

Exemple:
Pour unifier: t(X,b) avec t(i(j,k),b)
l'algorithme d'unification doit être capable de reconnaître la fin du terme t(i,j) qui doit être substitué à X.

La deuxième différence porte sur la génération des substitutions. Plusieurs mécanismes sont nécessaires pour la mettre en oeuvre:

- Reconnaissance d'une variable.
- Substitution de toutes les occurences d'une variable par le terme correspondant, qui peut logiquement être implantée de deux façons:
  . soit substituer réellement toutes les occurences de la variable par le terme.
  . soit générer une substitution sous la forme d'un couple (variable,valeur) en vérifiant, si cette variable apparaît dans un autre couple de substitution, que les deux valeurs liées à cette variable sont unifiables entre elles.

Notons que quelque soit la solution retenue, la substitution d'une variable ne peut se faire en un temps linéaire, ce qui va à l'encontre des spécifications temporelles imposant de suivre la cadence de transfert de l'unité de disque.

Enfin une dernière différence se situe dans le fait que non seulement des variables peuvent apparaître dans les buts recherchés, mais également dans les clauses lues sur disque, notamment lorsque celles-ci sont des règles. Là aussi un mécanisme de génération de substitutions sera à mettre en oeuvre.

Les différences exposées ci-dessus semblent entraîner un tel surcroît de travail par rapport au filtrage initial, que l'on peut douter de la possibilité de les mettre en oeuvre. Nous allons voir que les spécificités du contexte BC permettent de dégrader l'unification en des mécanismes plus simples effectuant une

sélection encore efficace.

III.3.3. Unification et BC

L'échec d'une unification a pour source l'une des deux causes suivantes:

- Soit deux symboles (non variables) de même position, dans l'un et l'autre des deux termes, sont différents.
  Exemple:
  t(c,X)
  et t(a,b)

- Soit les substitutions effectuées sur les occurences d'une même variable ne sont pas cohérentes:
  Exemple:
  t(X,X) t(a,X,X)
  et t(a,b). et t(Y,Y,b)

Au vu des discussions précédentes, la sélectivité du filtrage dans les BD ne peut être due qu'à la première cause d'échec, la deuxième n'existant pas. De même on peut estimer que dans un contexte BC, la première cause sera la plus fréquente, voire la seule. Des mesures, reportées dans [BER 85], confirment cette hypothèse, même pour des programmes PROLOG plus généraux.

On peut arriver à la même conclusion en regard de considérations sémantiques; en effet une requête du genre R1(X,X) signifie que l'on recherche toutes les assertions où les deux arguments du prédicat R1 ont même valeur. On se convaincra aisément que ce cas, s'il existe, ne saurait être très fréquent ou ne saurait être le seul critère de sélection. Quant aux assertions de la BC de la forme R1(X,X), on peut les considérer comme quasiment "pathologiques".

En regard de ces considérations, notre idée est de définir une opération de "préunification", identique à l'unification si ce n'est qu'elle n'applique pas dynamiquement les substitutions générées, phase la plus coûteuse. Bien entendu la cohérence des substitutions générées n'est plus assurée, et il faudra recourir à

une deuxième étape pour achever l'unification en cas de succès de la préunication. Nous présenterons au chapitre suivant cette opération et verrons la sélection que l'on peut en attendre.

III.3.4. Gains

Certaines critiques s'adressant au filtrage s'appuient sur le fait que, vue la lenteur des unités de disque, le coût d'évaluation d'une requête est principalement fonction du nombre de pages à transférer, le temps de traitement par l'UC étant négligeable de par sa rapidité. Dès lors le gain obtenu par filtrage ne peut qu'être négligeable.

Si ces critiques sont déjà contestables pour les BD, elles deviennent totalement injustifiées dans notre contexte BC, comme le montrent les chiffres suivants.

Pour l'heure les "machines à inférences" (du moins dites telles) les plus rapides, quant à l'interprétation de PROLOG, atteignent environ 10 KLIPS, ce qui correspond à une inférence toutes les 100 microsecondes. En utilisant des unités de disques de performances moyennes, mais en optimisant les accès (déplacements sur les pistes voisines), le temps moyen d'accès et de transfert d'une piste de 16 Koctets est de l'ordre de 30 et 50 ms. En estimant que 16 Koctets représentent de 100 à 500 assertions, leur temps de traitement par l'UC sera compris entre 10 et 50 millisecondes. Si ce traitement, ou une partie, peut s'effectuer au vol, par filtrage, le gain obtenu n'est plus négligeable (surtout que sur des machines moins rapides, le temps d'une inférence peut facilement être multiplié d'un facteur 10).

Notons que nous ne confondons pas inférence et unification, cette opération représentant moins de 50% du temps d'une inférence logique. Mais dans notre contexte, il se trouve que le fait de supprimer une clause lue sur disque, donc par une tentative d'unification, supprime le déclenchement d'une inférence en mémoire centrale.

III.4. Les automates

Nous avons vu au paragraphe 1.3. de ce chapitre, que la technique de filtrage se base essentiellement sur la mise en correspondance de "modèles" avec les données à traiter. En fonction des outils disponibles, on peut complexifier de plus en plus les modèles, l'opération de filtrage en découlant pouvant alors évoluer d'une simple comparaison jusqu'à, pourquoi pas, la réalisation de l'unification.

Les outils les plus performants sont les automates. Ceux-ci peuvent être classés en plusieurs types, les plus sophistiqués aboutissant aux microprocesseurs ... Pour obtenir un rapport coût de mise en oeuvre/traitement effectué le plus bas possible, il est nécessaire d'ajuster au mieux possibilités et besoins. En effet il serait désastreux, d'un point de vue économique, de concevoir et d'utiliser un microprocesseur pour n'utiliser qu'une opération de comparaison.

III.4.1. Divers outils

Nous fournissons ici un rapide aperçu de différents outils, tous issus de l'opération de base du filtrage qu'est la comparaison:

- Comparateurs en parallèle: chaque modèle, réduit à une seule valeur, est stocké dans un des comparateurs. Chaque donnée issue du disque est comparée en parallèle aux modèles stockés dans les comparateurs, ce qui assure de travailler à la vitesse du disque. Par contre aucun développement de ce processus n'est envisageable.

- Mémoire associative: le principe est le même si ce n'est que les comparaisons sont effectuées sur des blocs de données de taille égale à celle de la mémoire associative. En fait on remplace un algorithme logiciel de recherche par un algorithme câblé [DEF 73], [HSI 76].

- Mémoire circulante: c'est une solution originale qui consiste à stocker la BD sur des mémoires circulantes; la recherche des modèles se fait lors des cycles de rotation. L'inconvénient est que pour répondre à une requête booléenne, tel le "ou" du à la recherche de plusieurs modèles, il est nécessaire d'effectuer plusieurs cycles en mémorisant le résultat issu de chacun [LIN 76], [SCH 78].

- Mémoire cache: il s'agit simplement de simuler l'effet d'un filtre en ayant toujours à disposition une partie de la BD. Cette solution est limitée par la taille de la BD.

Tous ces outils n'autorisent pas la recherche de modèles complexes, qui sont, par exemple, spécifiés en plusieurs champs. C'est pourquoi nous nous sommes tournés vers l'utilisation d'automates d'états finis, permettant des traitements plus sophistiqués, sans requérir des coûts de mise en oeuvre trop élevés.

III.4.2. Les automates d'états finis

La notion d'automate est utilisée dans de nombreux domaines, et on peut donc l'aborder de différentes façons. Nous en parlerons dans le contexte de la reconnaissance de langage. Nous ne nous intéresserons qu'aux automates dits "d'états finis" et renvoyons à [VAU 78] pour une présentation et classification de ces outils, précisant leurs définitions et capacités en termes de reconnaissance de différents types de grammaire.

III.4.2.1. Présentation théorique

La définition théorique d'un automate d'états finis le présente comme un quintuplet de la forme suivante:

$$A = (Vt , Q , q1 , F , m)$$

où Vt est un ensemble fini : le vocabulaire d'entrée.
Q est un ensemble fini : les états de l'automate.
q1 (appartenant à Q) : l'état initial.
F (ensemble inclus dans Q): les états terminaux.
m : une application de (Q X Vt) -> Q.

Le langage reconnu par cet automate est:

$$L(A) = \{ x / x \in Vt^* \text{ et } \hat{m}(q1,x) \in F \}$$

où $\hat{m}(q,\text{elt neutre}) = q \quad \forall q \in Q$

$\hat{m}(q,xa) = m(\hat{m}(q,x),a)$

$q \in Q, \quad x \in Vt^*, \quad a \in Vt$

Il est clair que la puissance d'un tel outil n'est pas du même ordre que celle des outils précédents. En effet on ne parle plus ici de reconnaître quelques modèles, mais tout un langage, dès lors que celui-ci est d'état fini. Ceci dit, son implantation reste suffisamment simple pour assurer un compromis coût/traitement satisfaisant, comme nous le verrons par la suite.
Nous allons tout d'abord voir la représentation logique que l'on utilisera par la suite.

III.4.2.2. Application au filtrage

Classiquement on représente un automate par un graphe où:
- Les noeuds représentent les états.
- Les arcs représentent les transitions, en d'autres termes matérialisent l'application 'm' (encore appelée table de transitions).

Suivant la définition, étant donné un vocabulaire d'entrée, pour chaque état on doit avoir autant d'arcs sortant que d'éléments dans le vocabulaire. Par convention, on ne représente pas sur le graphe les arcs aboutissant à un état d'échec, ce qui assure la lisibilité du graphe.

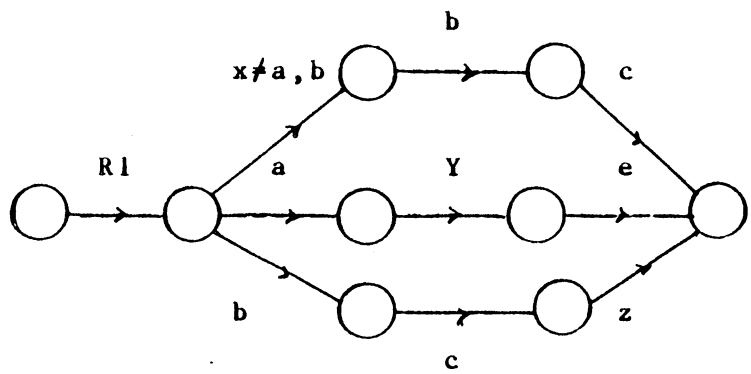
Pour une utilisation à des fins de filtrage, le vocabulaire d'entrée est constitué de toutes les données atomiques de la base (dans notre cas un code interne). Les mots à reconnaître sont les modèles issus de la requête où les différents constituants fournissent les transitions, et les structures de ces modèles établissent la succession des états de l'automate.

Exemple:

1: R1 (X,b,c)

2: R1 (a,Y,e)

3: R1 (b,c,Z)

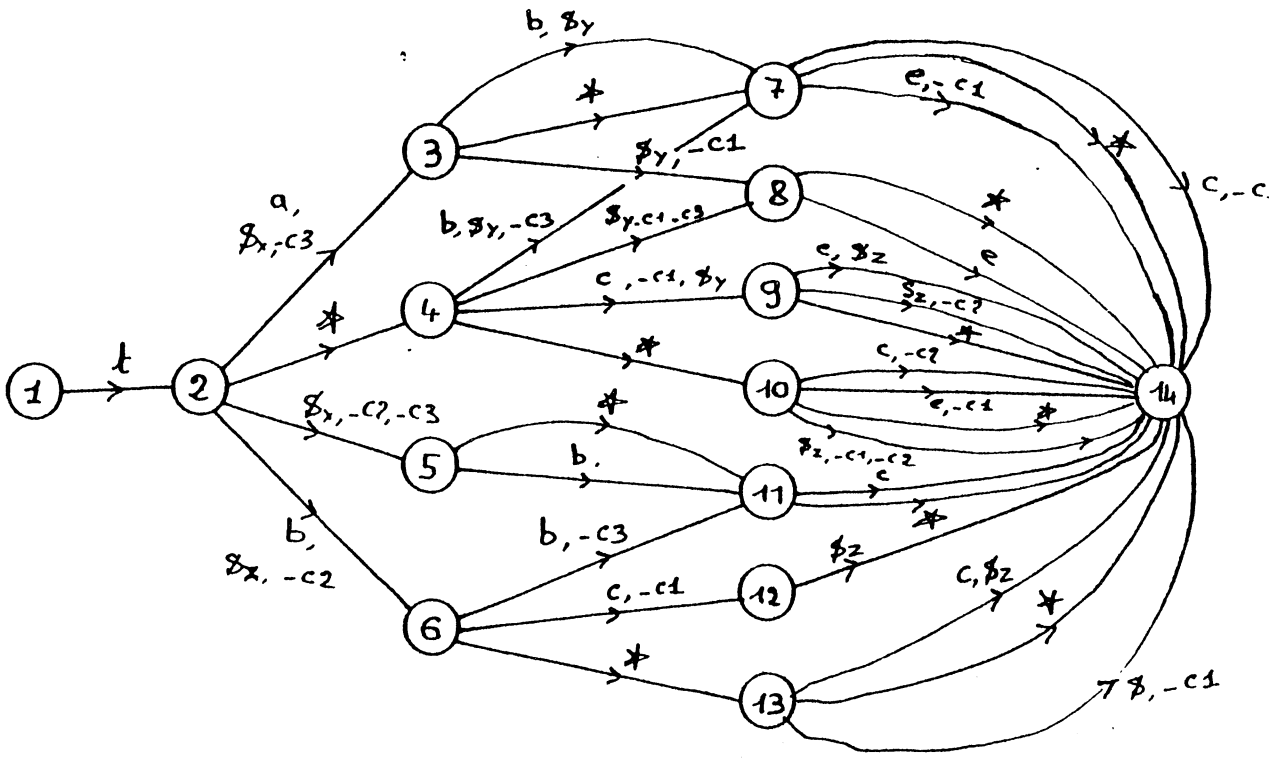


L'analyse d'une donnée par l'automate aboutit soit à un échec, soit à un succès sans autre renseignements supplémentaires.

III.4.3. Limites

Pour une BC, les mécanismes mis en oeuvre ci-dessus, ne peuvent être utilisés directement de par la possibilité de trouver des variables dans les données. En effet une "variable donnée" peut être substituée à n'importe quel terme des buts, sans éliminer aucun d'eux pour la suite de la reconnaissance. L'automate devient alors indéterministe et sa transformation en un automate déterministe, entraîne une explosion de sa taille comme le montre la figure III.1. L'exemple du paragraphe précédent nécessite maintenant 14 états et 36 transitions, au lieu de 10 états et de 14 transitions. Cette transformation est néamoins nécessaire pour que la reconnaissance reste une opération linéaire du point de vue du temps d'exécution.

De plus l'indication du ou des buts reconnus, ainsi que la génération des substitutions, sont deux mécanismes supplémentaires permettant d'assurer un traitement plus complet que la simple sélection. En effet sans ces améliorations, l'acceptation d'une clause entraîne la nécessité de l'unifier avec tous les buts



- "-C1" : signifie que le but i n'est plus reconnu.
- $id_var: spécifie la génération d'une substitution pour la variable id_var.
- "*" : une variable disque.

Figure III.2:

recherchés en parallèle. On conçoit la somme de travail redondant effectué car, dans la plupart des cas, seul un des buts sera unifiable, l'échec de l'unification étant le plus souvent dû à la discordance de deux symboles plutôt qu'à l'incohérence des substitutions. Or cette cause d'échec est déjà détectée par le filtre.

La mise en oeuvre de solutions répondant à ces problèmes constitue la base de l'étude. Avant d'entreprendre leur élaboration, nous allons faire le tour des différentes implantations des automates, source d'idées pour la résolution de nos difficultés.

III.4.4. Implantation

De nombreuses possibilités existent quant à l'implantation des automates chacune étant spécifique au contexte d'utilisation. Nous allons rapidement en faire le tour, nous étant inspiré de plusieurs d'entre elles. Pour un exposé plus complet le lecteur pourra consulter l'ouvrage référencé [RHO 80].

Question terminologie, un état est un ensemble de couples (T,S) où T est une "transition" reconnue par l'automate le faisant passer de l'état courant à l'état successeur S.

Le problème à résoudre est de déterminer, en fonction de la transition d'entrée courante, si elle est acceptée par l'automate, et en ce cas passer dans l'état successeur. Pour ce faire deux méthodes d'organisation, et donc d'accès, se distinguent.

La première consiste à considérer l'ensemble des couples (T,S) composant un état comme une liste:

- Liste non triée: la recherche de l'état successeur est triviale, puisqu'elle consiste en un parcours séquentiel de la liste en comparant chaque transition. L'efficacité est de toute évidence liée aux tailles des listes ...

- Liste triée: l'amélioration immédiate est de trier les listes en fonction des transitions acceptées. La recherche peut alors s'effectuer par dichotomie.

La deuxième repose sur l'indexation des entrées, permettant d'atteindre en un seul accès l'état successeur:

- Indexation simple: les entrées indexées fournissent l'adresse de l'état successeur. Cette méthode nécessite de réserver pour chaque état autant de successeurs que d'entrées possibles. Cette solution n'est envisageable que pour un ensemble d'entrée réduit.

- Représentation de BIRD: de même l'indexation des entrées fournit la position d'un bit dans la chaîne de bits de l'état courant. Si ce bit est à "1", on trouve l'adresse de l'état suivant en incrémentant l'adresse de l'état courant, du nombre de bits à "1"

entre ce bit et le début de la chaîne.

- La dernière amélioration consiste en des automates tabulés. On renvoie le lecteur à [RHO 80] pour les explications d'une mise en oeuvre sophistiquée.

Nous verrons dans la partie traitant de l'implantation matérielle, comment tirer partie de ces différentes solutions ce, suivant les spécificités de l'automate à construire.

III.5. Conclusions

La sélection joue un rôle déterminant pour les performances d'un SGBD, une première forme étant la restriction des champs de recherche. Une autre mise en oeuvre consiste à filtrer les données, lors de leur transfert de la mémoire secondaire vers la mémoire centrale, filtrage effectué en fonction d'un certain nombre de modèles. Cette possibilité n'est pas toujours exploitée et à ceci deux raisons:

- Elle nécessite un matériel spécialisé.

- Son efficacité n'est pas reconnue par tous.

Cette dernière objection tombe dans notre contexte, surtout si on essaie de mettre un peu plus "d'intelligence" dans le filtrage. En effet l'unification, opération de base du traitement des connaissances, repose essentiellement sur le même concept de mise en correspondance que celui utilisé dans le filtrage. Elle n'en est qu'une version améliorée, principalement par l'introduction de variables, permettant d'exprimer des modèles plus complexes.

Ainsi on envisage l'utilisation d'un matériel spécialisé, permettant d'une part une sélection efficace, et d'autre part la préparation de l'unification complète. Ce matériel déchargerait en partie l'UC du traitement coûteux de cette opération.

Parmi les outils développés pour la réalisation de filtre, nous retiendrons les automates d'états finis, même si leur utilisation n'est envisageable qu'au prix de modifications importantes.

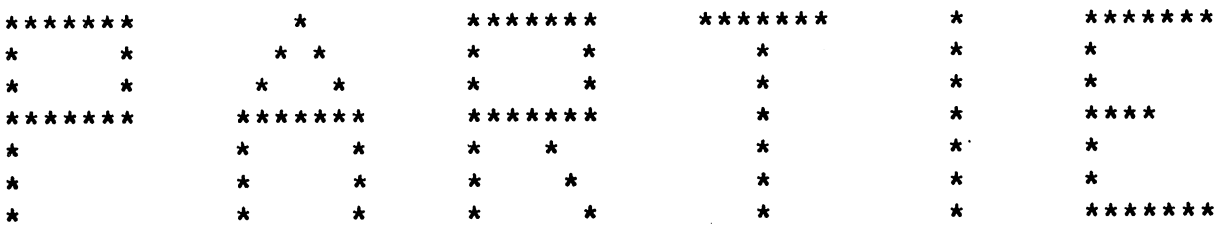
Ce chapitre termine la première partie, consacrée aux spécifications d'un opérateur d'unification pour une machine BC, au cours de laquelle nous avons exposé:

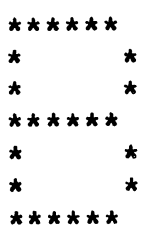
- L'utilisation de PROLOG dans un contexte BC et les spécificités de ce contexte.

- La stratégie de recherche qui exploite ces spécificités.

- Les spécifications du filtrage découlant de cette stratégie de recherche.

Ceci posé, il s'agit de déterminer les mécanismes capables de satisfaire ces spécifications, en conservant l'objectif final de leur implantation matérielle. Nous développerons tout d'abord les algorithmes sous leurs formes logicielles, avant d'envisager l'implantation matérielle de certains d'entre eux.





MAQUETTE LOGICIELLE

RESUME

La deuxième partie concerne le côté logiciel de l'étude. Elle apporte les idées des solutions sous formes d'algorithmes satisfaisant les spécificités établies lors de la première partie, ainsi que leurs validations par leurs implantations. Elle servira donc de base à la partie ayant trait à la réalisation matérielle. Elle se compose de trois chapitres.

Le premier (chapitre IV de la thèse) introduit la notion de préunification comme la composante de filtrage de l'algorithme d'unification, ce en regard des considérations émises sur le filtrage au chapitre III. Pour satisfaire les exigences de la stratégie de recherche, il est nécessaire d'unifier non pas un, mais un ensemble de buts en parallèle. Pour ce faire, nous proposons un codage des buts, puis un algorithme de préunification opérant sur ce codage. Cet algorithme assure, d'une part la détection d'un fort pourcentage des échecs de l'unification, et d'autre part, en cas de succès, la réduction du traitement nécessaire à l'unification. Sa définition tiendra compte des objectifs matériels en le décomposant en plusieurs primitives à la fois simples et relativement indépendantes.

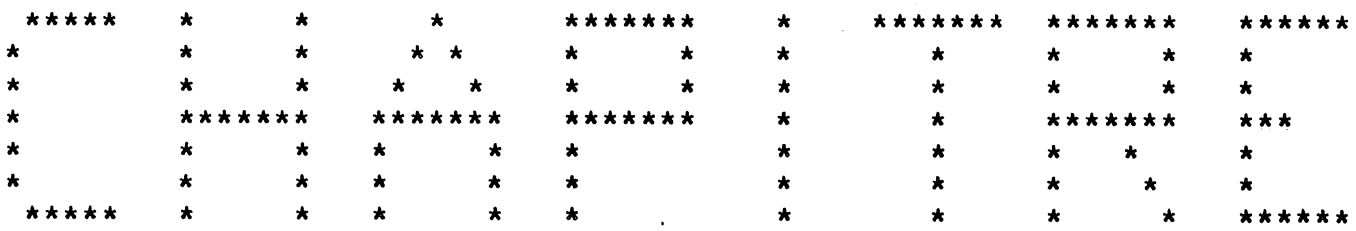
Le chapitre V fournit la validation des idées énoncées au chapitre précédent, par l'implantation des algorithmes. Toujours en regard de l'issue finale, ceux-ci doivent, en plus de répondre aux fonctionalités établies, prendre en compte des considérations matérielles que l'on exposera en début de ce chapitre. A la suite de quoi, nous détaillerons la mise en oeuvre des trois opérations principales:

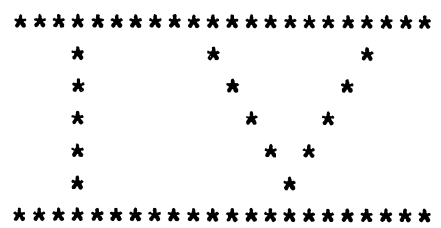
- La "compilation" des buts, assurant le codage des buts en la représentation définie au chapitre précédent.
- L'algorithme de préunification lui-même, décomposé en ces primitives.

- La phase d'association, terminant, si nécessaire, les unifications.

Pour chacune de ces phases, nous rappelons tout d'abord les spécifications, puis exposons l'algorithme avant de présenter des mesures quant à son temps d'exécution.

Le dernier chapitre (chapitre VI) de cette partie expose l'intégration d'une maquette logicielle du filtre, au sein de notre interpréteur classique. Cette interprétation se fait par le biais de deux prédicats évaluables, utilisant les mécanismes élaborés précédemment lors de l'accès aux fichiers de la Base. Une fois fournies les spécifications de ces deux prédicats, nous abordons le problème du à la coopération entre la stratégie séquentielle et parallèle de la maquette. Nous distinguons plusieurs points délicats liés, soit à la coopération, soit au contexte Bases de Connaissances, et proposons nos solutions. Les travaux se rapportant au contexte Bases de Connaissances fournissent une première approche des difficultés à résoudre pour aboutir à la machine finale. Nous discutons alors rapidement des performances, des mesures précises s'avérant délicates à effectuer. Pour finir, nous proposons une autre utilisation de l'algoritme de préunification, permettant d'effectuer la recherche d'un ou plusieurs éléments dans une liste, fonction de base pour bien des applications.





UNIFICATION PARALLELE

CHAPITRE IV

RESUME

Ce chapitre fournit les algorithmes répondant aux spécifications établies, quant à la définition du filtre. Partant d'un algorithme d'unification, nous introduisons un mécanisme de préunification, représentant la composante filtrage de l'unification. Pour mettre en oeuvre cet algorithme, non pas sur un, mais sur un ensemble de buts, nous proposons un codage de ces buts permettant de décomposer la préunification en plusieurs primitives pouvant s'exécuter en parallèle. Enfin nous verrons les traitements à effectuer en cas de succès de la préunification.

Chapitre IV

UNIFICATION PARALLELE

Les problèmes soulevés par l'utilisation de Prolog dans un contexte BC ne peuvent se résoudre sans des modifications notables du mécanisme "classique" d'interprétation. Au chapitre précédent nous avons esquissé les principes d'une stratégie de recherche basée sur la manipulation d'ensembles de solutions. Une telle approche ne doit pas être un point singulier, mais doit être généralisée à l'ensemble de la démarche sous peine de perdre les bénéfices de chaque amélioration "locale" de par la création de goulots d'étranglement.

Dans cet esprit nous allons définir un algorithme permettant d'unifier un terme à un ensemble de buts (désormais nous appellerons ainsi les "modèles") permettant de satisfaire les exigences de la stratégie de recherche. La mise en oeuvre de cet algorithme d'unification parallèle repose sur deux idées:

- La première est l'extraction de la composante filtrage, sujet que nous avons exposé au chapitre précédent (III.3). Nous la matérialisons par un algorithme dégradé d'unification, n'appliquant pas dynamiquement les substitutions: la préunification.

- La deuxième consiste en un codage des buts, scindant en deux composantes les informations qu'ils contiennent: les structures et les valeurs.

Rappelons que l'objectif d'implanter matériellement ces opérations sous-tend notre démarche, et par conséquence influe sur l'élaboration des solutions que l'on expose dans ce chapitre.

IV.1. Algorithme classique

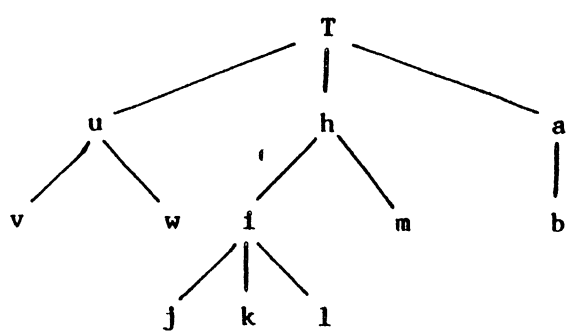
Nous fournissons tout d'abord quelques définitions permettant de décrire formellement un algorithme classique d'unification.
Pour commencer nous définirons un terme par la grammaire suivante :

```
Terme                 ::= Symbole_fonctionnel(<Suite-de-termes>)/
                          Atome /
                          variable
Suite-de-termes       ::= <Terme> /
                          <Terme> <Suite-de-termes>
Atome                 ::= Constante_symbolique /
                          Donnée_élémentaire
Symbole_fonctionnel   ::= Constante_symbolique
Donnée_élémentaire    ::= nombre / caractère / chaîne
```

On définit la position d'un élément (noeud ou feuille) dans un terme comme étant le chemin à parcourir depuis la racine pour atteindre cet élément: ce chemin peut être représenté par la suite des rangs de chaque élément parmi les fils du noeud père. Ce codage correspond en fait à la suite des valeurs de i (indice des fils) empilées dans les niveaux de récursivité de l'algorithme d'unification, présenté ci-après. On note pos(n,t) la position de l'élément n dans le terme t.

Exemple:

Soit le terme T (u(v,w) , h(i(j,k,l),m) , a(b)) ;



pos (T,l) = (2,1,3)

Une fonction "classique" d'unification opère sur deux termes et fournit soit un échec soit un succès. En cas de succès chaque terme est instancié par les substitutions nécessaires.

```
fonction unifie (t,u) ;
début
   si t est une variable alors
   début
      substituer t par u dans toutes ses occurences ;
      retour (succès) ;
   fin ;
   si u est une variable alors
   début
      substituer u par t dans toutes ses occurences ;
      retour (succès) ;
   fin ;
   si t est un atome alors
      si t = u alors retour (succès) sinon retour (échec) ;
   si t est un arbre et u est un arbre alors
   début
      si symbole_fonctionnel (u) ≠ symbole_fonctionnel (t) alors
         retour (échec) ;
      pour i = 1 jusqu'a nombre_de_fils (t) faire
      début
         ft = fils (t,i) ;
         fu = fils (u,i) ;
         si unifie (ft,fu) = échec alors retour (echec) ;
      fin ;
   fin sinon retour (échec) ;
   retour (succès) ;
fin.
```

- La fonction fils (t,i) retourne le ième fils du terme t.
- La fonction nombre_de_fils (t) retourne le nombre de fils du terme t.
- La fonction symbole_fonctionnel (t) retourne le symbole fonctionnel du terme t.

Le point délicat réside en l'implantation de la procédure de substitution. Voici les grandes lignes de la technique utilisée en logiciel (il existe différentes variantes cherchant, soit une diminution de l'espace mémoire requis et/ou sa récupération, soit un gain de performances, mais la "philosophie" demeure) :

- A chacun des deux termes à unifier est associé un **environnement** . Un environnement est un ensemble de **doublets** constitués par un pointeur environnement sur un autre environnement et un pointeur terme sur un autre terme (ces pointeurs pouvant être nuls).

- Substituer une variable par un terme (instancier la variable) revient alors à affecter au doublet représentant la variable :
  * l'adresse du terme au pointeur terme.
  * l'environnement du terme au pointeur environnement (éventuellement nul si le terme ne contiemt pas de variable, notamment si c'est un atome).

Une telle mise en oeuvre n'assure pas un fonctionnement correct de l'algorithme présenté ci-dessus. En effet lors de l'unification, en cas de reconnaissance d'une variable, il faut parcourir l'éventuel chaînage créé par des substitutions précédentes, ce afin d'atteindre le terme à unifier. Nous allons décomposer un exemple pour clarifier ces points et discuter des problèmes à résoudre.

Exemple:
Soit à unifier les 2 termes suivants

T1 : t (X , X , d) ;
T2 : t (a(b,Y) , a(b,c) , Y) ;

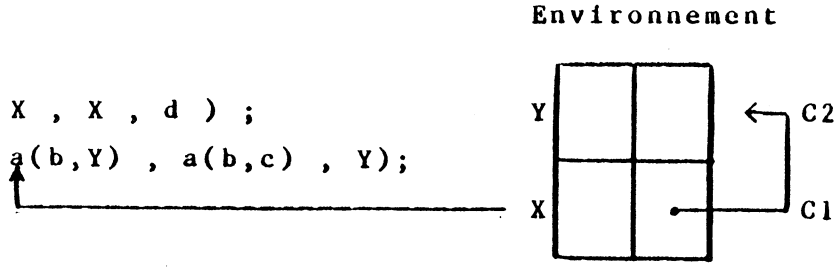
Voici les états successifs de l'algorithme exposé ci-dessus avec une représentation des substitutions lui assurant un fonctionnement correct. Les appels sont suffixés par la position du noeud en cours de traitement.

```
appel 1   :  t ( X , X , d ) ;
             t ( a(b,Y) , a(b,c) , Y) ;     [ t = t ]
appel 11  :  t ( X , X , d ) ;
             t ( a(b,Y) , a(b,c) , Y) ;     [ => X = a(b,Y) ]
appel 12  :  t ( a(b,Y) , a(b,Y) , d ) ;
             t ( a(b,Y) , a(b,c) , Y ) ;    [ a = a ]
appel 121 :  t ( a(b,Y) , a(b,Y) , d ) ;
             t ( a(b,Y) , a(b,c) , Y ) ;    [ b = b ]
appel 122 :  t ( a(b,Y) , a(b,Y) , d ) ;
             t ( a(b,Y) , a(b,c) , Y ) ;    [ => Y = c ]
appel 13  :  t ( a(b,c) , a(b,c) , d ) ;
             t ( a(b,c) , a(b,c) , c ) ;    [ => ECHEC c <> d ]
```

Voici les représentations obtenues à différentes étapes en utilisant la technique décrite précédemment:

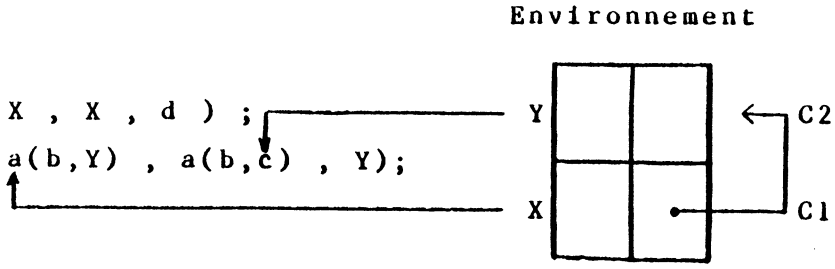
appel 12 :

```
      C1:  t ( X , X , d ) ;
      C2:  t ( a(b,Y) , a(b,c) , Y);
```



appel 13 :

```
      t ( X , X , d ) ;
      t ( a(b,Y) , a(b,c) , Y);
```



En résumé le choix de l'implantation de la procédure de substitution se pose en ces termes:

- Soit on utilise une technique de recopie de même type que celui de la forme précédente et cela nécessite une recopie dynamique de toutes les occurences d'une même variable dès que celle-ci est instanciée. Cette opération n'est pas réalisable en un temps linéaire car elle dépend, d'une part du terme à substituer, et

d'autre part du nombre d'occurences. Il est difficile d'espérer pouvoir suivre le débit du disque dans de telles conditions.

- Soit on emploie une technique de "structure partagée" pour la mémorisation des différentes substitutions ce qui complexifie l'algorithme d'unification. Son exécution ne peut plus alors se dérouler en un temps linéaire, car elle dépend du parcours des liens créés par les différentes substitutions. On aboutit à la même conclusion que précédemment.

En fonction de ces conclusions, nous avons opté pour une décomposition de l'unification en deux sous-étapes plus simples. Notre idée est de ne pas réaliser "en ligne" la procédure de substitution. Une première étape que l'on appellera **préunification** se contentera d'exécuter la fonction unifie générant les substitutions sans les appliquer. Une deuxième étape, appelée **association**, permettra d'assurer la cohérence des substitutions, c'est à dire vérifier que les occurences d'une même variable soient instanciées par des termes unifiables. Nous allons maintenant définir plus formellement ce que l'on entend par pré-unification, reportant à la fin du chapitre une présentation plus complète de l'association.

IV.2. Préunification

Nous venons de voir qu'intuitivement la préunification est une dégénérescence de l'unification en ce sens que les substitutions générées ne sont pas appliquées dynamiquement. Plus formellement on dira que deux termes T1 et T2 sont préunifiables si :

$\forall$ (N1 $\in$ T1, N2 $\in$ T2 tq pos(N1,T1) = pos(N2,T2)),
Si N1 $\neq$ N2 alors
(N1 est une variable ou N2 est une variable).

Les noeuds d'un terme sont définis par leur symbole fonctionnel et leur arité (nombre de fils). Deux noeuds sont donc égaux s'ils ont même symbole fonctionnel et même arité.

On extrait deux propositions de cette condition :

- La condition de préunification est nécessaire pour que deux termes soient unifiables (mais non suffisante). En effet s'il existe deux noeuds N1 de T1 et N2 de T2 tels que :

  pos (N1,T1) = pos (N2,T2) et
  N1 $\neq$ N2 et
  N1 $\neq$ variable et N2 $\neq$ variable

  alors l'algorithme d'unification échoue.
  Par contre elle n'est pas suffisante, pour preuve voici un contre-exemple :
  Soient : T1 : t (a,b) ; et T2 : t (X,X) ;

  T1 et T2 sont préunifiables mais non unifiables.

- Deux arbres préunifiables sont unifiables si chacun des sous-arbres susbstitués aux différentes occurences d'une même variable (s'il en existe) sont unifiables entre eux (d'aprés l'algorithme exposé).

La deuxième proposition assure la possibilité d'achever la préunification, en cas de succès. Notons que les unifications porteront sur des sous-termes ce qui diminuera leur complexité.
Mais revenons à la première proposition que l'on peut énnoncer également comme suit:

- Si deux arbres T1 et T2 ne sont pas préunifiables alors ils ne sont pas unifiables.

Cette propriété exprime la composante filtrage de l'unification. Son exploitation est corrélée aux deux hypothèses suivantes, justifiées au chapitre précédent, en partie par des considérations reposant su les spécificités d'un contexte BC:
- L'unification a un taux d'échec élevé.
- La préunification détecte un fort pourcentage de ces échecs.

Les mesures effectuées par Gilles Berger-Sabbatel ont permis de valider ces hypothèses en ce qui concerne le taux d'échec de l'unification, et le taux de détection de ces échecs par la préunification [BERb 85].

La préunification consistera d'une part à vérifier que deux termes sont préunifiables et d'autre part à générer la liste des substitutions induites. L'algorithme de préunification se déduit de celui d'unification en remplaçant la procédure de substitution par une procédure de génération de substitution.

En cas de succés de la préunification il suffit de vérifier la cohérence des substitutions générées, étape que l'on appellera **association** . Nous détaillerons cette phase plus en détail aprés avoir défini l' algorithme de préunification

IV.3. Codage des buts

Nous rappelons que l'objectif issu de l'anayse effectuée au chapitre III, quant à la stratégie de recherche à adopter dans un contexte BC, est d'unifier en parallèle non pas un, mais un ensemble de buts. Tenant compte de la discussion précédente, nous tâcherons tout d'abord d'effectuer une préunification sur cet ensemble de buts. Il est clair que ces objectifs vont entraîner une remise en question totale de l'algorithme de préunification opérant sur un seul but. Notre idée est de séparer en différents éléments les informations dont sont composés les buts et de déterminer les opérations à effectuer sur chacun de ces éléments pour aboutir à la préunification de cet ensemble de buts. Ainsi on espère d'une part réduire la complexité de chaque sous-traitement, et d'autre part pouvoir les effectuer en parallèle.

IV.3.1. Décomposition d'un but

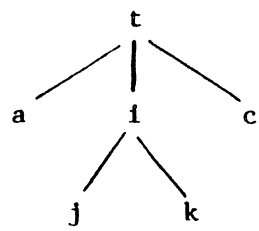
Un but ayant une structure d'arbre, on peut le scinder en deux composantes :
- Sa structure.
- La valeur des noeuds de cette structure.

On définira la structure d'un but par un graphe. Les sommets de ce graphe sont reliés par les transitions Fils, Frère, Père obtenues par le parcours préfixé de l'arbre source. Chaque sommet est alors identifié par sa position dans cet arbre. Pour conserver toute l'information on associe à chaque sommet la valeur du noeud source.

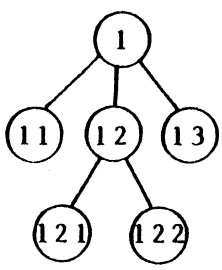
Par convention on notera les transitions comme suit :
- Fils => F
- Frère => S
- Père => P

La notation Tij (Ni,Nj) signifie que les sommets Ni, Nj sont reliés par la transition Tij.

Exemple: Soit le terme t (a , i(j,k) , c) ;



Structure



Positions

1 : t
11 : a
12 : i
13 : c
121 : j
122 : k

Cette représentation offre deux possibilités :

- La première est de pouvoir décomposer la préunification en plusieurs processus exécutables parallèlement. Nous les expliciterons dans la suite.

- La deuxième est liée à la possiblité de partager les informations, point que nous allons maintenant aborder.

IV.3.2. Mémorisation d'un ensemble de buts

Nous allons tout d'abord définir la notion de structure "englobante" de plusieurs buts. Intuitivement, celle-ci est constituée de l'union des structures des buts sources. Plus formellement on donnera la définition suivante :

- Soient n buts C1,...,Cn.
  Soient les structures S1,...,Sn issues de ces n buts.
  La structure englobante de S1,...,Sn est la structure S vérifiant les deux conditions suivantes :

  i) Pour tout Si (i=[1,n])
     Pour tout Nj,Nk,Tjk appartenant à Si tq Tjk (Nj,Nk) alors
     il existe Nu,Nv,Tuv de S tq
     - Tuv (Nu,Nv)
     - Pos (Nu,S) = Pos (Nj,Si)
     - Pos (Nv,S) = Pos (Nk,Si)

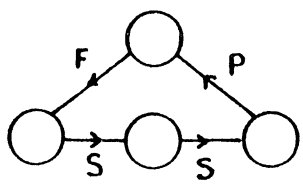
Et réciproquement :

  ii) Pour tout Nu,Nv,Tuv appartenant à S tq Tuv (Nu,Nv) alors
     il existe au moins un Si (i=[1,n]) tq
     il existe Nj,Nk,Tjk tq
     - Tjk (Nj,Nk)
     - Pos (Nj,Si) = Pos (Nu,S)
     - Pos (Nk,Si) = Pos (Nv,S)

Cette structure existe et nous en donnons pour preuve son algorithme de construction. Celui-ci est commenté au chapitre suivant et explicité dans l'annexe 3.

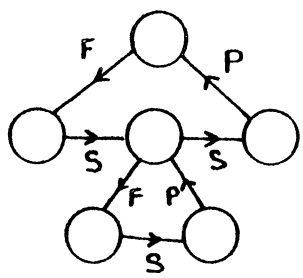
Exemple:

t(a,b,c).



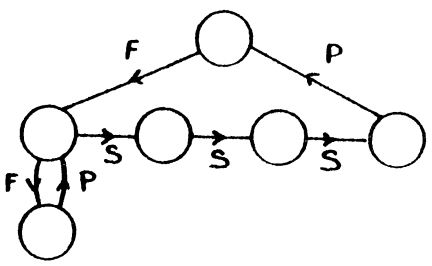
S1

t(a,i(j,k),d).



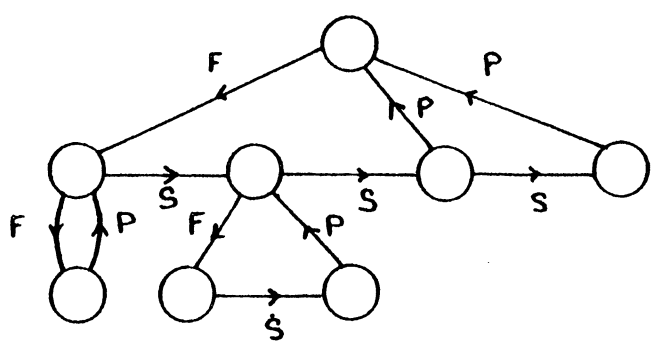
S2

p(f(g),b,c,d).



S3

La structure englobante S est la suivante :

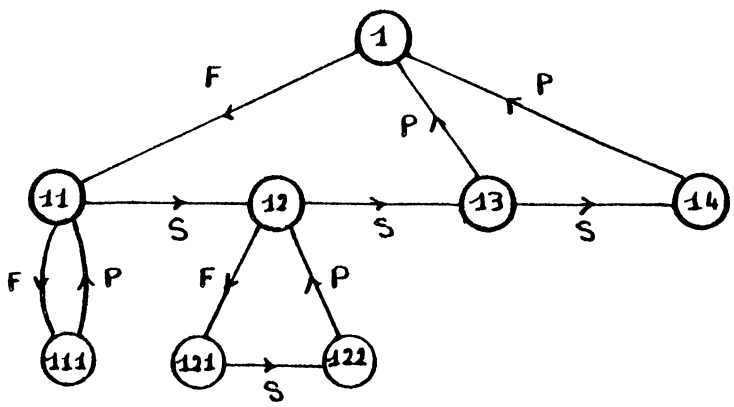


Pour un but une seule valeur était associée à chaque noeud de la structure sous-jacente. De façon identique, pour un ensemble de buts, on associe à chaque noeud N de position pos (N,S), dans la structure englobante S de cet ensemble de buts, l'ensemble des valeurs des noeuds de même position dans les buts sources.
En reprenant le formalisme et le contexte utilisés pour définir la structure englobante, on donnera la définition suivante d'un ensemble de valeurs :

A tout noeud N de S on associe l'ensemble de valeurs défini par

{ val (Nj) / pos (Nj,Si) = pos (N,S) }
pour tout Si de l'ensemble des buts sources.

où val(N) retourne la valeur du noeud N.



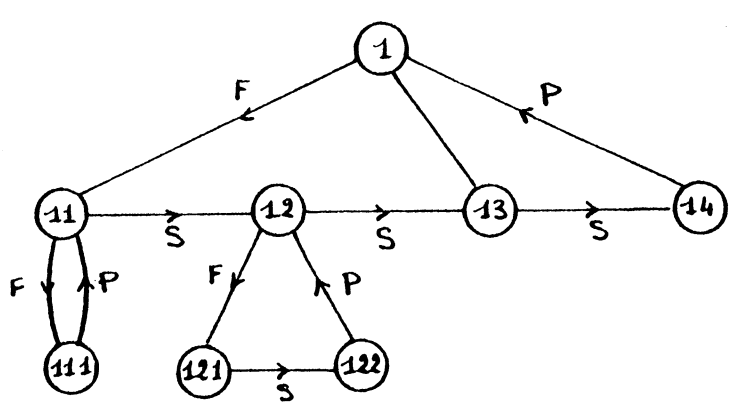
Pos		listes valeur
1	:	(t,p)
11	:	(a,f)
12	:	(b,i)
13	:	(c,d)
14	:	(d)
111	:	(g)
121	:	(j)
122	:	(k)

Enfin, pour conserver toute l'information fournie par les buts de départ, il suffit de mémoriser pour chaque valeur, le ou les buts sources. Pour cela on affectera un indice à chacun, permettant de l'identifier. Plus précisément, à tout élément Vk d'un ensemble de valeur V associé à un noeud N de S, on attache un ensemble d'indices défini par :

{ i / pos(Nj,Si) = pos(N,S) et val(Nj,Si) = Vk }

En indiçant comme suit les buts de notre exemple, on obtient la représentation finale suivante:

indices		buts
1	:	t(a,b,c).
2	:	t(a,i(j,k),d).
3	:	p(f(g),b,c,d).



Pos		valeurs et indices
1	:	(t (1,2) , p (3))
11	:	(a (1,2) , f (3))
12	:	(b (1,3) , i (2))
13	:	(c (1,3) , d (2))
14	:	(d (4))
111	:	(g (1))
121	:	(j (2))
122	:	(k (2))

Un point délicat réside en la représentation des variables. Nous avons choisi une implantation favorisant l'exécution de l'algorithme de préunification, au détriment du codage lui-même. Lorsqu'une variable apparaît en un noeud d'un but, il faut rajouter à chaque valeur de la liste attachée à ce noeud l'indice du but car une variable représente n'importe quelle valeur. Si l'on reprend l'exemple précédent en modifiant le but d'indice 1 ainsi,

```
t ( a , b , c ) ;      =>      t ( a , X , c ) ;
```

On obtient le codage suivant des listes de valeurs et d'indices :

```
. Pos      valeurs et indices
  1    :   ( t (1,2) , p (3) )
  11   :   ( a (1,2) , f (3) )
  12   :   ( X (1) , b (1,3) , i (1,2) )
  13   :   ( c (1,3) , d (2) )
  14   :   ( d (4) )
  111  :   ( g (1) )
  121  :   ( j (1,2) )
  122  :   ( k (1,2) )
```

Notons la répercussion de l'apparition d'une variable en un noeud sur les listes associées à ses noeuds descendants (noeuds de position "121" et "122" sur l'exemple). Nous justifierons cette implantation en présentant l'algorithme.

IV.4. Algorithme

Avant de fournir le corps de l'algorithme, nous allons définir plusieurs primitives sur lesquelles il est construit. Nous verrons que la plupart découlent "naturellement" du codage des buts. Nous essaierons également de préciser les liens logiques entre ces différentes opérations. Nous reviendrons au chapitre VIII sur les relations temporelles et notamment sur les possibilités de parallélisme.

IV.4.1. Décodage

Le décodage réalise en quelque sorte l'interface entre la représentation du terme analysé et la mémorisation de l'ensemble de buts. A cette fin il fournit les indications suivantes :

- Le type du symbole courant.

- Son arité si c'est un symbole fonctionnel, 0 sinon.

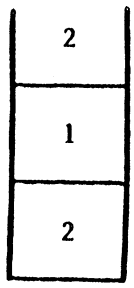
Cette primitive assure également un travail de synchonisation en ce sens qu'elle initialise les autres pour chaque nouveau symbole traité. Elle fournit ses résultats à la primitive analysant la structure du terme lu, et active la primitive de recherche.

IV.4.2. Génération des transitions

En fonction de l'arité du symbole lu, cette primitive va générer la transition en découlant. Pour celà elle mémorise, à l'aide d'une pile, la structure du terme en cours d'analyse en stockant le nombre de fils à reconnaitre pour chaque noeud ancêtre du nouveau noeud.

Exemple:

Soit le terme t (a , b (c , d(e,f)) , g) à analyser, et supposons que le noeud courant soit celui ayant la valeur d. La pile serait la suivante :



(d => 2 fils à reconnaître)

(b => 1 fils à reconnaître (c déja vu))

(t => 2 fils à reconnaître (a déja vu))

La primitive génère, à l'aide de cette pile et de l'arité du symbole courant, une des transitions F, S, P, reçues par la primitive parcours. On fournit en annexe 4 l'algorithme de génération des transistions

IV.4.3. Parcours

Cette primitive assure le déplacement dans la structure englobante. Elle peut se conceptualiser comme un automate d'états finis, ayant pour entrées les transitions Fils, Frère, Père. En fonction de celles-ci l'automate change d'état et fournit un certain nombre de sorties :

- Le code de position du noeud courant.
- Un indicateur de substitution.
- L'adresse de l'ensemble de valeurs associé au noeud courant.
- Eventuellement un échec si la transition reçue n'est pas reconnue en l'état courant.

Les indications relatives à la génération des substitutions, seront interprétées par la primitive de gestion des substitutions, alors que l'adresse de l'ensemble de valeurs sera dirigée vers la primitive de recherche.

IV.4.4. Recherche

Il s'agit tout simplement de la recherche de la valeur du symbole courant dans l'ensemble identifié par la primitive parcours (l'ensemble est matérialisé sous forme d'une liste). Deux issues sont possibles:

- La valeur est trouvée et l'adresse de la liste d'indices (les ensembles d'indices sont également représentés sous forme de liste) attachée à cette valeur est envoyée à la primitive "et".

- La recherche échoue et un signal d'échec est émis en direction de la primitive de gestion des substitutions.

IV.4.5. "et"

Cette primitive effectue, pour chaque nouveau symbole lu, la mise à jour de l'ensemble de buts encore préunifiables à ce stade de l'analyse. Pour ce faire elle réalise l'intersection ("et" logique) entre la liste couramment adressée et la liste résultante des intersections précédentes. Si la liste devient vide elle émet un signal d'échec en direction de la primitive de décodage qui recherchera la fin du terme courant.
Si l'analyse du terme aboutit elle fournira les indices des buts préunifiés.

IV.4.6. Gestion des substitutions

La gestion des substitutions a deux rôles :

- Générer les substitutions c'est à dire contrôler leurs débuts et fins. Les débuts lui sont signalés par la primitive parcours. La fin est déterminée à l'aide de la pile mémorisant la structure du terme en cours d'analyse.

- Rechercher la fin d'une substitution si la primitive de recherche émet un signal d'échec alors qu'une substitution est en cours. Une fois la fin de la substitution atteinte, elle repositionne l'automate de parcours et réactive la primitive de décodage pour relancer l'analyse de la suite du terme.

IV.4.7. Corps de l'algorithme

Nous fournissons ici l'algorithme construit sur les primitives décrites précédemment.

```
/* état automate = racine de la structure ; pile_vide ;
   lecture de la première donnée  */

Tantque ( DECODAGE ≠ fin de terme ) faire
   Si (variable_lue) alors générer substitution var disque
   Sinon début
      GESTION début de substitution;
      Activation de la primitive RECHERCHE;
      Si (signal recherche fin substitution) alors
         Si (substitution en cours) alors rechercher sa fin
                                    sinon allera ECHEC;
      Activation primitive ET;
      Si échec alors allera ECHEC;
   Fin sinon
   GESTION fin de substitution;
   GENERATION et PARCOURS de la transition;
   Si échec alors allera ECHEC;
   Lecture de la donnée suivante;
Fin tantque

ECHEC : Traitement de fin de clause;
/* Si échec recherche fin de clause
   sinon transmission de la queue */
Fin tantque
```

Notons que cet algorithme n'est certes pas optimal au point de vue efficacité, ce qui s'explique par le fait que l'objectif final n'est pas une réalisation logicielle. Cette implantation permet de satisfaire deux objectifs:

- D'une part la validation de l'algorithme.

- D'autre part une esquisse d'une simulation logicielle.

Si le premier objectif a été atteint (cf. chapitre suivant) le deuxième s'est vu limité par le manque d'outils permettant d'exprimer le parallélisme et plus généralement des notions temporelles.
Par contre la fonctionalité de chaque procédure a pu être validée et nous verrons au chapitre VIII les détails d'une simulation plus complète.

En cas de succès de la préunification, rappelons qu'une deuxième étape est nécessaire, étape que nous avons appelé association et que nous allons maintenant présenter.

IV.5. Association

L'étape d'association assure en fait deux tâches qui se trouvent "naturellement" étroitement imbriquées dans leur implantation :

- D'une part elle met en forme les résultats de la préunification et sert ainsi d'interface avec une représentation plus classique des termes (le codage proposé n'étant pas à priori extensible à l'ensemble de l'interprétation).

- D'autre part elle vérifie la cohérence des substitutions générées et termine ainsi l'unification.

IV.5.1. Interface

En cas de succès, l'algorithme de préunification fournit les informations suivantes :
- les indices des cibles atteintes.
- Les substitutions générées.

Les substitutions sont émises sous la forme de couples (POS , TERME) où :
- POS indique la position, dans la structure englobante, du noeud où la substitution a été générée.
- TERME est le terme lu sur le disque, et peut éventuellement être une variable.

Réaliser l'interface revient à faire correspondre aux positions des substitutions les positions identiques des noeuds des buts atteints, chaque noeud de ces buts étant codé sous le format de couple (POS,VALEUR) où VALEUR peut-être soit un atome soit une variable.

Dès lors la cohérence des substitutions peut être effectuée de part la possibilité d'identifier les occurences d'une même variable. En fait nous allons voir que les deux tâches sont réalisées simultanément.

IV.5.2. Cohérence

Assurer la cohérence revient à vérifier si les termes substitués aux occurences d'une même variable (apparaissant dans un but ou dans les données), sont unifiables entre eux. Il est clair que s'il n'existe pas plusieurs occurences d'une même variable la cohérence est automatiquement assurée. Nous discuterons de la fréquence des différents cas de figure au chapitre suivant. Pour l'heure nous exposons la technique utilisée dans sa généralité et verrons comment optimiser les différents cas dans ce même chapitre.

La mise en oeuvre est voisine de celle exposée en IV.1. On attribue un contexte d'association au but atteint courant (tous les buts atteints par un terme sont traités maintenant séquentiellement), et un autre contexte au terme traité. Un contexte est un ensemble de doublets, un doublet étant constitué de deux pointeurs l'un sur le but courant l'autre sur les substitutions générées.

Pour chaque substitution générée sous la forme (POS,TERME), l'alternative suivante se présente

- Soit la valeur du noeud de position POS dans le but atteint est une variable: l'identificateur (pour le but courant) de cette variable fournit l'entrée dans le contexte d'association et ainsi l'état de cette variable. Si celle-ci est libre, alors on la lie au terme (pouvant éventuellement être une variable) de la substitution traitée. Sinon, une procédure d'unification est activée entre le terme de la substitution et le terme fourni par le contexte.

- Soit la valeur du noeud de position POS dans la cible atteinte est un symbole: ceci signifie que le terme de la substitution est une variable. Les mêmes opérations sont exécutées mais avec le contexte de la clause traitée.

En cas de succès des différentes unifications nécessaires, les deux contextes permettent de récupérer les substitutions dont la cohérence est dès lors assurée.

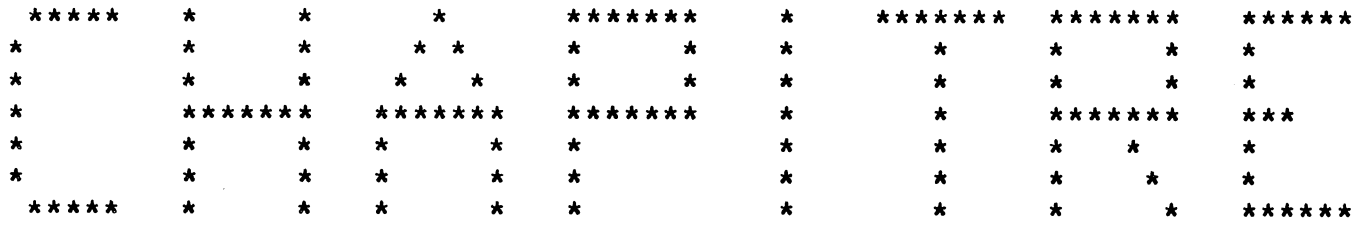
IV.6. Conclusions

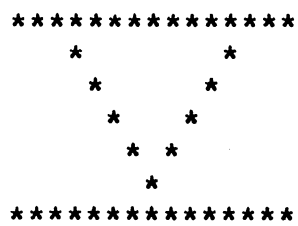
Au terme de ce chapitre les idées, quant à l'élaboration du processus de recherche d'une part, et quant à la symbiose entre ce processus et la stratégie de recherche d'autre part, sont en place. Si les spécifications fonctionnelles semblent satisfaites, d'un point de vue temporel, l'évaluation est plus délicate. Les difficultés, pour atteindre les performances énoncées, se situent en deux points:

- Le premier est lié à la nécessité de coder les buts sources, en la représentation présentée en IV.3. Cette phase de "compilation" doit être suffisamment rapide pour ne pas dégrader les performances, nous aborderons le sujet au chapitre suivant.

- Le deuxième concerne la mise en oeuvre d'un matériel supportant l'algorithme de préunification. Dans ce sens, la définition de plusieurs primitives relativement simples et indépendantes (au sens de l'exécution), permet d'envisager une intégration satisfaisant nos contraintes.

D'un point de vue matériel, ce chapitre fournit les spécifications de l'opérateur à réaliser, et servira de point de départ pour la troisième partie, traitant du sujet. D'un point de vue logiciel, il est la base de l'implantation que nous allons maintenant décrire.





IMPLANTATION LOGICIELLE

CHAPITRE V

RESUME

Ce chapitre concerne l'implantation logicielle des algorithmes. Il propose tout d'abord des considérations matérielles, en raison de l'objectif final, qui aboutissent à des spécifications énoncées sous formes de contraintes. A la suite de quoi, on détaillera l'implantation de l'algorithme de compilation en fournissant des mesures quant à ses performances. Puis on précisera celui d'unification, construit et exposé dans l'optique de dégager des primitives transposables en matériel. Une discussion, sur l'achèvement de l'unification au terme d'un succès de la préunification, termine ce chapitre.

Chapitre V

IMPLANTATION LOGICIELLE

Les chapitres précédents nous ont permis de développer les idées orientant nos travaux. Si notre objectif demeure la réalisation matérielle de la machine supportant ces idées, il n'est pas pensable pouvoir atteindre ce but sans une étape intermédiaire de simulation logicielle. Ceci étant, il n'est pas non plus question de présenter immédiatement la simulation complète de la machine. Nous nous attacherons tout d'abord à valider les algorithmes de compilation et d'unification, la stratégie de recherche faisant l'objet d'autres travaux avant de pouvoir prétendre simuler tout ou partie de la machine.

Il est clair que si ces deux études doivent aboutir pour qu'une machine puisse voir le jour, rien n'interdit de ne réaliser matériellement, dans un premier temps, que l'une ou l'autre des deux parties au vu de leur indépendance à ce niveau physique. Ainsi nous pensons, pour une première étape, élaborer une maquette ayant une stratégie de recherche implantée par logiciel et une partie processeur disque, intégrant l'unification, réalisée par matériel.

Dans ce chapitre, nous discuterons des considérations matérielles sous-jacentes aux idées précédentes, puis nous présenterons successivement l'implantation des algorithmes de compilation, de préunification, d'association et nous terminerons par l'intégration de ce "processeur disque logiciel" dans notre interpréteur YAAP. De plus nous fournirons et discuterons les mesures effectuées sur chacun.

V.1. Considérations matérielles

Pour respecter les objectifs et la démarche adoptée jusqu'alors, les algorithmes décrits le seront en tenant compte des exigences et possibilités matérielles. Dès lors il faut essayer de réaliser un consensus entre :

- La réalisation d'une validation purement fonctionnelle.

- La volonté de prendre en compte l'issue matérielle.

Dans cette optique, nous allons rappeler ou expliciter les différentes contraintes qui influeront sur cette implantation logicielle.

V.1.1. Contraintes temporelles

Nous nous plaçons ici au niveau du processeur disque qui est vu par le reste de la machine comme un élément autonome. Il reçoit les requêtes sous la forme d'un ensemble de buts et d'un espace de recherche sur le disque, et retourne les solutions trouvées.

Pour ce faire le processeur disque (Fig V.1) est muni:
- D'un micro-processeur pouvant effectuer la compilation des buts et l'étape d'association de l'algorithme d'unification,
- D'un filtre réalisant la préunification,
- D'un contrôleur disque.

L'objectif est de réaliser l'unification au vol c'est à dire sans ralentir le balayage de l'espace de recherche disque. En explicitant on obtient les contraintes suivantes :

- Effectuer la compilation pendant le temps de positionnement de l'unité de disque.
- Exécuter la préunification au rythme imposé par le contrôleur disque.
- Assurer que l'association, étape finale de l'unification, demande moins de temps que l'unification complète.

Si la préunification est exécutée par matériel, la compilation et l'association seront à la charge du microprocesseur. De ce fait

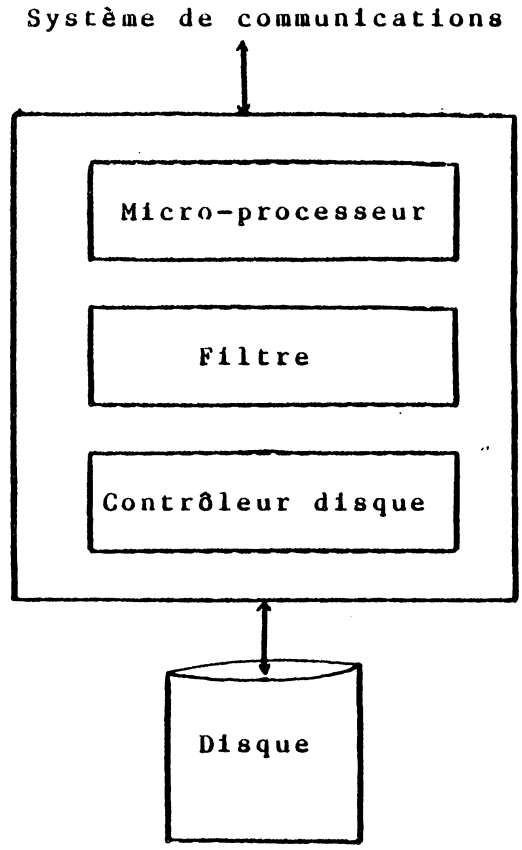


Figure V.1

il est raisonable d'essayer d'implanter de façon optimale ces opérations, leurs spécifications étant achevées. Par contre, pour ce qui est du filtre, nous allons voir les incidences de l'issue matérielle sur l'implantation logicielle.

V.1.2. Contraintes matérielles

Nous venons de voir que les considérations matérielles affectent essentiellement l'algorithme de préunification, sa description fonctionnelle ayant déjà été guidée par celles-ci. Nous fournirons au cours de ce chapitre un raffinement de ces contraintes exprimées en un premier temps, par l'extraction des différentes primitives.

En revanche cette implantation délaissera totalement les considérations temporelles, "inter-primitives", encore trop peu définies. Celles-ci seront affinées à l'occasion de ce travail, ce qui permettra d'aboutir à une véritable simulation que nous présenterons dans la deuxième partie.

Nous nous contenterons ici de respecter les possibilités matérielles classiques principalement vis à vis des structures de données utilisées et de la complexité des algorithmes les manipulant. Par exemple une recherche étant nécessaire nous préférerons une recherche séquentielle plutôt que dichotomique ...

Mais avant d'aborder ces problèmes dans la description de l'algorithme de préunification, nous allons voir celui de compilation.

V.2. Compilation

La compilation est l'opération qui génère, à partir d'un ensemble de buts codés sous forme de terme suivant la grammaire définie en IV.1., la représentation exposée en IV.3.2. Elle constitue donc le premier maillon de la chaîne de traitement.

Si les spécifications fonctionnelles de ce processus sont bien définies, la stratégie de recherche influe sur son implantation, en fonction de considérations temporelles. Les problèmes de temps de traitement sont en effet primordiaux, et nous en discuterons en fournissant des mesures.

V.2.1. Stratégie adoptée

Deux stratégies sont envisageables quant à l'exécution de la compilation :

- Compilation statique : le code est généré en fonction de l'ensemble de buts, ce qui implique que cet ensemble soit fixé et présent au début de l'opération et ne pourra être modifié au cours du déroulement de celle-ci.

- Compilation dynamique : à l'inverse le code est généré de façon incrémentale prenant en compte les buts les uns après les autres.

A la vue de la stratégie de recherche adoptée, un ensemble de buts va être généré non pas d'un seul bloc mais paquet par paquet. De ce fait il est clair que la deuxième solution s'impose pour assurer la cohérence des relations temporelles des différentes opérations. Qui plus est, cette caractéristique peut allouer, à la phase de compilation, un laps de temps plus important que celui du positionnement de l'unité de disque, dès lors que l'arrivée des buts s'échelonne dans un délai supérieur à ce temps de positionnement.

De plus cette solution permet de disposer, au terme de la compilation de chaque but, d'un code complet. Ceci offre la possibilité de déclencher la préunification d'une façon plus autonome et plus asynchrone, sachant que les buts non encore arrivés seront traités dans un deuxième passage. On dispose ainsi de tous les

moyens nécessaires pour éviter les goulots d'étranglement et utiliser les temps d'attente liés à l'unité de disque.

En fonction de ces arguments, nous avons opté pour une compilation dynamique. Notons toutefois qu'une compilation statique, pourrait se révéler intrinséquement plus rapide, mais ce gain se trouve annulé par le contexte. Quoiqu'il en soit elle entraînerait une rigidité de par la nécessité d'attendre et de fixer un ensemble de buts. Voyons dès lors les principales caractéristiques de cet algorithme.

V.2.2. Spécifications

Une partie des spécifications est directement issue de l'objectif de générer le codage exposé en IV.3.2. L'autre partie est induite par la volonté de réaliser par matériel les primitives de l'algorithme de préunification. De ce fait le code doit être scindé en autant de composantes que de primitives, ou plus exactement que de procédures manipulant des données issues des buts. Nous distinguons quatre constituants dont nous précisons la sémantique :

- Parcours: contient la représentation de la structure englobante. Plus précisément, chaque noeud doit être matérialisé par:
  - Les transitions qu'il accepte et les noeuds ainsi atteints.
  - Le début et la longueur de la liste associée.
  - Un indicateur de substitution précisant si une variable appartient à la liste de valeurs.

- Recherche: constitue les listes de valeur associées à chaque noeud. Chaque élément d'une liste est un doublet composé d'un symbole et d'un pointeur vers la liste d'indices qui lui est attachée.

- Et: encode les listes d'indices.

Ces trois parties de code serviront à "programmer" les trois opérations correspondantes. Par contre la dernière partie sera utilisée par le micro-processeur pour effectuer l'association.

- Positions: associe à chaque symbole de chaque but sa position dans la structure englobante.

V.2.3. L'algorithme

On fournira en annexe 5 le corps de l'algorithme ainsi que quelques exemples de code généré pour différents ensembles de buts. Nous ne présentons ici que les caractéristiques de cet algorithme qui se déroule en deux étapes.

V.2.3.1. Génération du code

La première étape est guidée par la construction de la structure englobante. A partir d'une structure vide, la construction s'effectue en intégrant les buts les uns après les autres. L'analyse consiste à parcourir simultanément, en ordre préfixé, la structure issue du but courant et la structure englobante. On se trouve donc toujours en des noeuds de position identique dans l'un et l'autre. Pour chaque noeud du but courant on vérifie que la transition permettant d'accéder au noeud suivant existe dans la structure englobante. Si ce n'est le cas on la rajoute.

Pour chaque noeud atteint, on gère également la liste de valeurs associée. L'alternative suivante se présente:

- Soit la valeur du noeud courant du but ne se trouve pas dans la liste du noeud courant (donc de même position) de la structure englobante. Dans ce cas on ajoute la valeur à la liste et on lui attache l'indice du but courant.

- Soit la valeur se trouve déjà dans la liste. On ajoute alors à la liste d'indices attachée à cette valeur, l'indice du but courant.

Dans les deux cas il faut effectuer une recherche, opération coûteuse d'un point de vue temps d'exécution.

Pour terminer la description de cette première étape, précisons que simultanément est généré le code positions en associant à chaque symbole sa position dans la structure englobante, matérialisée par l'adresse du noeud courant.

V.2.3.2. Répercussion des variables

Deux opérations restent à faire à l'issue de la première étape, opérations liées à des considérations matérielles:

- Répercuter les effets des apparitions de variables en des noeuds.

- Transformer les listes chaînées en listes physiquement contiguës.

La première opération est effectuée sur tous les noeuds ayant une, ou plusieurs variables soit dans leurs listes de valeurs, soit dans une liste de l'un de leurs noeuds ancêtres. Elle consiste à rajouter aux listes d'indices attachées à chaque élément de la liste de valeurs le (ou les) indices du (ou des) but(s) possesseur(s) de la (ou les) variable(s).
D'un point de vue sémantique cela signifie que l'on considère qu'une variable peut se substituer à n'importe quelle valeur et qui plus est à n'importe quel terme. Ceci assure la cohérence de la représentation.

La deuxième opération permet d'une part d'effectuer facilement la répercussion des variables, et d'autre part de satisfaire aux exigences matérielles d'une recherche câblée séquentielle.

On fournit dans l'annexe 5 plusieurs exemples mettant en relief ces différents points.

V.2.4. La capacité

Une question délaissée jusqu'ici commence à se montrer pressante:

- "Combien de buts peut-on espérer unifier en parallèle ? "

La réponse est d'une part liée au temps de compilation, qui augmente pour chaque nouveau but, et d'autre part fonction des impératifs matériels. Nous proposons les objectifs suivants:

- Un maximun de 64 buts en parallèle.

- Un maximun de 128 valeurs pour l'ensemble des buts.

Nous développerons ci-dessous les éléments de réponse qui les justifient, quant à la compilation, repoussant au paragraphe suivant les justifications matérielles des chiffres avancés.

V.2.5. Mesures

Au vu de l'algorithme trois procédures se dégagent (on réunit les fonctions ayant le même objectif dans les deux étapes):
- La création des listes de valeurs.
- La création des listes d'indices.
- Le "parcours/création" de la structure englobante.

Les premières mesures ont confirmé cette émergence, quant aux temps d'exécution. Des jeux de tests plus sélectifs ont permis d'étudier plus particulièrement les paramètres importants de chaque opération et notamment d'en évaluer le coût. L'annexe 6 présente en détail ces mesures. La suite en fournit la synthèse.

Les résultats montrent que le point crucial réside en la création des listes de valeurs, le temps d'exécution dépendant du nombre final d'éléments des listes. Ceci s'explique aisément en rappelant que la création nécessite une recherche. En limitant la capacité du filtre à 64 cibles et 128 valeurs, le cas le plus défavorable consiste en la création de deux listes de 64 valeurs. Dans cette situation le temps de compilation est de 60 ms. se répartissant comme suit :

- 35 % création des listes de valeurs.
- 35 % parcours/création de la structure englobante.
- 25 % création des listes d'index.
- 5 % divers.

Au vu de ces résultats, le temps moyen de création des listes ne paraît pas nettement supérieur à celui des autres procédures. Ceci est la conséquence d'un premier travail d'optimisation, quant à l'écriture de la fonction de recherche, effort ayant pratiquemnt divisé par deux le temps d'exécution. De tels gains sont envisageables sur les deux autres fonctions (ainsi qu'une amélioration des

performances de la création). Le cas le plus défavorable devrait ainsi pouvoir être compilé en 40 ms, pour une implantation en langage C tournant sous une SM90.

Actuellement une compilation nécessitant la création d'une seule liste de 64 symboles se fait en 40 ms, et 64 buts ne nécessitant pas la création de listes aussi longues, demandent de 40 à 50 ms. Ceci apporte une première justification pratique à la limite de 64 buts. Avec des moyens de calcul plus puissants une limite de 256 buts est envisageable.

En optimisant le code, un cas de figure moyen sera compilé en moins de 30 ms. La compilation pourra ainsi s'effectuer, dans la plupart des cas, durant le temps de positionnement de l'unité de disque.

V.3. Préunification

L'implantation du corps de l'algorithme est le reflet fidèle de l'algorithme fonctionnel fournir en IV.4.7. Nous avons délaissé le facteur temporel pour deux raisons:

- Les problèmes temporels (synchronisation, parallélisme), primordiaux pour une réalisation matérielle, peuvent et doivent être écartés pour concentrer l'effort vers une clarification des fonctionnalités des différentes primitives.

- Peu, voire pas, d'outils sont disponibles pour les prendre en compte.

Nous nous sommes attachés à implanter chaque primitive en fonction de possibilités matérielles, repoussant à l'étape de simulation les problèmes cités ci-dessus.
De plus chacune est organisée de sorte à être la plus autonome possible, et ne nécessiter que le minimum de communications avec les autres. Pour ce faire nous nous sommes servis des concepts mis à jour dans la programmation orientée objet, et nous verrons comment ces concepts nous ont servi de guide quant aux spécifications matérielles.

V.3.1. Aperçu des primitives matérielles

Trois opérateurs matériels se dégagent de façon évidente, et nous avons respecté dans la programmation leur structure physique "naturelle":

- Primitive PARCOURS: rappelons que le concept sous-jacent est celui d'automates d'états finis. L'implantation matérialise un noeud sous forme d'un ensemble de champs. Trois d'entre eux permettent d'accepter ou de rejeter les transitions Fils, Frère, Père et en cas de reconnaissance fournissent l'adresse du prochain noeud. Les autres champs stockent les informations générées par la primitive: indicateur de substitution, début et fin de la liste associée. L'automate est constitué d'un ensemble de tels enregistrements, leurs adresses encodant les positions des noeuds dans la structure.

Un mot de l'automate requiert environ 40 bits. Le nombre de transitions possibles étant limité par le nombre de valeurs, fixées à 128, la taille de la mémoire pour l'automate parcours est de:

Tparc = 40 * 128 = 5120 bits

- Primitive RECHERCHE: un opérateur matériel simple est celui réalisant une recherche séquentielle. C'est pourquoi nous l'avons implantée de la sorte.
  En rappelant que les symboles sont codés sur 5 octets et que le nombre de valeurs est fixé à 128, la taille de la mémoire stockant les listes est de:

  Trec = 40 * 128 = 5120 bits

- Primitive ET: l'idée est d'associer à chaque but un bit dans une chaîne, dont la longueur est égale au nombre de buts filtrés en parallèle. L'indice du but correspond alors au rang du bit dans la chaîne. Ainsi la présence d'un but dans la liste se matérialise par le positionnement à "1" du bit le représentant et à "0" sinon. De ce fait la gestion des listes s'effectuent à l'aide de "et logiques".
  Au vu des limites proposées précédemment, 64 buts et 128 valeurs, la taille de la mémoire stockant la matrice de bits est de:

  Tbit = 64 * 128 = 8 Kbits

Les autres primitives n'ont pas une réalisation aussi immédiate, mais leur implantation logicielle a permis de définir les besoins matériels que nous exposerons dans la deuxième partie.

Le cumul des tailles des mémoires statiques, nécessaires au stockage du code issu de la compilation des buts, donne une taille globale d'environ 16 Kbits. Aussi on peut espérer qu'une telle dimension ne sera pas prohibitive quant à l'implantation matérielle de l'algorithme en un seul circuit. Notons que l'on a réfléchi à un compactage de la matrice de bits, basé sur la constatation que les lignes de cette matrice sont soit très vides, pratiquement un seul bit à un pour les chaînes attachées à un symbole, soit très pleines, presque tous les bits à un dans le cas des chaînes liées à une variable. Notre idée consiste à coder uniquement les bits à un ou à zéro (en mémorisant également pour chaque ligne le type de codage utilisé). Le gain de place peut s'avérer important, plus de

50% pour les limites proposées. La réalisation du "et" devient alors un peu plus complexe, mais encore largement réalisable dans le temps qui lui est imparti.

V.3.2. Mesures

On pourrait penser, à priori, que des mesures sur une implantation logicielle de la préunification n'auront pas une grande signification vis à vis d'une implantation matérielle. En fait nous en tirerons deux enseignements :

- Les primitives intuitivement pressenties comme les plus importantes se sont bien révélées comme telles, ce qui confirme les gains substantiels que l'on peut espérer d'une réalisation matérielle.

- D'autres primitives ont pu être mises en évidence et, point singulier, n'ont pas une complexité plus grande, voire même bien inférieure. Pour ce dernier cas, la simulation a permis de mettre en évidence leurs fréquences, qui fait qu'elles ne sont pas aussi négligeables que prévu.

De plus ces mesures permettent de montrer qu'une implantation logicielle pourrait déjà s'avérer intéressante. Nous allons les résumer ci-après.

V.3.2.1. Performances logicielles

Nous fournissons ci-dessous un tableau récapitulatif des performances comparatives d'un interpréteur PROLOG et de l'algorithme d'unification, les deux étant écrits en C et tournant sur une SM90. Plusieurs paramètres ont été étudiés que nous précisons :

- Sélectivité: nous définirons ce paramètre comme le rapport du nombre de termes lus sur le nombre de termes acceptés. Ce paramètre est sans effet sur l'interpréteur PROLOG. En revanche pour le filtre, plus la sélectivité diminue, plus le temps d'une unification élémentaire augmente (de plus en plus d'associations sont effectuées).

- Substitutions: elles ont un effet considérable sur l'interpréteur PROLOG: il faut le double de temps pour réaliser une unification nécessitant une substitution que pour unifier un terme n'en nécessitant pas.
  L'effet est moins "violent" pour le filtre, mais tout de même sensible, et il faut environ un tiers de plus pour unifier un terme entraînant une substitution qu'un terme n'en provoquant pas.

- Complexité: La comlexité des termes se ramène au problème de la sélectivité.

Nous fournissons ici un tableau récapitulatif des résultats des jeux de tests les plus significatifs. Les mesures ont été effectuées sur deux types de termes avec deux sélectivités, sur des ensembles de cibles comportant de 1 à 100 cibles.

- Structure de type 1 : t (ai , X) -> ;
- Structure de type 2 : t (X , ai , Y) -> ;

Les ai sont des symboles tous différents. La base de données est constituée de 100 termes de structure 1 ou 2 avec un terme (constant ou non c'est sans importance car il n'y a pas de variables identiques dans les cibles) à la ou des positions de la ou les variables des buts (pour les mesures sur 100 cibles la base contient 200 clauses).

Les différentes lignes du tableau représentent les informations suivantes:
- La première ligne fournit la structure.
- La deuxième donne la sélectivité en pourcentage.
- Les suivantes, les temps moyens d'une inférence élémentaire pour le filtre, exprimés en milliseconde, en fonction de la cardinalité de l'ensemble de buts (nombres de la colonne de gauche).
- La dernière est constituée des mesures obtenues sur l'interpréteur PROLOG, le nombre de buts n'intervenant pas du fait de la séquentialité du traitement.

Tableau récapitulatif

	FILTRE			
	t (ai , *x)		t (*x , ai , *y)	
	1000	100	100	1000
1	-	280.0	560.0	-
2	-	145.0	300.0	-
3	-	100.0	200.0	-
4	-	81.0	155.0	-
5	-	67.0	130.0	-
10	33.5	41.3	71.2	62.7
20	20.6	29.3	45.3	35.5
30	16.7	26.8	36.5	25.8
40	14.3	22.4	31.8	19.8
50	13.2	21.4	28.3	18.1
100	8.0	14.6	19.7	11.0
	320.0	330.0	640.0	630.0
	PROLOG			

V.3.2.2. Primitives de base

L'intérêt de ces mesures d'un point de vue matériel réside en la confirmation de l'émergence des primitives définies. Pour un cas moyen, les temps d'exécution se décomposent comme suit:

Recherche : 20% à 30%
Et : 10% à 15%
Gestion sub. : 10% à 15%
Parcours : 5% à 10%
Genération : 5% à 10%

Ces mesures ont également permis de mettre en évidence deux autres opérations demandant une part non négligeable du temps d'exécution:

- La recherche d'une fin de clause en cas d'échec de la préunification: 10% à 15%.
  Cette opération élémentaire est répétée un nombre important de fois de par le taux d'échec élevé. Ce résultat offre des perspectives intéressantes en ceci que cette procédure pourra être

implantée aisément en matériel (simple comparaison pour rechercher un délimiteur). De plus sa rapidité d'exécution allonge le temps imparti aux cas de succés. Nous en discuterons plus en détail dans l'analyse des performances matérielles.

- Le contrôle: 10% à 15%.
  Il sera important de tenir compte de cette donnée, lors de la mise en oeuvre matérielle, principalement lors de la réalisation des différentes synchronisations et de la mise en oeuvre du parallélisme.

En rappelant les arguments du début, ces mesures sont cruciales en ce sens que la préunification assure la totalité du travail dans un fort pourcentage des cas, ce au vu du taux élevé d'échec. Néanmoins, il ne faut pas négliger le temps requis par l'association, qui dans le pire de cas peut être d'une complexité équivalente à celle de l'unification. Cette question étant important nous allons ici la développer.

V.4. Association

Nous avons décrit les principes et l'implantation de l'association lors du chapitre précédent. Rappelons simplement les deux objectifs de cette opération:

- Remettre en forme les solutions.

- Assurer, si nécessaire, la cohérence des substitutions.

Nous allons effectuer ici une classification des différents cas de figure possibles lors de la vérification de la cohérence, puis fournir différentes mesures.

V.4.1. Liaisons statiques ou dynamiques

Nous donnerons tout d'abord deux définitions, relatives aux liaisons entre variable d'un même terme:

- **Liaisons statiques:** on appellera liaisons statiques les dépendances issues des occurences d'une même variable au sein d'un terme.

  Exemple:

  t (X , terme , X)

- **Liaisons dynamiques:** on appellera liaisons dynamiques les dépendances créées par les liaisons de deux variables différentes d'un même terme part le biais d'un autre terme.

  Exemple:

  T1 : t (X , terme , X)
  T2 : t (Y , terme , Z)

  Une liaison dynamique sera créé dans T2 entre Y et Z au cours de l'unification avec T1.

En fonction de ces définitions, la cohérence des substitutions générées lors de l'unification de deux termes T1 et T2 n'est plus assurée dès l'apparition de liaisons statiques ou dynamiques. On va effectuer une classification, fonction de l'apparition de ces liaisons, et évaluer la fréquence et les

coûts de traitement de chaque cas. On considère qu'il y a toujours des variables dans les buts (sinon les opérations n'en seront que plus simples).

V.4.2. Classification et coûts

On effectue la classification suivante, en fonction du terme filtré:

- **Pas de variable**
  Dans un contexte de bases de données ce cas devrait s'avérer le plus fréquent. On peut alors le scinder en fonction des liaisons des variables du but atteint:

  * Pas de liaisons statiques
    Le seul travail consiste à la remise en forme, c'est à dire à une instanciation immédiate dans le contexte du but, opération d'un coût négligeable.

  * Liaisons statiques
    La procédure utilisée pour vérifier la cohérence est triviale car elle ne nécessite qu'une comparaison linéaire de deux termes du fait de l'absence de variables dans le terme filtré.

- **Variables**
  Ce cas concerne essentiellement les clauses ayant une queue, et sera donc nettement moins fréquent que le précédent. On peut de nouveau scinder en deux cas:

  * Pas de liaisons statiques ni dans le but atteint ni dans le terme filtré
    A nouveau la cohérence est assurée et le seul travail consiste à la remise en forme, se résumant à des instanciations immédiates dans l'un et l'autre des contextes concernés.

  * Les autres cas
    Tous les autres cas ne demandent pas un travail très complexe, sauf si des liaisons dynamiques sont induites, cas qui devrait être très rare dans notre contexte.

Les liaisons statiques des buts sont déterminées à la compilation, celle des termes filtrés lors de la préunification. De ce fait on peut directement sélectionner la procédure d'association appropriée la plus simple et donc la moins coûteuse. Dans tous les cas, le travail à effectuer sera d'un coût moindre que celui nécessaire à une unification "classique".

V.4.3. Mesures

Une synthèse est difficile en raison des domaines étendus dans lesquels peuvent varier les paramètres à prendre en compte:

- La sélectivité, qui si elle décroît trop, va entraîner un coût important de l'association dû à un nombre élevé d'appels.

- La complexité des termes substitués à des variables concernées par des liaisons statiques, qui demandera un effort d'autant plus important qu'elle est élevée.

- Le nombre de variables qui va accroître le temps d'exécution de toutes les procédures (ce qui est encore plus vrai pour l'unification classique).

Ceci étant les tests sur des exemples déjà pénalisants ont montré que l'accroissement du temps d'exécution n'excède pas 15%.

V.5. Conclusions

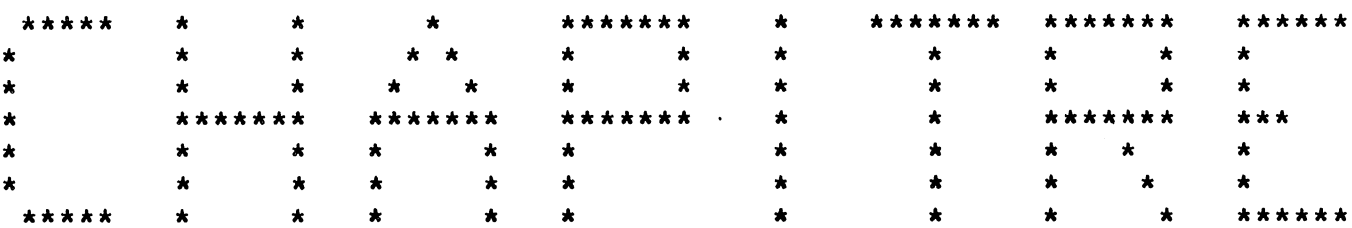
L'implantation logicielle a atteint ses deux objectifs:

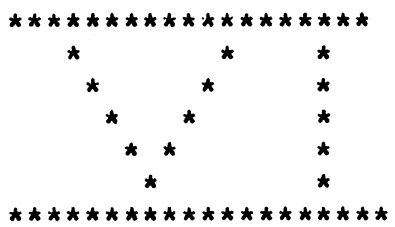
- La validation des algorithmes proposés.
- La mise en place des outils logiciels (compilation, association) nécessaires à la suite de l'étude.

De plus elle a permis:

- Une première approche de la simulation, qui s'est soldée par l'élaboration de spécifications plus complètes des primitives élaborées au chapitre précédent.
- Une évaluation des performances qui sugère de pousser la réalisation logicielle.

Avant de conclure sur cette première partie qui avait pour trait la validation de nos idées et les spécifications des algorithmes en découlant, nous présenterons, en regard des performances obtenues, une première retombée de ces travaux. Ces retombées s'expriment par la concrétisation logicielle de l'idée de filtrage et son intégration à un interpréteur séquentiel classique. Nous allons montrer, au cours du chapitre suivant, comment nous sommes parvenus à ces fins et les gains obtenus.





COOPERATION DES STRATEGIES

CHAPITRE VI

RESUME

Ce chapitre termine la partie logicielle de l'étude, en exposant l'intégration des algorithmes en une première maquette logicielle. La démarche consiste à implanter l'unification parallèle au travers de deux prédicats évaluables, et de réaliser une coopération entre interprétation parallèle et séquentielle. Nous discutons alors des performances obtenues avant de proposer une autre utilisation des algorithmes, toujours par le biais d'un prédicat évaluable, effectuant une recherche d'éléments dans une liste.

Chapitre VI

COOPERATION DES STRATEGIES

De prime abord l'idée d'intégrer le mécanisme d'unification exposé dans les chapitres précédents, au sein d'un interpréteur séquentiel, peut surprendre : quel intérêt de posséder un outil dont les possibilités ne semblent jamais devoir être exploitées, un interpréteur séquentiel ne manipulant qu'une solution à la fois ? Certes aucun, et nous en sommes conscients. Aussi notre idée est de mettre en oeuvre une coopération entre deux formes d'interprétation, l'une séquentielle, l'autre parallèle. La coopération permettra de choisir, en fonction du contexte, la méthode la plus appropriée. Nous exposons, au cours de ce chapitre, différents exemples où cette coopération permet un gain notable de performances.

Synthétiquement, les difficultés de mise en oeuvre portent sur les points suivants:

- L'interface interprétation séquentielle (solution par solution), interprétation parallèle (par ensemble de solutions) que l'on notera 'P -> S'.
- L'interface duale notée 'S -> P'.
- La gestion des ensembles de solutions.
- La modification de la procédure de "backtrack".
- La manipulation de "données externes".

Notons que les problèmes posés par la gestion des ensembles de solutions seront essentiels dans la réalisation de la machine OPALE, et les travaux effectués ici serviront de première approche.

Conceptuellement nous avons matérialisé la coopération en nous appuyant sur l'interprétation classique à laquelle nous avons

rajouté des prédicats évaluables dont l'évaluation utilise le mécanisme d'unification parallèle. Au cours de ce chapitre nous verrons tout d'abord deux prédicats, dédiés à l'interrogation de bases de données, et nous terminerons par un prédicat de "pattern", opérant sur une liste, montrant ainsi que le principe peut être utilisé dans divers contextes.

VI.1. Spécifications de deux prédicats BD

La mise en oeuvre des deux prédicats appelés Interrobas(X,Y) et Pipe(X,Y) se justifient par deux arguments:

- Dans le cadre du projet, ils sont une première approche (voire étape), de la machine cible.

- Dans une optique plus générale, ils fournissent le support d'une coopération efficace des deux méthodes d'interprétation, l'une séquentielle et l'autre parallèle.

Ces deux prédicats permettent d'interroger une base de connaissances "externe" (ce qui en PROLOG n'est ni courant ni immédiatement implantable). Interrobas (X,Y) autorise l'accès à un fichier alors que Pipe(X,Y) offre la possibilité de consulter plusieurs fichiers au sein d'une même requête. Nous développons ci-après les spécifications de chacun.

VI.1.1. Interrobas (X,Y)

Ce prédicat doit être considéré comme l'émission d'une requête ne portant que sur une seule relation de la base. Les deux arguments doivent respectivement être liés à :

- X : la liste des variables qui seront utilisées dans la suite de la clause d'appel.
- Y : une relation fournie sous la forme d'un terme où:
  . le symbole fonctionnel est le nom de la relation. les arguments sont interprétés dans le contexte de la clause d'appel.

Exemple:

```
Soit la clause:  Q (X,Y) -> interrobas (Y.Z , R1 (X,Y,Z,U))
                 écrit (Y)
                 écrit (Z) ;
```

Alors l'appel -> Q (a,X) ; génère, par l'intermédiaire d'interrobas, la requête R1 (a,Y,Z,U) vers le processus de filtrage. Dans la suite de la clause on pourra alors accéder à toutes les valeurs satisfaisantes de Y et Z, mais pas à celles de U. En terme de Base de Données, cet appel est équivalent à une opération de sélection suivi d'une opération de projection.

Vis à vis de l'utilisateur ce prédicat se comporte de la même façon que tout autre prédicat défini par lui-même. Il retourne une solution lors du premier appel puis, en cas de "backtrack", successivement toutes les solutions fournies par le processus de filtrage. En fin de solution, il retourne un échec.

VI.1.2. Pipe (X,Y)

A l'inverse du précédent, ce prédicat permet de spécifier une succession d'appel à la Base. Les deux arguments doivent être respectivement liés à:
-X: la liste des variables qui seront utilisées dans la suite de la clause d'appel.
-Y: un prédicat qui doit s'unifier avec un en-tête de clause, laquelle devra vérifier les conditions précisées après l'exemple.

Un exemple typique de ce cas d'utilisation est la jointure de deux relations :

```
Q (X,Y) -> pipe (X.Y , JOINT_R1_R2 (X,Y))
           écrit (X)
           écrit (Y) ;
JOINT_R1_R2 (X,Y) -> R1 (X,Z) R2 (Z,Y) ;
```

où R1 (X,Y) et R2 (X,Y) sont deux relations de la B.D.

Dans une première version, la queue de la règle doit être constituée uniquement de termes entraînant des appels à la B.D. Une extension sera la possibilité d'insérer des appels à des prédicats évaluables, voire même définis par l'utilisateur à condition que ces appels n'engendrent pas de ou-parallélisme.
De même que pour interrobas (X,Y), les variables apparaissant dans pipe (X,Y) seront interprétées dans le contexte de la clause d'appel. Les solutions sont transmises à l'interpréteur de la même façon que pour interrobas (X,Y).

VI.2. Coopération des stratégies

Dans l'introduction du chapitre, nous avons énoncé les différents points délicats de l'implantation. Nous allons décrire les solutions employées en nous attardant sur les problèmes de gestion des ensembles de solutions. En fait nous utiliserons des solutions simples (dont l'implantation pourrait être optimisée) qui mettent en évidence les différents points où des solutions plus efficaces sont envisageables.

VI.2.1. Les interfaces

Rappelons que la coopération des deux stratégies de recherche nécessite les interfaces:

-"S -> P" (Séquentielle -> Parallèle):
Cette procédure active le processus d'unification parallèle en lui fournissant d'une part l'espace disque ou doit s'effectuer la recherche (dans notre mise en oeuvre il s'agit essentiellement d'un nom de fichier), et d'autre part le contexte de la clause d'appel. Principalement ce contexte sera constitué des instances des variables liées et des informations quant aux projections et jointures à effectuer.

-"P -> S" (Parallèle -> Séquentielle)
L'interface duale est plus complexe et se trouve étroitement dépendante d'une part des modifications de la procédure de "backtrack", et d'autre part de la technique de représentation des substitutions. Elle intègre dans le monde de l'interpréteur séquentiel une nouvelle solution, ce à chaque sollicitation de la procédure de "backtrack". En cas de fin de solutions elle retourne un échec. Nous allons expliciter la technique employée pour intégrer une solution, issue de l'interprétation parallèle donc "externe", dans le contexte de l'interprétation séquentielle ne manipulant que des objets "internes".

VI.2.2. Intégration de "données externes"

Rappelons que notre interpréteur utilise un dictionnaire assurant le codage des termes sous format interne, et emploie une technique de "partage de stucture" pour représenter les substitutions. Dans un contexte Base de Connaissances, la taille du dictionnaire ne lui permet plus de résider en mémoire centrale. Une technique de pagination permet de le stocker sur disque, et de n'amener en mémoire que les pages requises à un instant de la résolution. Pour assurer la cohérence, quant à la représentation des substitutions, une solution est recopiée sous la forme d'une clause dans l'espace des clauses de l'interpréteur. Un espace tampon, réservé à cet effet, permet de stocker la solution courante à la place de la précédente, donc en l'écrasant.

VI.2.3. Gestion des ensembles de solutions

Les problèmes de gestion des ensembles de solution s'imbriquent en deux niveaux, l'un interne au prédicat, l'autre posé par l'enchaînement de plusieurs appels de prédicats base de données.

Pour ce qui est du niveau interne, Interrobas(X,Y) ne pose pas de difficultés en ceci qu'il ne génère qu'un seul ensemble de solutions, issu du seul appel à la base. Par contre Pipe(X,Y) manipule autant d'ensembles de solutions qu'il y a d'appels dans la règle activée (cf figure VI.1.). En effet son évaluation forme un pipe-line (d'où son nom !) où chaque station est composée d'un appel.

On peut définir plus précisément le fonctionnement du pipe-line régissant l'interprétation du prédicat Pipe(X,Y). Soit une station Si du pipe-line. Cette station utilise deux ensembles de solutions que l'on appellera commme suit :

-L'Ensemble Générateur, issu de la station précédente Si-1, et permettant à Si de générer les buts.

-L'Ensemble Courant dans lequel sont stockées les données issues de la station Si.

Soit l'appel:

```
clause ( ) -> ... , pipe (Liste , R ( )) , ... ;
```

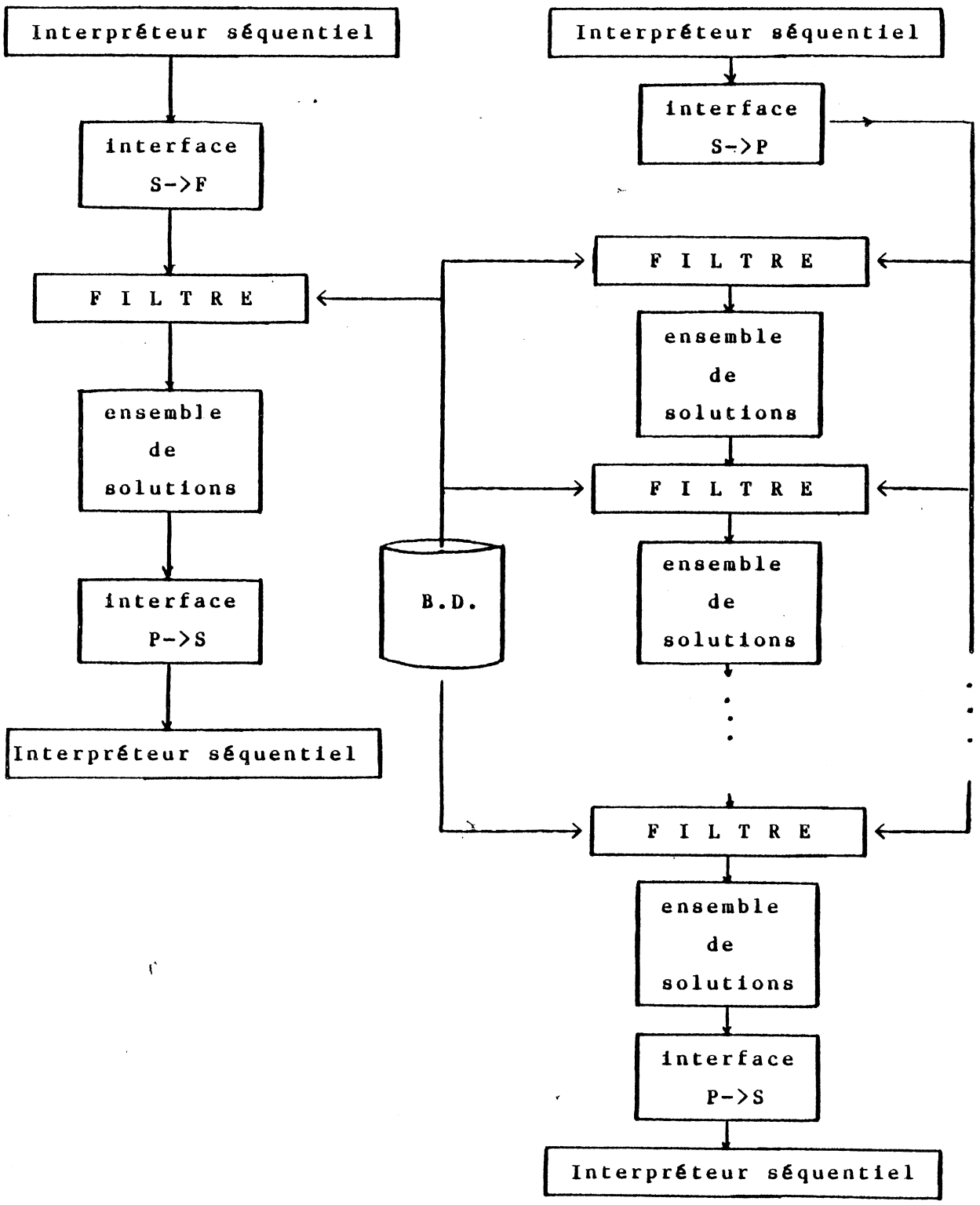


Figure VI.1

R ( ) -> R1 ( ) , ... , Ri ( ) , ... , Rn ( ) ;

Appelons FILTRE (R,C,EG) la fonction effectuant le filtrage de la relation R en fonction du contexte C et de l'ensemble générateur

EG, et retournant un ensemble de solutions EC. Les équations régissant le fonctionnement de Pipe (X,Y) sont alors les suivantes:

$$EC0 = 0$$

$$\text{Pour } 0 < i < n \quad ECi = FILTRE\ (Ri, Di, EGi-1)$$

$$\text{avec} \quad EGi-1 = ECi-1$$

On voit clairement ici les différents parallélismes à exploiter:

-Le premier lié à l'algorithme du filtre, donc efficient pour chaque station du pipe-line, transformant l'interprétation séquentielle "depth-first" en interprétation parallèle "breath-first".

-Le deuxième lié au pipe-line, dont cette implantation logicielle ne peut bénéficier pleinement pour des raisons bien évidentes.

La deuxième source de difficultés se situe au niveau des imbrications de plusieurs appels des deux prédicats dans une même clause appelante. En effet il faut alors gérer les différents ensembles de solutions issus des différents processus de filtrage, afin de préserver leur cohésion. Nous nous sommes confrontés ici à bon nombre de difficultés qu'il faudra résoudre pour aboutir à la machine OPALE. Les problèmes sont posés ... (les solutions utilisées pour cette implantation logicielle sont trop simples, pour être efficaces dans l'environnement réel d'une machine matérielle).

VI.3. Estimation des performances

L'évaluation des performances est délicate pour deux raisons:

-Les accès disques constituent plus de 50% du temps d'exécution. Ces délais ne sont pas significatifs en ceci que l'on passe par le système de gestion de fichiers de UNIX pour accéder physiquement au disque, ce qui entraîne une perte de temps non négligeable, mais difficile à chiffrer avec exactitude. Des mesures absolues semblent donc impossibles

-Pour ce qui est de mesures comparatives, peu, voire pas, d'interpréteur PROLOG permettent de manipuler efficacement des bases de données. La mise en oeuvre d'applications bases de connaissances, à l'aide de ces interpréteurs, leur demande un tel surcroît de travail que toute comparaison objective des temps d'exécution devient alors impossible.

Voici comment nous proposons de détourner la deuxième difficulté, délaissant pour l'heure la première.
Au vu des mesures effectuées sur l'algorithme d'unification seul, les temps d'exécution de l'algorithme classique d'unification et de l'algorithme "préunification parallèle et association" sont très proches lorsqu'un seul but est à unifier. Partant de ce résultat, on considère que le prédicat Interrobas(X,Y) peut servir de référence (il semblerait même meilleur de par, en cas de "retour arrière" la simplification de la procédure en une simple instanciation, au lieu d'une boucle complète de résolution sur une alternative).
On évalue alors la différence entre les deux interprétations en réalisant la même requête, soit par une succession d'Interrobas(X,Y), soit par un Pipe(X,Y).

Exemple: Soient les relations

```
Professeur ( Nom , Matière_enseignée , No_prof ) ;
Cours ( Nom_classe , Salle , Heure , No_prof ) ;
Elève ( Nom , Nom_classe ) ;
```

et soit la question:

Quels élèves suivent des cours de Latin ?

On mesurera les temps d'exécution des deux mises en oeuvre suivantes (équivalentes quant aux réponses). Pour distinguer plus facilement les variables on les préfixera par une "*" plutôt que par une majuscule:

```
R1 -> interrobas ( *no_prof , Professeur (*nom,Latin,*no_prof))
      interrobas ( *nom_classe , Cours (*nom_classe,*s,*h,*no_prof))
      interrobas ( *nom_élève , Elève (*nom_élève,*nom_classe))
      écrit (*nom_élève) ;

R2 -> pipe ( *nom_élève , R (Latin,*nom_élève))
      écrit (*nom_élève) ;
  avec:
      R (*matière,*nom_élève) -> Professeur (*nom,*matière,*no_prof))
                                 Cours (*nom_classe,*s,*h,*no_prof))
                                 Elève (*nom_élève,*nom_classe)) ;
```

L'interprétation de la première forme de la solution sera pratiquement identique à une interprétation classique, l'unification ne portant que sur un seul but pour chaque appel. Le "retour arrière" de l'appel d'un Interrobas(X,Y) "fils", relance son exécution avec la solution suivante issue du "père".

Pour la deuxième solution, l'unification parallèle jouera une première fois lors de la recherche sur la relation Cours, en retournant, pour tous les professeurs de Latin sélectionnés par le balayage de la relation Professeur, les classes associées. Puis l'algorithme fonctionne encore à plein pour fournir tous les élèves appartenant à ces classes, ce lors du balayage de la relation Elève.

Les écarts des temps d'exécution, relevés entre les deux formes de la réponse, s'avèrent nettement plus élevés que ceux obtenus lors des mesures effectuées sur les deux algorithmes seuls; a ceci la raison suivante:

-L'utilisation du prédicat Pipe(X,Y) réduit l'espace de recherche. En effet dès que plusieurs solutions sont retournés par l'un des appels cela entraîne une multiplication des balayages du fichier dans le cas d'Interrobas(X,Y) alors que pour Pipe(X,Y) un seul balayage suffit, tant que la capacité du filtre n'est pas atteinte. Aussi les performances semblent encore meilleures que celles obtenues lors des mesures de l'algorithme seul, mais il faut malgré tout modérer l'enthousiasme, en fonctions des remarques formulées quant aux accés des espaces disques balayés.

Objectivement il est difficile de faire la part de l'amélioration liée à l'algorithme lui-même et celle due à la diminution de l'espace de recherche, cette phase étant excessivement pénalisante dans notre mise en oeuvre. En effet il est certain que les temps d'accès et de balayage des fichiers peuvent être notablement réduit, et par voie de conséquences il en serait de même de l'écart entre les deux implantations. Ceci dit, même un balayage plus rapide ne supprimera pas l'écart lié aux répétitions.

VI.4. Prédicat Pattern(X,Y)

Les principes de logique sous-jacents à la mise en oeuvre de l'algorithme, laissent supposer que l'on peut appliquer le mécanisme (ou de légères variantes) à la résolution de toute une classe de problèmes. En effet le moteur de l'algorithme est la recherche d'un élément dans une liste, donc la réalisation d'un "ou" logique, puis d'une intersection de deux ensembles, donc un "et" logique. Un tel regard pousse intuitivement à utiliser cet algorithme pour évaluer des requêtes constituées de "et" et de "ou". Explorant cette voie, nous allons montrer comment le mécanisme permet, sans n'en rien changer, d'effectuer une recherche d'éléments dans une liste, opération de base dans bien des applications d'Intelligence Artificielle.

VI.4.1. Spécifications

On définit un prédicat évaluable Pattern(X,Y) à deux arguments:
-X est une liste représentant l'ensemble des éléments recherchés.
-Y est une liste représentant le domaine de recherche.

L'appel du prédicat Pattern(X,Y) retourne un succès si un des éléments de la première liste appartient à la deuxième, sinon un échec.
Un exemple immédiat d'application est celui de la fonction membre(X,L) où 'X' est un élément et 'L' une liste d'éléments. Elle s'écrit tout simplement:

```
membre(X,L) -> pattern(X,L) ;
```

VI.4.2. Implantation

L'idée de l'implantation réside en ce que la recherche d'un élément 'e' dans une liste 'L' peut-être décomposée en n requêtes, chacune vérifiant si le n_ième élément de la liste est l'élément 'e'. Si on peut évaluer ces n requêtes en parallèle, la simple lecture de la liste permettra de déterminer si l'élément appartient ou non à cette liste.
L'extension à la recherche de plusieurs éléments est immédiate en cela qu'il suffit de créer un ensembles de requêtes pour chacun des éléments recherchés.

La mise en oeuvre consiste à générer l'ensemble de "patterns", puis d'activer l'unification de chacun avec la liste constituant le domaine de recherche.

Par exemple pour vérifier si 'a' appartient à la liste L, possédant trois éléments ou moins, on essaiera d'unifier les trois "patterns" suivants avec L:

1: a.X0
2: X0.a.X1
3: X0.X1.a

L'utilisation de l'algorithme d'unification est alors possible moyennant une limitation liée au nombre de buts que l'on peut rechercher en parallèle. Celui-ci détermine le nombre d'éléments acceptés lors d'une activation, ce en fonction du nombre d'éléments de la liste de recherche. Cette limite pourrait être reculée moyennant une légère modification de l'algorithme de base.
Soulignons qu'il n'est pas nécessaire de connaître la cardinalité du domaine de recherche. La recherche s'effectuera sur les n premiers éléments de la liste où:

n = Max buts acceptés par l'algorithme / Nb éléments recherchés

Pour une version logicielle, et moyennant quelques modifications, le maximum de buts unifiables en parallèle peut facilement s'élever à plusieurs centaines. La version proposée permet de préciser la longueur de la recheche, c'est à dire le nombre d'éléments de la liste représentant le domaine de recherche, qui seront effectivement pris en compte, les autres étant ignorés.

VI.4.3. Exemple

Un exemple typique d'application est la recherche de mots-clés sur un des constituants d'un n-uplet d'une relation. Nous fournissons ci-dessous l'implantation en "pur PROLOG", puis celle obtenue à l'aide du prédicat Pattern(X,Y).

Soit la relation: Ouvrage(Titre,Mot-clé,No_ouvrage) ;
Soit la question: Quels sont les titres des ouvrages traitant de (ayant pour mots-clés):

trou_noir ou quasar ou pulsar ou étoile_à_neutrons

La réponse en "pur PROLOG" s'écrit, en préfixant les variables par des "*":

```
R1 (*liste_mc) ->
        interrobas (*titre.*mot_clé , ouvrage(*titre,*mot_clé,*n))
        membre_ou (*liste_mc , *titre)
        écrit (*titre) ;

membre_ou ([*x,*sliste] , *l) ->  membre (*x,*l) / ;
membre_ou ([*x,*sliste] , *l) ->  membre_ou (*sliste,*l) / ;
membre_ou ([] , *l) ->  échec ;

membre (*x , [*x,*sliste]) -> / ;
membre (*x , [*y,*sliste]) -> / membre (*x,*sliste) ;
membre (*x , []) -> échec ;
```

La réponse en utilisant le prédicat de Pattern s'écrit:

```
R2 (*liste_mc) ->
        pattern (*liste_mc , *mot_clé)
        interrobas (*titre , ouvrage(*titre,*mot_clé,*n))
        écrit (*titre) ;
```

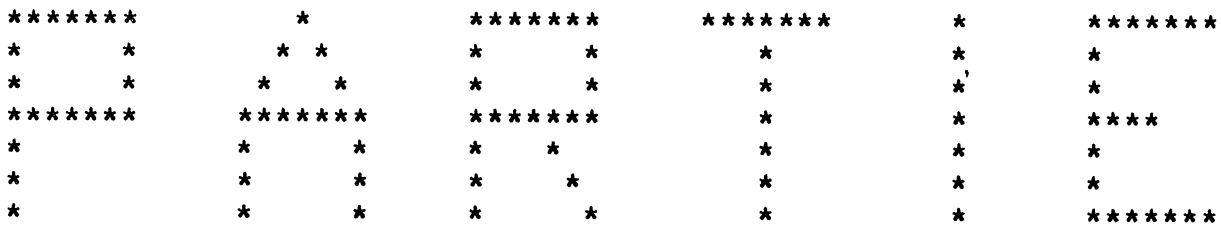
Au vu des implantations et sachant que le prédicat Pattern(X,Y) ne fait que générer des buts, opération peu coûteuse s'il en est, on se persuade aisément que si le prédicat Interrobas(X,Y) est implanté matériellement le coût des deux réponses ne pourra être comparable.

VI.5. Conclusions

Au long de ce chapitre nous avons montré que la coopération des deux méthodes d'interprétation l'une séquentielle, l'autre parallèle est d'une part possible, et d'autre part efficace ce à l'aide de l'algorithme d'unification développé au chapitre précédent. Cette implantation confirme les espérances de gain que doit apporter une réalisation matérielle et fournit de plus un outil logiciel efficace.

Pour une implantation encore plus efficace un effort est à fournir quant à la gestion des ensembles de solutions, problème que nous retrouverons lors de la mise en oeuvre de la stratégie de la machine complète. Pour l'heure nous en resterons à ce stade du développement logiciel, l'objectif final étant une réalisation matérielle.

Au terme de cette deuxième partie, on estime qu'une valorisation pourrait être concrétisée sans gros efforts supplémentaires, sous la forme d'une maquette logicielle plus complète du processeur disque. En effet les mesures effectuées permettent d'envisager un gain notable de performances, même avec une implantation logicielle de l'algorithme de préunification. Pour l'heure nous allons poursuivre l'étude en abordant la phase matérielle ayant trait aux spécifications des différents composants.



SPECIFICATIONS MATERIELLES

RESUME

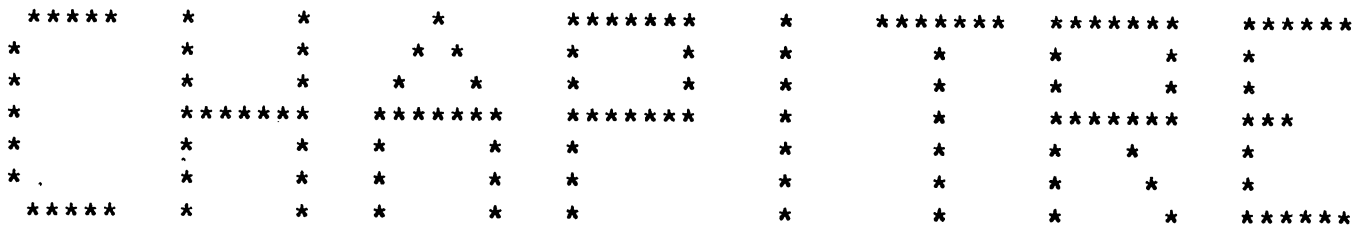
Cette troisième et dernière partie concerne l'aspect matériel du sujet. Les algorithmes étant spécifiés et validés d'un point de vue fonctionnel, il s'agit dès lors de procéder aux mêmes travaux en vue d'une implantation matérielle. Pour ce faire nous aurions pu suivre une démarche habituelle, c'est à dire guider la conception en fonction d'un modèle d'architecture classique, "style" microprocesseur, ayant une partie contrôle et une partie opérative. Ce modèle ne respectant pas la philosophie jusqu'alors adoptée pour spécifier le filtre, nous proposons d'élaborer un autre modèle, et par la même une aute approche, permettant d'exploiter le parallélisme exprimé dans un algorithme.

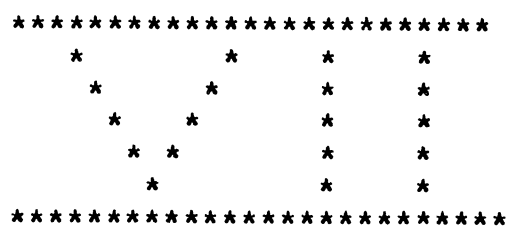
Le chapite VII débute par une réflexion sur l'écart actuel séparant les fonctionnalités des machines et les nécessités des applications, la conclusion étant qu'il est déjà par trop important. Nous passons en revue les différents modèles d'architecture "non Von-Neumann" essayant d'apporter une amélioration de cet état de fait, modèles tentant essentiellement d'exploiter le parallélisme ou de réaliser des coopérations de processeurs spécialisés. Nous discutons alors des principaux choix à effectuer dans l'élaboration de ces modèles.
Ceci posé, nous proposons alors une autre approche, guidée par les concepts issus de la programmation orientée objet, basée sur la notion de module et de communications par envoi de messages. L'objectif recherché est d'implanter physiquement une coopération efficace de modules autonomes. On aboutit ainsi à l'élaboration d'une machine modulaire.

Le chapitre VIII met en application la méthodologie amorcée au chapitre précédent, en proposant la réalisation du module disque (le processeur disque). Aussi nous décrirons rapidement ce module utilisant deux sous-modules: un module contrôleur disque et un module filtre. Délaissant le module contrôleur disque n'offrant pas de problèmes nouveaux, nous nous pencherons sur l'élaboration du module filtre. Ce dernier est lui-même composé de plusieurs sous-modules, quant à eux élémentaires, assurant les primitives de l'algorithme de préunification. Nous décrirons alors chacun d'eux

d'un point de vue fonctionnel et architectural. Nous définirons également un protocole de communications par messages le plus simple possible. Pour finir nous justifierons cette mise en oeuvre par une évaluation de la fréquence de ces messages.

Le dernier chapitre aborde la phase de conception au niveau VLSI. Pour ce faire, une simulation plus proche du matériel est proposée. Cette simulation englobe la notion de temps permettant de valider la nouvelle structure de l'algorithme basée sur le protocole de communication assumant les envois et réceptions de messages. De même on valide également la notion d'états d'un module, introduite pour mettre en oeuvre le protocole (pertes de messages, attente d'émission...). Ceci établi, nous présentons les différents outils développés au sein de la méthodologie de conception CAPRI, ayant pour objectif la réalisation d'un "compilateur de silicium". Nous verrons successivement le langage de description, l'extracteur des actions opératives et actions contrôles, les générateurs correspondants et enfin le simulateur. Nous suggérons alors d'intégrer séparément chaque module, en vue de la réalisation d'une première maquette. Nous développons pour exemple les résultats obtenus par la description "hardware" du module de recherche.





ARCHITECTURE ORIENTEE MODULE

CHAPITRE VII

RESUME

Ce chapitre fournit tout d'abord une revue des différents modèles d'architecture existant, la plupart basée sur une exploitation du parallélisme. Suite à une analyse des résultats obtenus par ces architectures, nous exposons les principaux choix à effectuer lors de la définition d'une architecture parallèle. Nous proposons alors une approche modulaire, basée sur les concepts logiciels de la programmation orientée objet. Nous essaierons d'appliquer ces idées à la définition d'une méthodologie de conception d'une machine modulaire, après avoir défini la notion de module.

Chapitre VII

AOM : Architecture Orientée Module

Comme énoncé en introduction, l'évolution des machines est freinée plus par des problèmes de conception, que de réalisation technique. Ainsi, si désormais l'intégration d'un million de transistors sur un circuit VLSI est envisageable, leur agencement en un circuit complexe demande un effort de recherche et de validation de plusieurs hommes années. De même, la conception d'une machine nécessite un travail considérable.

Dans le domaine des machines, les difficultés avaient pour source, de l'avis général, l'accès à la mémoire, autrement nommé "goulot d'étranglement de Von Neumann". Nombre de tentatives ont échoué, quant à la définition d'architectures "non Von Neumann", et l'on peut se demander si le problème se situe réellement en ce point. Il semble plutôt que cette structure soit l'une des composantes fondamentales d'une machine, seule son utilisation pouvant être modifiée. Par analogie, est-ce qu'il viendrait à l'esprit de vouloir modifier la représentation et la manipulation binaires de l'information, au sein d'une machine? Pourtant un ordinateur traite sans difficultés des caractères, ceux-ci étant codés sous forme numérique pour satisfaire aux impératifs physiques.

La mise en oeuvre des circuits VLSI se heurte, quant à elle, à sa prope puissance. En effet, sous peine de ne plus maîtriser la complexité des circuits, les premières réalisations ont copié l'architecture des machines "discrètes" pour en réaliser des versions "intégrées". Si des gains significatifs en ont découlé, aucune évolution de l'architecture elle-même n'a été enregistrée. Pire, un nouveau "goulot de complexité" a été mis en évidence, par la distinction entre partie opérative et partie contrôle.

Notre sentiment est qu'un fossé n'a cessé de croître entre les primitives de base du matériel, n'ayant pratiquement pas évolué, et les applications, ayant quant à elles dépassé l'imagination des premiers concepteurs. De ce fait, de plus en plus de couches intermédiaires sont empilées entre matériel et applications finales, les pertes de performances étant dès lors inévitables.

Si la recherche d'amélioration des performances technologiques est sans nul doute nécessaire, elle n'est plus suffisante. En effet, au vu des exigences nouvelles des applications, il semble qu'il devient impératif de définir des primitives matérielles plus sophistiquées que celles élaborées jusqu'ici, afin de supprimer un certain nombre de couches intermédiaires. Pour ce faire, nous proposons la conception de machines modulaires, organisée en une architecture hiérarchique, chaque "branche" de la machine assumant une fonction spécifique, supportée par des "feuilles" matérielles adaptées.

VII.1. Nécessités et moyens

Une machine classique est construite autour d'une unité centrale accédant à une mémoire, et de diverses unités d'Entrées/Sorties. Il découle de cette organisation un cycle machine type, constitué de la recherche de l'instruction et de ses opérandes, de l'exécution de cette instruction et finalement du stockage du résultat. Une instruction d'un langage type PASCAL s'adapte évidemment bien à un tel mécanisme, la définition du langage ayant été influencée par ce mécanisme !... Par contre, l'exécution d'une "instruction" de type IA, que l'on peut schématiser par une inférence pour le langage PROLOG, ne se réalise qu'au terme d'une séquence d'instructions machines dont le nombre moyen est toujours supérieur d'au moins deux ordres de grandeur à celle nécessaire à la réalisation d'une instruction de type PASCAL. Pour exemple, nous citons les mesures effectuées sur notre interpréteur PROLOG, mis en oeuvre sur le micro-processeur MC 68000. Typiquement, une unification PROLOG demande 0.1 à 1 ms, ce qui signifie de l'ordre de 100 à 1000 instructions machines. Une instruction PASCAL nécessite en moyenne de 1 à 5 instructions du 68000. Pour éviter toute polémique, nous précisons tout de suite que :

- Les chiffres fournis ne doivent en aucun cas être pris pour "argent comptant", car ils dépendent de trop nombreux paramètres pour être fournis avec autant de précision. Leur seule prétention est de fournir un **ordre de grandeur** .

- Nous ne comparons pas des coûts, car il faudrait évaluer les traitements effectués au terme de chaque instruction (IA ou PASCAL), mais **l'adéquation** des machines actuelles à exécuter tel ou tel langage.

VII.1.1. Essais de parallélisme

Malgré, l'adéquation des langages de type PASCAL aux "machines Von-Neumann", et vice-versa, les temps de traitement des algorithmes numériques, sont rapidement devenus prohibitifs. Une issue a été entrevue dans l'exploitation du parallélisme lié a une application. Nous résumons les principales mises en oeuvre matérielles

ayant vu le jour :

- Les machines "pipe-line" :
  La technique du "pipe-line", première approche du parallélisme, ne s'est montrée avantageuse que pour certaines classes de problèmes, domaines trop restreints pour le coût de telles machines (Cray1, Cyber205). Actuellement cette technique est utilisée comme complément (efficace), mais plus comme support principal.

- Les tableaux de processeurs :
  Ces machines sont formées d'une collection de processeurs séquentiels classiques, possèdent un contrôle centralisé et une interconnection fixe des différents processeurs. Là aussi le gain de performances n'est appréciable que pour des applications bien spécifiques (ILLIAC 4, ICL).

- Les multi-processeurs :
  Il s'agit d'un ensemble d'unités partageant une mémoire commune, interconnectées par un réseau. Chaque unité est soit un processeur (classique ou vecteur) soit un groupe de processeurs séquentiels (Cm*, Cray2). Pour l'heure, cette architecture est la plus développée.

- Les réseaux systoliques :
  Cette solution, issue de la technologie VLSI, comporte un ensemble de processeurs élémentaires identiques, dont la fonction et l'interconnection sont spécifiques à chaque application. Bien entendu ceci n'est utilisable que pour certains circuits (arithmétiques ou autres), mais non pour une machine dans son ensemble.

Ce résumé permet d'évoquer les résultats peu satisfaisants et trop spécifiques des réalisations actuelles (nous ne mentionnons pas la technique "data flow" de part le manque d'aboutissement commercial). A noter que ces résultats se dégradent complètement si l'on sort du domaine numérique. Nous proposons l'analyse de ces échecs et l'avancement des recherches actuelles.

VII.1.2. Analyse des échecs

Aussi important que soit le développement des architectures parallèles, il ne faut pas pour autant que celles-ci imposent sans conditions leurs exigences, sous peine de réitérer les erreurs commises lors du développement des machines séquentielles. Ceci semble maintenant largement admis et les efforts se tournent également vers la définition des concepts, et donc des besoins, d'une algorithmique parallèle. Devant la complexité et l'ampleur de la tâche, les premières recherches s'efforcent d'instaurer une classification des différents algorithmes. Les principales difficultés se résument d'un point de vue logiciel en les questions suivantes :

- Comment extraire et exprimer le parallélisme ?
- Sur quels critères se baser pour établir une classification ?
- Jusqu'à quel point peut-on, et doit-on, faire abstraction de la machine utilisée ?

et d'un point de vue matériel se situent en les choix principaux suivants :

- Utilisation de processeurs universels ou spécialisés ?
- Communications intensives ou mémoire partagée ?
- Contrôle centralisé ou réparti ?
- Interconnection statique ou dynamique ?

Au cours de ce chapitre nous préciserons quelques uns des critères influençant les choix matériels. Pour l'aspect logiciel nous renvoyons le lecteur aux différentes communications publiées dans [COMP 84], car une discussion même partielle de ces questions, nécessiterait un ouvrage complet.

VII.2. Architecture parallèle

Tout ce qui nous entoure peut-être représenté sous la forme d'un assemblage de systèmes se décomposant récursivement en sous-systèmes jusqu'à atteindre une "brique de base". L'informatique, non seulement n'échappe pas à cette règle, mais en a fait l'un de ses fers de lance. Nous en citons pour exemple les aboutissements les plus retentissants : la programmation structurée, les niveaux d'interprétations d'un langage, la CAO ... Seule l'architecture résiste encore à cette modélisation, une machine se décomposant conceptuellement en un seul niveau en une partie contrôle et une partie opérative.

Une telle organisation n'offre guère de possibilités de parallélisme, la seule envisageable consistant en une "anticipation" de certains traitements, technique qui s'est matérialisée sous forme de machines "pipe-line".

Les recherches actuelles proposent d'utiliser la machine classique comme "brique de base". Dès lors la possibilté de parallélisme devient évidente chaque brique pouvant travailler en simultanéité avec les autres. Se posent alors les choix évoqués au paragraphe précédent:

- "Briques" universelles ou spécialisées:
  En faveur de briques universelles, on peut argumenter sur le fait que n'importe quelle brique inoccupée peut effectuer n'importe quelle tâche. Naturellement, on peut défendre la spécialisation par les gains de performances espérés. Par ailleurs des briques homogènes seront plus faciles à interconnecter que leurs duales, qu'elles soient universelles ou spécialisées.

- Communications ou mémoire partagée:
  Ce choix n'est évidemment pas aussi délimité, les deux techniques pouvant coexister. Les critères importants sont, pour les communications, les coûts qu'elles entraînent, et pour le partage de mémoire, l'incompatibilité "naturelle" avec une volonté de parallélisme. Ce point détermine les lignes directrices de l'architecture de la machine, d'où son importance.

- Configuration statique ou dynamique:
  La configuration statique est de toute évidence la plus simple à mettre en oeuvre, les connections en découlant étant immédiates. Par contre, offrir au système la potientalité de se reconfigurer dynamiquement, signifie une machine d'autant plus apte à utiliser pleinement ses ressources. La configuration peut être soit automatique, et ainsi régulée en fonction de l'évolution des différents processus existant à un instant donné, soit effectuée en fonctions d'indications fournies par l'utilisateur. La mise en oeuvre de tels mécanismes est d'une part, plus coûteux en ressource matérielle et d'autre part, plus complexe à implanter.

- Contrôle centralisé ou réparti:
  Le contrôle peut-être soit décentralisé, ce qui assure une autonomie complète des différents éléments, mais en revanche induit une synchronisation délicate, soit être centralisé, ce qui règle ces problèmes mais supprime une partie de l'autonomie.

Face à ces parallélismes extrêmes, l'un fourni par un assemblage de processeurs, l'autre concentré en une technique pipe-line, nous préconisons une autre voie qui répartit le parallélisme en plusieurs niveaux, en permettant une exploitation plus fine.

Par l'introduction de plusieurs niveaux, on espère non seulement un gain significatif de performances, mais également des éléments de réponses aux questions précédentes. Pour ce faire, nous allons nous rapprocher de l'algorithmique au travers des principes issus de la Programmation Orientée Objet, qui a le double avantage de représenter les besoins actuels et d'exprimer implicitement des notions de parallélisme.

VII.3. Approche modulaire

L'engouement actuel pour la Programmation Orientée Objet est identique à celui exercé, il y a quelques années, par l'apparition de la programmation structurée. Les objectifs sont d'ailleurs identiques, si ce n'est que cette méthodologie de programmation a l'ambition de fournir les moyens nécessaires à la résolution de problèmes de complexité plus grande. Notre pensée est d'utiliser certains de ces concepts, à la définition d'une architecture, en regard des réflexions suivantes:

- La Programmation Orientée Objet permet de dégager un parallélisme "naturel" même si tous les Langages Orientés Objets ne le supportent pas, à l'inverse de la plupart des Langages Acteurs.

- Elle offre une méthodologie pour maîtriser ce parallélisme, et plus généralement la complexité des applications.

- Les principes logiciels semblent avoir une dualité matérielle.

Ceci dit, nous n'envisageons pas de concevoir une "machine langage orienté objet", à l'instar des machines PASCAL, LISP, voire PROLOG. En effet vouloir transposer l'ensemble des concepts en conservant leur philosophie nécessite de toute évidence un travail substantiel. Aussi nous nous contenterons de nous inspirer de certaines idées de ce style de programmation, en vue de définir une méthodologie appliquable à la conception d'une architecture.

VII.3.1. La Programmation Orientée Objet

La Programmation Orientée Objet a pour concepts générateurs les notions d'objet et de classe, introduites historiquement pour la première fois par [BIR 73]. Le langage SIMULA en fut une première concrétisation. Suivit le langage SMALLTAK [ING 78] qui assura définitivement leur vulgarisation.

Depuis différentes extensions ont été (et sont) proposées. Le lecteur intéressé trouvera nombre d'articles dans [BIG 83] et [BIG 84].

Pour notre part, nous ne rappellerons ni les concepts

génériques, ni les extensions les plus évoluées mais seulement les notions actuelles les plus communément partagées. Voici les définitions, assorties de quelques commentaires, nécessaires à la compréhension de la suite :

- Un **objet** associe de manière insecable données et procédures d'exploitation. Ces dernières sont appelées **méthodes** et leurs noms sont appelés **sélecteurs.**

- Une **classe** est un **modèle** d'objets en ce sens que l'instanciation d'une classe par un ensemble de données fournit un objet. Une classe permet donc de regrouper (concept de générique) les objets de description et comportement semblables, c'est à dire ayant la même structure de données et les mêmes procédures d'exploitation.

- Un **message** est constitué d'un nom de **destinataire** , d'un **sélecteur** et éventuellement de **paramètres** . Le destinataire doit permettre d'identifier un ou plusieurs objets, et, le sélecteur désigne, en fonction des points d'entrée, la méthode à activer.

Les règles suivantes fournissent les concepts de la Programmation Orientée Objet, qui peut-être vue comme une approche de programmation (et en ce sens ne pas nécessiter obligatoirement un langage, mais alors subir les incommodités que cela entraîne; on peut essayer d'écrire un système expert en BASIC...):

- Toute **entité** manipulée est un **objet.**

- Tout objet est l' **instance** d'une classe. De plus toute **classe** est **sous-classe** d'une autre, une "méta-classe" étant sous-classe d'elle-même. Ceci permet d'établir une hiérarchie des classes et d'introduire la notion d'héritage, toute sous-classe héritant des propriétés (structure des objets) et méthodes de sa sur-classe.

- Toute **communication** s'effectue par **envoi de messages** . Cette règle induit notamment qu'une méthode ne peut directement en activer une autre. Pour ce faire elle doit passer par l'intermédiaire d'une émission de message, qui devient ainsi l'unique moyen de contrôle.

Nous arrêterons notre description à ce niveau de détail, pour

conserver la modularité qu'il en ressort. Nous allons maintenant nous attacher à extraire le parallélisme lié à cette modélisation.

VII.3.2. Parallélisme "naturel"

Des notions de parallélisme se dégagent implicitement de cette approche. En effet chaque objet est doté des moyens nécessaires à l'exécution autonome d'un traitement. L'idée immédiate est de profiter de cette faculté pour activer simultanément plusieurs objets. Cette possibilité est d'ailleurs de plus en plus développée dans les langages, de part l'évolution des envois de messages.

Initialement, un message désignait un destinataire unique, et généralement attendait une réponse retournée par le message. Ce contrôle essentiellement séquentiel, s'est avéré beaucoup trop rigide vis à vis des objectifs de la Programmation Orientée Objet. Aussi certains langages, et notamment LRO (Langage de Représentation des Objets) [ROC 85] proposent des extensions permettant d'envoyer un message à une classe, message qui sera alors transmis à tous les objets instances de cette classe. Mieux encore, un message peut-être émis sans aucune indication quant à son destinataire, tout objet étant un récepteur potentiel. Le sélecteur du message joue alors le rôle de filtre vis à vis des points d'entrée (sélecteurs) des méthodes des différents objets. Il est clair que plusieurs objets peuvent ainsi être activés simultanément, tout au moins d'un point de vue "logique", car concrètement, les outils logiciels, ainsi que les machines, ne le permettent pas.

VII.4. Transposition matérielle

Notre point de départ, quant à l'utilisation des concepts énoncés précédemment, réside en la matérialisation de la notion d'objet. L'idée sous-jacente est de pouvoir exploiter le parallélisme "naturel", plusieurs objets matériels distincts pouvant être activés simultanément. Dans cette optique, la transposition immédiate d'un objet logiciel en un objet matériel n'est pas possible, de par le partage des méthodes attachées à une classe. En effet, si dans un contexte logiciel, un tel partage ne pose pas de problème (parallélisme non exploité, ou le cas échéant, code réentrant), des objectifs matériels ne peuvent le supporter. Nous regroupons donc les notions de classe et d'objet en une seule et même entité matérielle, que nous appellerons **module** . Il est clair qu'une telle décision nous éloigne , voire nous sépare, de certains concepts des Langages Orientés Objet, mais nous la justifions en rappelant que nous ne cherchons pas à réaliser une machine Langage. Nous reviendrons sur ce point dans la conclusion de ce chapitre, et abordons pour l'heure la définition d'un module élémentaire.

VII.4.1. Module élémentaire

Du point de vue de l'architecture (Fig VII.1.), un module est conforme à un "objet logiciel", et en ce sens il possède:

- Un modèle de données.
- Des méthodes opérant sur ce modèle.

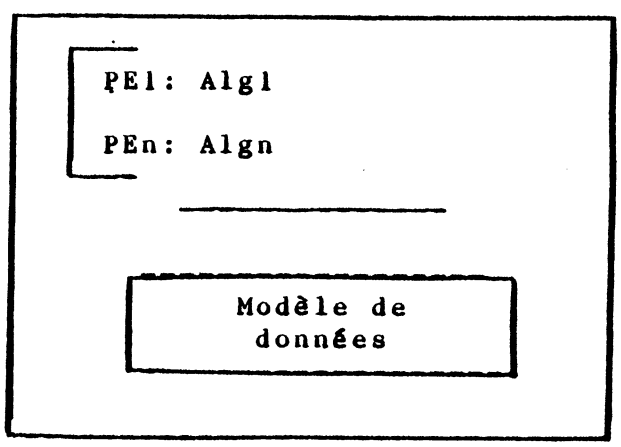


Figure VII.1

Une méthode se scinde en deux constituants:
- Un point d'entrée (équivalent au nom d'une procédure).
- Un algorithme (équivalent au corps de la procédure).

Un module élémentaire se distingue, à la fois par son modèle de données, composé uniquement d'éléments de mémorisation (non nécessairement homogènes), et par ses méthodes, dont les algorithmes sont bâtis uniquement à l'aide d'instructions élémentaires.

Le comportement d'un module élémentaire est également comparable à celui d'un "objet logiciel". Il s'active sur la réception d'un message dont le sélecteur correspond à l'un des points d'entrée de l'une de ses méthodes. Bien entendu, un module élémentaire peut émettre des messages au cours de l'exécution de l'algorithme de la méthode activée.

Un message est formé d'un sélecteur et éventuellement d'un ensemble de données. Dans une première approche le destinataire, ne peut être fourni car cela semble contraire à une volonté de parallélisme. En effet l'émission d'un message doit permettre de faire abstraction du "qui" et du "comment", l'important étant que les requêtes, énoncées par le biais du message, soient satisfaites.

L'intérêt d'utiliser un module élémentaire pour "brique de base" réside en la possiblité de celui-ci d'effectuer un traitement local de façon autonome. Pour dégager le maximun de parallélisme, il est nécessaire de définir le plus de primitives (relativement) indépendantes, pouvant être exécutées sur des modules élémentaires distincts.

La "brique de base" de notre modèle étant définie, nous allons voir comment construire une machine, par l'agencement de plusieurs de ces briques.

VII.4.2. Module

La philosophie de la démarche, aboutissant à la réalisation de modules complexes, est la suivante : la coopération de plusieurs primitives, supportées par des modules élémentaires, permet de construire une fonction de plus haut niveau, qui pourra à son tour être intégrée en une fonction. On établit ainsi de manière récursive des fonctions de plus en plus sophistiquées exploitant le

parallélisme à tous les niveaux. Notons que nous retrouverons ainsi les notions de classe et de sous-classe...

Dans la suite nous appellerons module soit un module élémentaire soit un module complexe. Nous proposerons tout d'abord une définition plus formelle et discuterons ensuite de la réalisation et mise en oeuvre de ces modules.

VII.4.2.1. Définitions

Fonctionnellement un module complexe est identique à un module élémentaire : il s'active sur réception de message, effectue un traitement local, et éventuellement émet des messages. La différenciation apparaît à un niveau architectural en ceci qu'un module complexe possède:

- Un modèle de données complexe, lui permettant d'intégrer non seulement des éléments de mémoire, mais également un assemblage de sous-modules.

- Des méthodes complexes, pouvant faire appel aux méthodes des "sous-modules".

Nous définirons l'architecture d'un module comme suit :

```
<Module>                    ::= <Module élémentaire> /
                                <Module complexe>
<Module élémentaire>        ::= <Modèle de données élémentaire>
                                <Ensemble de méthodes élémentaires>
<Module complexe>           ::= <Modèle de données complexe>
                                <Ensemble de méthodes complexes>
<Modèle de données élémentaire>
                            ::= Eléments physiques de mémorisation.
<Ensemble de méthodes élémentaires>
                            ::= <Méthode élémentaire>
                                <Ensemble de méthodes élémentaires> /
                                Vide
<Modèle de données complexe>
                            ::= <Modèle de données élémentaire>
                                <Assemblage de modules>
```

```
<Ensemble de méthodes complexes>
                        ::= <Méthode complexe>
                            <Ensemble de méthodes complexes> /
                            Vide
<Méthode élémentaire>   ::= <Point d'entrée>
                            <Suite d'instructions élémentaires>
<Méthode complexe>      ::= <Point d'entrée>
                            <Suite d'instructions complexes>
<Suite d'instructions élémentaires>
                        ::= <Instruction élémentaire>
                            <Suite d'instructions élémentaires> /
                            Vide
<Suite d'instructions complexes>
                        ::= <Instruction complexe>
                            <Suite d'instructions complexes> /
                            Vide
<Instruction élémentaire>
                        ::= Instruction câblée /
                            Emission de messages
<Instruction complexe>::= <Instruction élémentaire> /
                            Appel à une "sous-méthode"
<Assemblage de modules>
                        ::= <Module>
                            Interconnection physique
                            <Assemblage de modules> /
                            Vide
<Point d'entree>        ::= Sélection physique des sélecteurs.
```

On peut visualiser modules élémentaire et complexe respectivement sur les figures VII.1 et VII.2. Ces définitions précisant l'environnement des modèles et méthodes complexes, nous allons pouvoir préciser les solutions matérielles envisagées, quant à leur implantation. A la suite de quoi, nous proposerons l'interprétation de ces définitions, vis à vis de la réalisation d'une machine.

VII.4.2.2. Modèle de données complexe

Suivant sa définition, un modèle de données complexe est composé de l'interconnection d'un ensemble de modules (le modèle de données élémentaire ne posant pas de problème, nous n'en parlerons pas). L'interconnection se scinde en deux entités, l'une physique l'autre logique :

- La définition et réalisation des chemins physiques de données:
  La solution la plus ouverte consiste à relier entre eux tous les modules par l'intermédiaire d'un bus commun auquel chaque module sera connecté.

- Le protocole de communication:
  Il est défini par l'émission et la réception des messages, leur gestion pouvant être assurée de différentes manières, ce en fonction des caractéristiques des messages (fréquence, synchronisation, longueur ...).

On peut se demander, a priori, si l'on ne va pas finalement se confronter aux problèmes de départ. On peut affirmer qu'il n'en est rien pour les raisons suivantes :

- Le nombre de modules à interconnecter, au sein d'un même modèle complexe doit être faible (<10). Si ce n'est le cas, il est sûrement possible, et même nécessaire sous peine de perdre tous les avantages de la méthode, de scinder le module englobant en plusieurs sous-module. Ceci assure que:
  . la connection par bus sera une solution satisfaisante.
  . le nombre de messages sera réduit ce qui autorise une mise en oeuvre simple de la gestion des messages, donc peu coûteuse.

- Un module assure une fonction bien spécifique, ce sur ses données locales. Ceci signifie que les modules ne partageront pas de mémoire, sans que cela entraîne un surcroît inacceptable de communications.

VII.4.2.3. Méthode complexe

Au sein d'un module, une méthode complexe (activée par un message) peut tenir l'un des deux rôles suivants (Fig VII.2): IP - 2 Soit elle assume la requête exprimée par le message, en déroulant son algorithme et en utilisant éventuellement le modèle de données complexe du module comme un "opérateur".

- Soit elle assure seulement la liaison entre l'extérieur du module et un des sous-modules en lui retransmettant le message, ainsi que les paramètres, la requête étant ainsi totalement "sous-traitée" par le "sous-module".

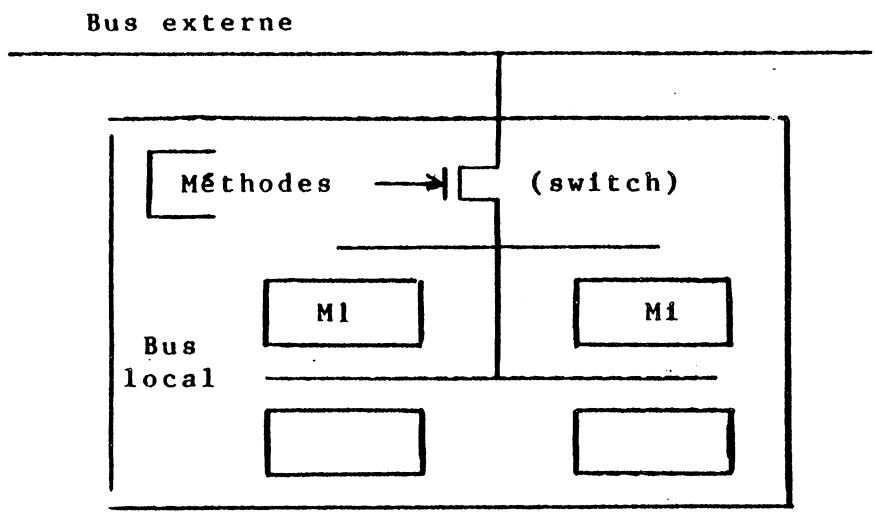


Figure VII.2

Dans le premier cas les mécanismes nécessaires sont définis au paragraphe précédent. Par contre, dans le premier cas, il suffit de permettre à la méthode d'accéder au bus connectant les sous-modules, sur lequel elle pourra émettre un appel à une sous-méthode par l'intermédiaire d'un message.

Pour le deuxième cas, la structure complexe du module doit être reliée au monde "extérieur" pour pouvoir récupérer d'éventuels paramètres. L'idée est de relier le **bus local** de l'assemblage de sous-module au **bus externe** connecté au module. Un "switch" coupe les deux bus et permet, suivant sa position, de les relier ou non (Fig VII.3). Dès lors, tout sous-module de

l'assemblage peut recevoir des données externes au module.

Si un seul niveau d'imbrication des modules (les modules d'un assemblage sont tous élémentaires) semble un objectif relativement aisé à atteindre, il est clair que la réalisation d'une hiérarchie plus complète ne sera pas aussi facile, comme nous allons le voir.

VII.4.3. Machine modulaire

Les principes exposés ci-dessus aboutissent à une architecture modulaire hiérarchique (Figure VII.3). En effet on peut voir la machine comme un arbre dont chaque noeud est constitué d'un module. Les transitions entre noeuds sont assurées par les bus des modèles des données complexes. La machine elle-même peut ainsi être considérée comme le module "racine".

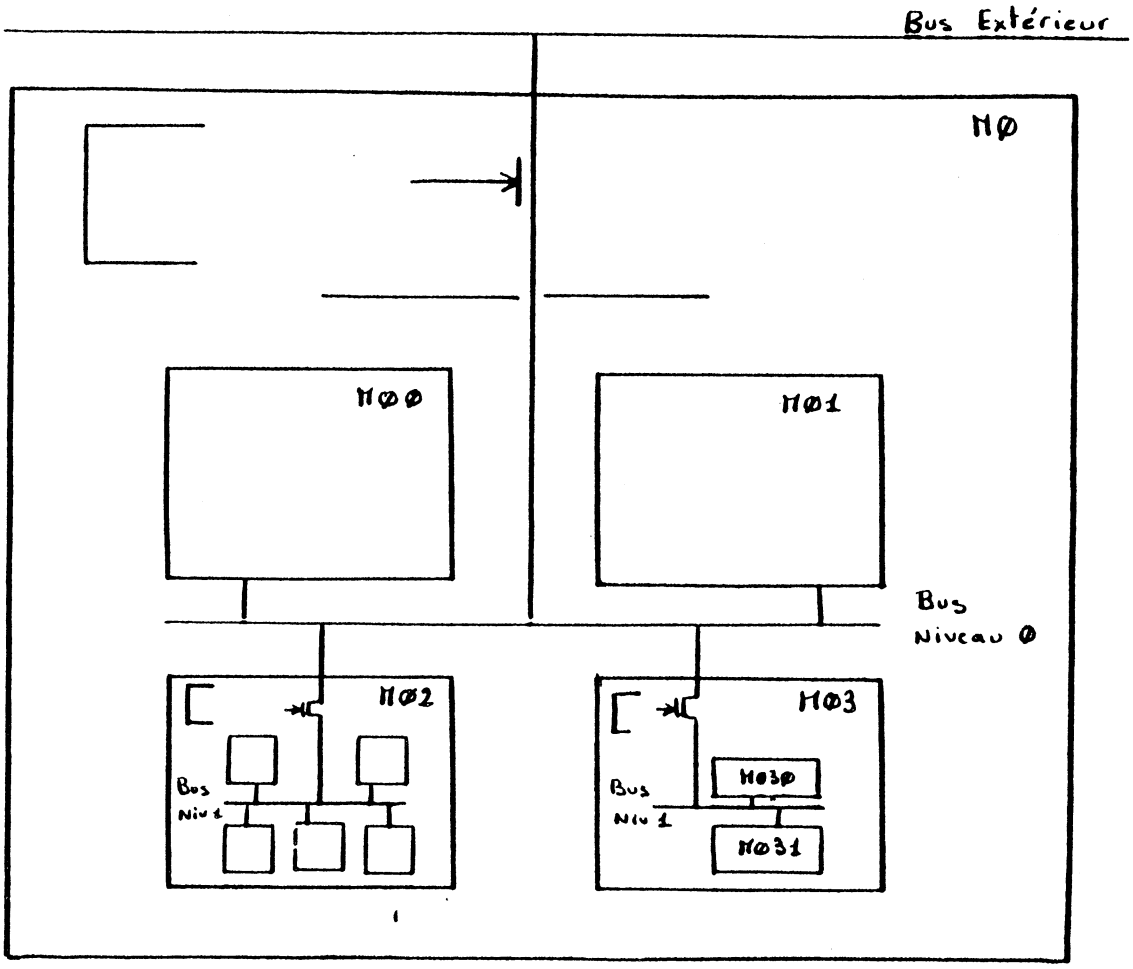


Figure VII.3

Appelons M0 ce module "racine" possédant le bus de niveau 0. Deux cas se présentent pour un module Mij, où i est le niveau du module dans l'arbre, et j une numérotation du module, au sein du module l'englobant:

- Soit le module Mij est élémentaire. Dans ce cas il n'a pas de descendance, et il est connecté au bus de niveau i-1.

- Soit le module Mij est complexe. Il est également connecté au bus de niveau i-1, mais possède un bus de niveau i qu'il peut, selon les besoins, relier au bus de niveau i-1.

Pour qu'une telle organisation effectue un traitement parallèle, il faut admettre que les modules complexes ne servent, pour certaines requêtes, que de relais vers des modules inférieurs (rôle de liaison des méthodes complexes). Une fois le message transmis, ils peuvent à nouveau satisfaire une nouvelle requête, et éventuellement la traiter ou de nouveau l'aiguiller vers un niveau inférieur. Ceci entraîne que plusieurs méthodes, d'un même module, peuvent être actives simultanément, voire réentrante à l'instar des procédures logicielles. Ceci dit leur activité peut se cantonner à une attente, notamment des résultats fournis par les modules inférieurs, dans le cas où elles ne servent que "d'aiguillages".

A l'inverse, un module élémentaire ne peut avoir qu'une seule de ses méthodes actives à un instant donné. En effet, ne pouvant se décharger du traitement sur un sous-module, il est probable qu'il utilisera pleinement ses ressources lors du déroulement de chaque méthode, empêchant de ce fait le déroulement d'une autre.

Globalement, si une telle architecture se rapproche de celle développée dans le système Cm* [SWA1 77] [SWA2 77], son utilisation s'en singularise à tous points de vue. En effet dans Cm* le but est d'interconnecter différents processeurs pouvant partager les mémoires associées à chacun. Un aperçu de diverses "machines arbres" est reporté dans [TRE 82]. Dans notre optique, l'interconnection ne sert qu'à communiquer les informations strictement nécessaires. Rappelons les objectifs fixés en début de chapitre:

- D'une part il n'est pas question de partage de mémoire en ce sens que la philosophie de la machine réside en une coopération de ses différents "opérateurs" ne pouvant communiquer que par envoi et réception de messages.

- D'autre part on travaille à un niveau plus fin que celui d'un processeur, et c'est à l'intérieur de celui-ci que l'on veut réaliser une hiérarchie de différents "opérateurs".

VII.5. Conclusions

De toute évidence, les propositions de ce chapitre ne sont que les balbutiements d'une méthodologie plus complexe. Plusieurs points sont à approfondir, voire à aborder, peut-être en suivant de plus près les concepts originaux de la programmation orientée objet.

A ce sujet, revenons un instant sur la notion de classe que nous avons confondu avec celle d'objet en un unique concept de module. Il semble possible de la retrouver en partie à l'aide d'un module complexe, contrôlant différents sous-modules qu'il gèrera tout comme un système d'exploitation gère ses ressources matérielles. Ainsi il pourra représenter une infinité de sous-modules en acceptant et mémorisant tous les messages qui leur sont destinés, avant de les redistribuer en fonction de leur charge.
Par exemple un module "Spooler" pourrait être composé d'un certains nombres de modules élémentaires "Imprimante", et accepterait tous les messages d'impression, en ayant pour fonction de les rediriger sur les imprimantes, en fonction de l'occupation de celles-ci.

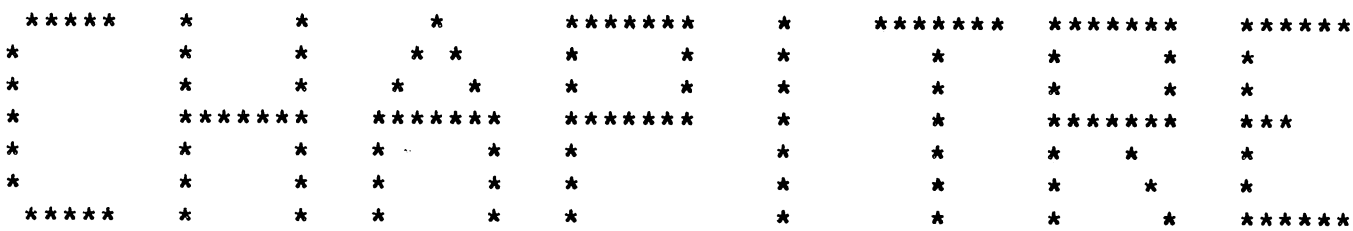
Un autre écart, vis à vis des concepts originaux, se situe au niveau du principe d'héritage des propriétés. En effet dans le modèle proposé, si tout est en place pour l'implanter, de part la connection des sous-modules au module de plus haut niveau, aucun mécanisme intrinséque ne permet de l'exploiter. En effet, dans le cas où aucune méthode "locale" ne peut être utilisée pour satisfaire un requête, la recherche d'une méthode répondant au message reste à la charge de la méthode englobante. On pourrait ainsi introduire un mécanisme redistribuant tout message, non localement satisfait, à partir de la racine.

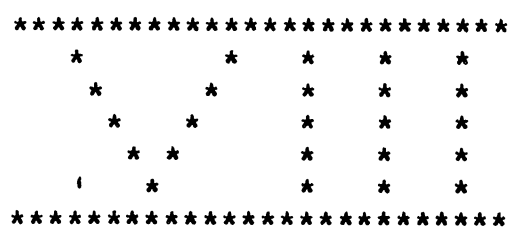
Pour finir, cette approche se heurte à différentes difficultés techniques ou théoriques:

- La première résulte du manque d'outils permettant de spécifier le parallélisme, ce qui empêche la simulation et donc la validation de telles définitions de machines.

- Une autre difficulté est liée au problème évoqué en début de chapitre, sur la classification des différents algorithmes parallèles, et par la même de l'extraction de primitives pouvant s'exécuter en parallèle.

- Enfin la dernière a trait au domaine du VLSI, pour lequel la hiérarchisation de différents modules entraîne des difficultés non encore maitrisées, ayant pour principaux effets d'augmenter de façon inconsidérée les parties opératives et parties contrôles, ainsi que les liaisons permettant de les assembler.





DESCRIPTION MATERIELLE

Chapitre VIII

DESCRIPTION MATERIELLE

Si le processus de filtrage présenté au cours des chapitres précédents apporte, dans sa version logicielle, un gain de performances substantiel, seule une implantation matérielle permettra d'effectuer ce traitement "au vol". Plus précisément, un opérateur matériel devrait être capable d'exécuter l'algorithme de préunification lors du transfert des données, issues du disque, sans le ralentir, tout en réduisant le flux d'informations grâce au filtrage effectué. Pour réaliser cet opérateur, nous nous appuierons sur l'ébauche de méthodologie exposée au chapitre précédent, au vu des raisons suivantes:

- La rapidité du "circuit" est une contrainte impérative, quant aux gains espérés. Dès lors, il paraît nécessaire de mettre en oeuvre au niveau matériel, tout le parallélisme dégagé dans l'algorithme.

- La décomposition "naturelle" de l'algorithme en plusieurs primitives relativement indépendantes, incite à reproduire matériellement cette organisation.

En effet, ces particularités et nécessités entraîneraient, dans l'optique d'une réalisation plus classique, une complexité accrue due au partage de ressources des primitives devant s'exécuter en parallèle. Aussi nous nous orientons vers notre approche modulaire, jugeant que le surplus matériel, lié à cette méthode de par la redondance de certains éléments, devrait s'avérer négligeable devant la différence des coûts de validation et de mise en oeuvre existant entre circuits complexes et circuits simples.

VIII.1. Module disque

Pour préciser l'implantation matérielle du filtre, il convient de présenter tout d'abord l'environnement au sein duquel il s'intègre. Selon l'approche modulaire, cet environnement constitue en fait le module englobant, en l'occurence le module disque. Au sein de la machine OPALE, ce module se situe lui-même entre les éléments de traitement et une unité de disque (cf Fig I.1. où le module disque se substitue au processeur disque). Notons que le concept de module s'applique parfaitement en ce sens qu'un module disque peut-être vu par le reste de la machine comme un élément de traitement totalement autonome, capable d'assumer seul un certain ensemble de fonctions que nous allons maintenant préciser.

VIII.1.1. Fonctions

Les fonctions assurées par le module disque se scindent en deux catégories:

- Les fonctions "classiques" d'un canal d'entrées/sorties, permettant d'accéder au disque, que ce soit en lecture ou en écriture.

- Les fonctions liées à la mise en oeuvre du filtrage, pouvant éventuellement faire appel aux fonctions précédentes.

Nous délaisserons les premières, ayant opter pour les solutions et outils "classiques" permettant de les implanter; par contre, nous présentons ci-dessous la liste exhaustive des secondes:

- Compilation d'un ensemble de buts.

- Préunification d'un ensemble de buts.

- Association et vérification de la cohérence des substitutions générées au cours de la préunification.

La définition précise des méthodes (au sens de l'approche modulaire exposée au chapitre précédent), intégrant ces fonctions, ne pourra se faire qu'une fois spécifiés différents points, concernant la machine dans son ensemble. Par exemple, le choix entre le déclenchement de la préunification par le module disque lui-même ou

par un autre élément de la machine reste à faire. Ceci étant, ces points ne sont pas à proprement parler des difficultés, puisque la plupart réside en des choix. A l'inverse des méthodes, le modèle de données complexe du module peut d'ores et déjà être précisé, comme nous allons le voir.

VIII.1.2. Architecture

La structure complexe du module disque est constituée de trois sous-modules, de deux buffers d'entrée et deux buffers de sortie (les termes entrée et sortie étant relatifs au module filtre), interconnectés par un bus (Fig VIII.1.):

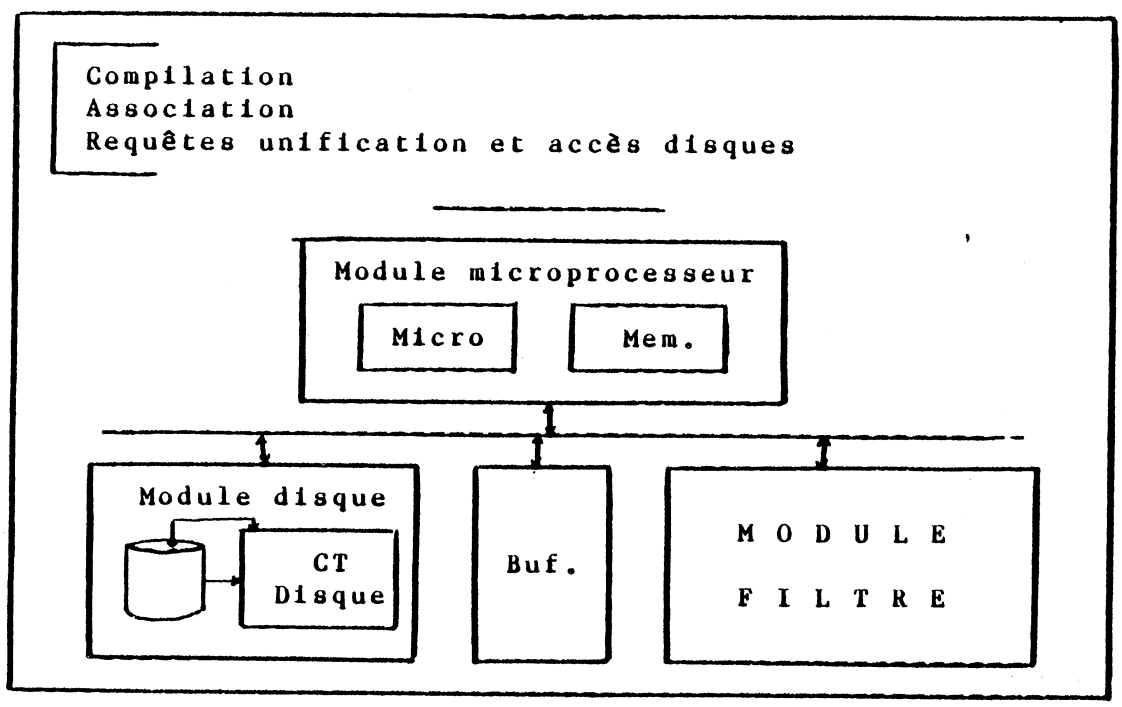


Figure VIII.1.

- **Le module micro-processeur**

  Il comporte un micro-processeur doté d'une mémoire locale et assume trois sortes de fonctions:

  . les fonctions d'interface avec le reste de la machine (module englobant du module disque).

  . la retransmission des requêtes d'entrées/sorties "simples" (sans opération de filtrage) vers le sous-module contrôleur disque.

. Une partie des fonctions de filtrage, plus précisément la compilation des buts, l'initialisation de la préunification et du balayage de l'espace disque, et pour finir, l'association des substitutions générées par la préunification.

- **Le module contrôleur disque**
  Il est constitué essentiellement du contrôleur disque et de l'unité disque elle-même. Ses fonctions permettent d'accéder physiquement au disque à partir des données classiques: face, piste, secteur (éventuellement cylindre). Pour la suite nous considèrerons qu'il fournit, en réponse à une requête, une succession de buffers de données, chacun de la taille d'un transfert élémentaire (a priori un secteur).

- **Le module filtre**
  Nous détaillerons dans les paragraphes suivants ce module, coeur de la réalisation matérielle, car effectuant la préunification.

Du point de vue des communications, le module micro-processeur sera le gestionnaire du bus local, les modules contrôleur disque et filtre lui signalant leurs requêtes en utilisant ses mécanismes d'interruptions. Cette gestion assurée, les communications entre modules s'effectuent par émission et réception de messages, suivant la "philosophie" exposée au chapitre précédent.

VIII.2. Module filtre

Le module filtre est aussi un module complexe, regroupant les sous-modules élémentaires assurant les différentes primitives dégagées dans l'algorithme de préunification (cf chapitre IV). Vis à vis de l'extérieur, il effectue la préunification mais a en fait essentiellement un rôle de "coordinateur", se déchargeant des tâches de "base" sur ses sous-modules. Plus précisément il assure:

- L'initialisation de l'algorithme de préunification.
- La transmission des résultats de la préunification, au module micro-processeur.
- La réception des buffers de données issus du module contrôleur disque. Pour assumer ce rôle d'interface, il est muni de deux buffers d'entrée et de deux buffers de sortie.

Nous allons décrire, à travers les messages qu'il reçoit, les méthodes qu'il supporte avant d'en préciser l'architecture.

VIII.2.1. Méthodes

Trois méthodes sont attachées au module filtre. La première est activée sur un message (issu du module micro-processeur) lui signalant une requête de filtrage, accompagnée des codes générés par la compilation de l'ensemble de buts courant. L'algorithme de la méthode se résume en la connection du bus local du module au bus externe, afin que chaque sous-module puisse récupérer la partie du code le concernant. En fin de retransmission, la méthode émet un signal d'initialisation sur le bus local du module et se désactive.

La deuxième est activée sur la réception d'un message (issu du module contrôleur disque) indiquant la transmission d'un buffer de données. Si l'un des deux buffers d'entrée du module est vide, alors les données sont acceptées et stockées dans le buffer vide. Sinon, un message d'attente est émis, qui sera reçu par le module contrôleur disque. Un buffer d'entrée est soit en lecture (plein) et dans ce cas accessible par le sous-module d'entrée de la structure complexe, qui le "consomme", soit en écriture (vide) et dans ce cas peut être accedé par le filtre pour le remplir.

Enfin la dernière méthode est initialisée sur requête du module

de sortie, requête signalant qu'un des deux buffers de sortie est plein. Elle a pour rôle d'en référer au module micro-processeur et de lui transmettre les résultats de la préunification.
Pour ne pas risquer de ralentir le disque, on envisage l'exécution systématique de cette méthode en fin de chaque transfert d'un buffer d'entrée, ce qui d'une part assure un délai suffisamment important pour transmettre le buffer de sortie avant le prochain message du contrôleur disque, et d'autre part évite l'engorgement des buffers de sortie.

VIII.2.2. Architecture

D'après ce que l'on vient de voir, le module filtre assume trois méthodes. Son modèle de données complexe va lui permettre d'effectuer la préunification, et se compose de cinq sous-modules tous élémentaires, connectés par un bus comme représenté sur la figure VIII.2.

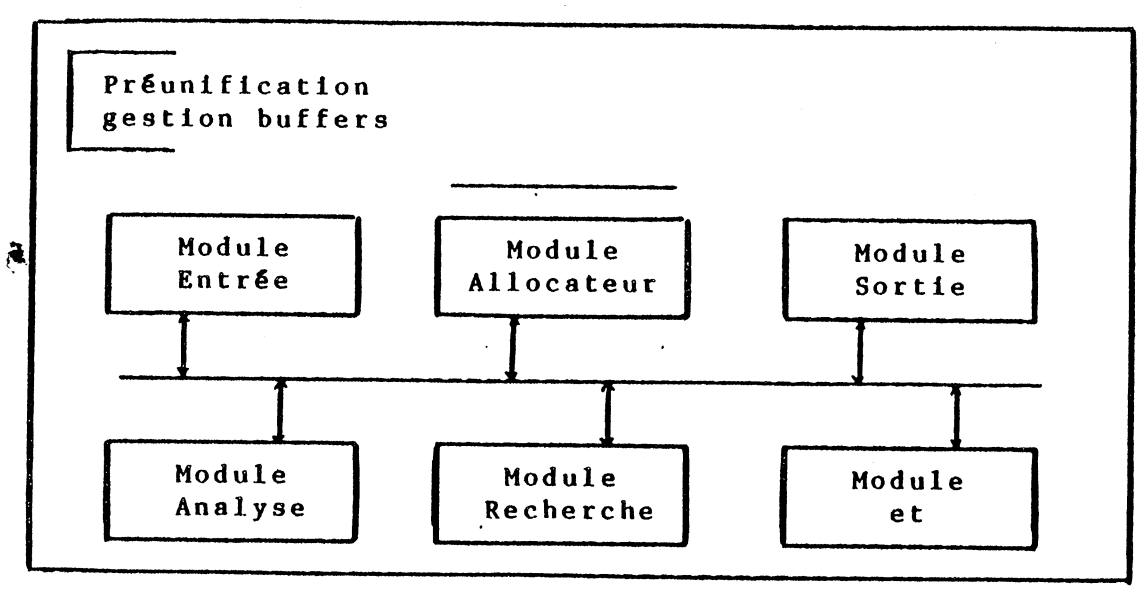


Figure VIII.2.:

Comme énoncé précédemment le bus local peut-être connecté au bus externe, le contrôle de la connection étant assuré par la méthode "filtre", permettant le chargement du code dans les sous-modules.
Notons que pour des raisons de performances, la transmission des résultats de la préunification ne s'effectue pas par

l'intermédiaire du bus local. En effet, les résultats sont directement envoyés au micro-processeur, sur sollicitation du module de sortie. De même les données en entrée sont directement stockées dans les buffers d'entrée, sans passer par le bus local.

Deux des sous-modules gèrent les entrées/sorties, explicitées ci-dessus, entre le modèle complexe et l'extérieur. Pour ce faire, ils possèdent des liaisons directes avec les méthodes du filtre. Ces grandes lignes exposées, nous allons maintenant détailler chaque sous-module, en commençant par celui s'occupant de la gestion des messages circulant sur le bus local du module filtre.

Notons encore qu'initialement nous ne prévoyons que deux buffers d'entrée, car nous estimons que le filtre devrait être suffisament rapide pour que cette seule bufférisation lui permette de suivre le débit. De plus en basculant sur l'un et l'autre des buffers, il est possible de recevoir des données issues du disque sans arrêter la préunification. Ces considérations justifient également l'utilisation de deux buffers de sortie.

VIII.2.3. Gestion des messages

Vis à vis des concepts originaux de la programmation orientée objet, nous avons introduit, par l'exploitation du parallélisme, la notion de temps. Ceci soulève plusieurs difficultés quant à la gestion des messages, et nous nous intéresserons notamment aux suivantes:

- Comment empêcher plusieurs modules d'émettre simultanément, évitant ainsi le "brouillage" mutuel des messages?

- Quel doit être le comportement d'un module actif, solicité par un autre message: mémorisation, prise en compte immédiate, perte, ou autre?

Les solutions et choix que nous proposons par la suite se justifient par deux arguments:

- Le premier est lié à l'application et concerne le nombre (moyen) de messages peu élevé, émis pour l'analyse d'un symbole (nous fournirons par la suite une évaluation chiffrée). De ce fait il est inutile de mettre en oeuvre un processus sophistiqué, largement sous-utilisé.

- Le deuxième est plus général en ce sens qu'il a trait à la philosophie de la démarche. En effet il est essentiel de conserver une unité dans la réalisation de toutes les fonctions de la machine, donc en particulier pour celle des communications, ce pour bénéficier des avantages de la modélisation.

Ceci posé, nous proposons d'ajouter à la structure complexe un module allocateur, responsable de la gestion du bus, à l'aide du protocole suivant:

- Initialement le bus est la propriété de l'allocateur, aucun autre module que lui ne pouvant émettre.

- L'allocateur émet alors sur le bus un message d'invite d'émission, précisant le module invité, étant entendu qu'un seul module l'est à la fois:
  . Si le module invité n'a pas de message à émettre, il ne répond pas, et l'allocateur sollicite le module suivant, bouclant ainsi sur l'ensemble des modules.
  . Sinon, le module invité signale à l'allocateur qu'il prend le bus, le conserve le temps nécessaire à l'envoi de son message, puis le rend à l'allocateur qui sollicite le module suivant.

Ce protocole reporte la gestion des messages au niveau de chaque module. En effet c'est au module émetteur de conserver le message jusqu'à l'obtention du bus. Pour ce faire chaque module a trois états logiques:

- **Réception**
  Le module surveille le bus et détecte ainsi tous les messages, émis soit par un autre module soit par l'allocateur. Dès qu'un message est détecté, le module passe en état d'exécution.

- **Exécution**

  Cet état comporte deux phases:

  . La première consiste en la sélection ou non du message: si le message appartient aux points d'entrée du module alors le message est reconnu et le module passe dans la deuxième phase d'exécution.

  Sinon le module retourne dans l'état de réception.

  Notons qu'un message d'invite sera tel qu'il ne pourra jamais correspondre à l'un des points d'entrée de l'un des modules autre que celui déclenchant la méthode d'émission.

  . la deuxième phase, si elle existe, consiste en l'activation de la méthode sélectionnée, puis au retour en l'état de réception.

- **Emission**

  Cet état permet d'assurer la gestion des messages par le module lui-même. Si au cours du déroulement d'une méthode un message doit être émis, le module passe dans l'état d'émission en se bloquant en ce point d'émission. Dès lors il surveille le bus en attente d'un signal d'invite le concernant, émet son message, ce qui débloque la méthode qui peut alors se poursuivre: le module revient en état d'exécution.

De toute évidence un mécanisme plus performant pourrait être implanté. Une amélioration consisterait à dupliquer les indicateurs d'attente d'émission de chaque module, au sein de l'allocateur. Dès lors celui-ci pourrait, les scruter, et directement allouer le bus aux modules en attente d'émission, tout en remettant à zéro les indicateurs des modules satisfaits. Si le gain de performances est clair, il est tout aussi évident que ce mécanisme entraîne des déviations sensibles de la philosophie initiale. En effet on en arrive à différencier certains messages, leur offrant des ressources privées et par voie de conséquence, un traitement spécifique: dans ces conditions, que reste-t'il de la méthodologie initiale et des gains de simplicité que l'on en espérait?.

Par contre l'avantage de la mise en oeuvre proposée réside en sa cohérence avec la philosophie générale d'envoi et de réception de messages, permettant ainsi d'être implantée en utilisant les mécanismes communs. Nous la justifierons plus amplement au fur et à

mesure de la description des modules élémentaires.

Par cette implantation, nous choisissons que chaque module ne mémorise pas l'ensemble des messages qu'il pourrait recevoir. En effet celui-ci est "aveugle" dès lors qu'il se trouve dans les états d'exécution ou d'émission. Si un message, dont il est un récepteur potentiel, passe sur le bus, il sera, pour lui, perdu.

Ce choix nous semble raisonnable en ce sens qu'un message émis à un instant donné dépend du contexte du moment, et qu'une exécution différée, en cas de mémorisation, risque, dans la plupart des cas, de ne plus avoir le même sens. Quant à la possibilité d'interruption d'une méthode en cours, pour en activer une autre, elle semble de prime abord beaucoup plus lourde à mettre en oeuvre sans non plus assurer, pour tous les cas, la cohérence du résultat.

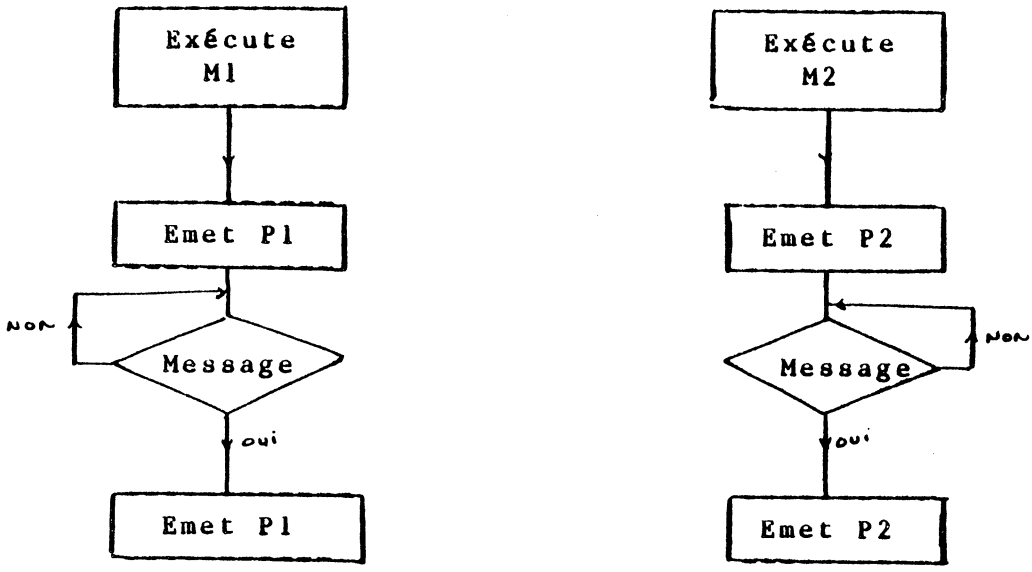
VIII.2.4. Synchronisation

Si la mise en oeuvre de la gestion de messages est simple, elle ne résout pas pour autant tous les problèmes et nous nous intéresserons à celui de la synchronisation. Plusieurs cas de figure peuvent se présenter, mais nous étudierons le seul rencontré par la suite. Il se pose en les termes suivants:

- Soient deux modules M1 et M2 déroulant simultanément et respectivement les méthodes m1 et m2.

- Soient deux points de synchronisation p1 et p2 tels que:
  . M1 envoie un message à M2 signalant qu'il est arrivé au point p1 et attend le message symétrique de M2.
  . M2 a un comportement symétrique à M1.

Il est clair qu'en utilisant sans précaution le mécanisme décrit ci-dessus, un des deux modules va se bloquer car il ne recevra pas le message de synchronisation, étant soit en exécution, soit en attente d'émission, émis par l'autre. Pour le débloquer il suffit que le module recevant le message de synchronisation, réémette le sien. Les algorithmes des modules deviennent ainsi:



Synchronisation de deux modules

VIII.3. Modules élémentaires

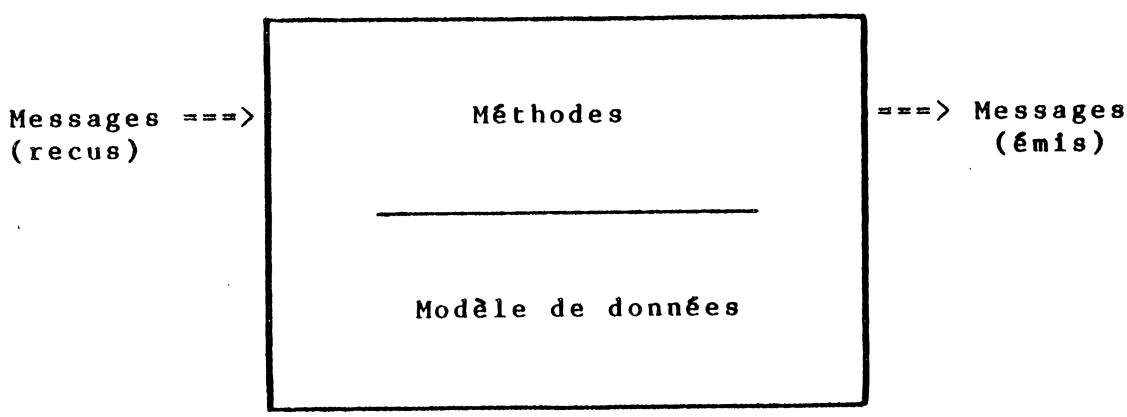
Comme énoncé précédemment, la structure complexe du filtre comporte cinq modules élémentaires. Deux d'entre eux assurent les fonctions d'interface en entrée et en sortie. Les trois autres effectuent la préunification proprement dite. La définition de ces modules a été guidée par les concepts énoncés jusqu'ici:

- Traitements locaux, sur des données locales.

- Déroulement parallèle des différents traitements.

Pour chaque module élémentaire nous allons d'une part préciser son modèle de données et d'autre part expliciter les différentes tâches qu'il assume, en fonction des messages qu'il reçoit. Eventuellement il est possible de se reporter au chapitre IV, pour se rappeler les primitives alors définies, que l'on retrouvera en tant que méthodes des différents modules.

Plus précisément on spécifiera pour chaque module, ses méthodes, son modèle de données (sans entrer dans les détails tels les registres de travail, les incrémenteurs-décrémenteurs etc ...), les messages qu'il reçoit (à gauche du schéma), les messages qu'il émet (à droite du schéma). Pour chaque message reçu, on explicitera la méthode concernée en précisant, si possible, les messages émis au cours de son déroulement.



Représentation d'un module

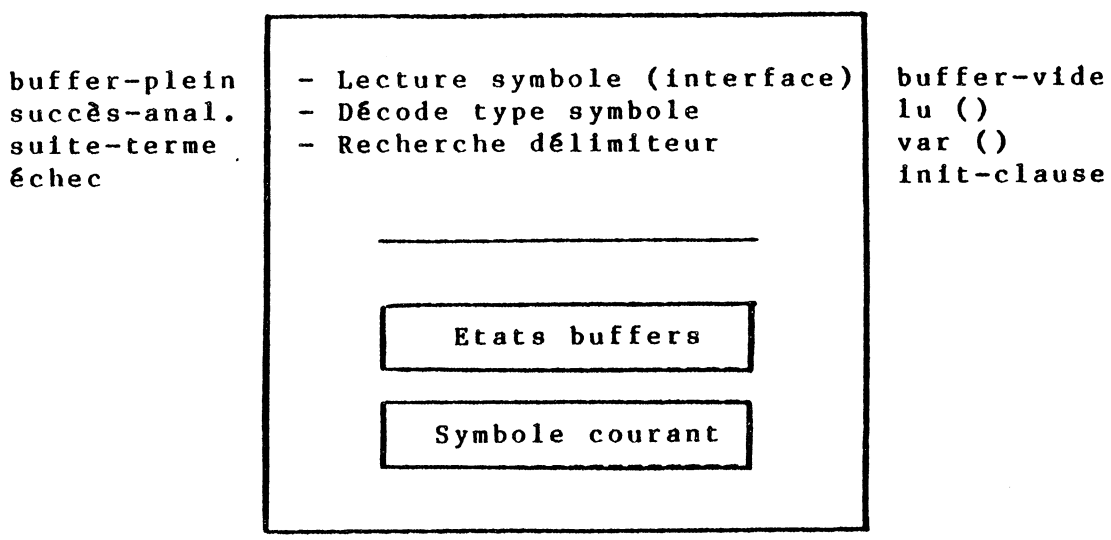
L'ordre de présentation des différentes méthodes n'est pas

significatif d'un point de vue algorithmique, ce en raison du parallélisme. Aussi nous nous appuierons sur l'algorithme séquentiel fourni au chapitre IV et exposerons les modules en fonction de l'ordre d'apparition des primitives dans cet algorithme.
Pour une vue d'ensemble des émissions et réceptions de messages, on peut se reporter au tableau du paragraphe IV de ce chapitre qui propose une synthèse des émetteurs et récepteurs.

VIII.3.1. Module d'entrée

Le module d'entrée assure deux types de tâches:

- L'interface d'entrée.
- Le contrôle des boucles de l'algorithme, tant pour le déroulement au sein d'une clause, que pour l'avancement d'une clause à l'autre.



Module d'entrée

L'interface en entrée permet d'avancer dans l'analyse de la clause courante, en fournissant le symbole suivant. Le contrôle en lecture des deux buffers d'entrée est assuré par le module, alors que l'écriture est effectuée par une méthode du filtre (cf paragraphe précédent). Le message "buffer-plein" lui signale que le buffer courant en écriture est plein ce qui relance, si nécessaire, la préunification. Inversement le module émet le message "buffer-vide" dès lors que le buffer courant en lecture est vide. Ces messages ne sont pas émis sur le bus local, mais en direction de la méthode du filtre chargée de recevoir les messages de lecture du module contrôleur disque.

L'élément atomique sur lequel se déroule la préunification est le symbole. Aussi le module répond au message "suite-terme" en fournissant aux autres modules le symbole courant, par l'émission

du message "lu(symbole)" ou "var(symbole)", en fonction du type du symbole. Il recharge alors son registre courant pendant l'analyse du symbole qu'il vient de fournir.

Le message "succés-analyse" signifie que l'en-tête de clause analysée est préunifiable avec au moins l'un des buts. Dès lors le module avance dans le buffer d'entrée jusqu'au début de la clause suivante, en transmettant directement la queue de clause dans le buffer de sortie, en recherchant le délimiteur de fin de clause. Il émet alors le message "init-clause" indiquant à tous les modules le début de l'analyse d'une nouvelle clause.

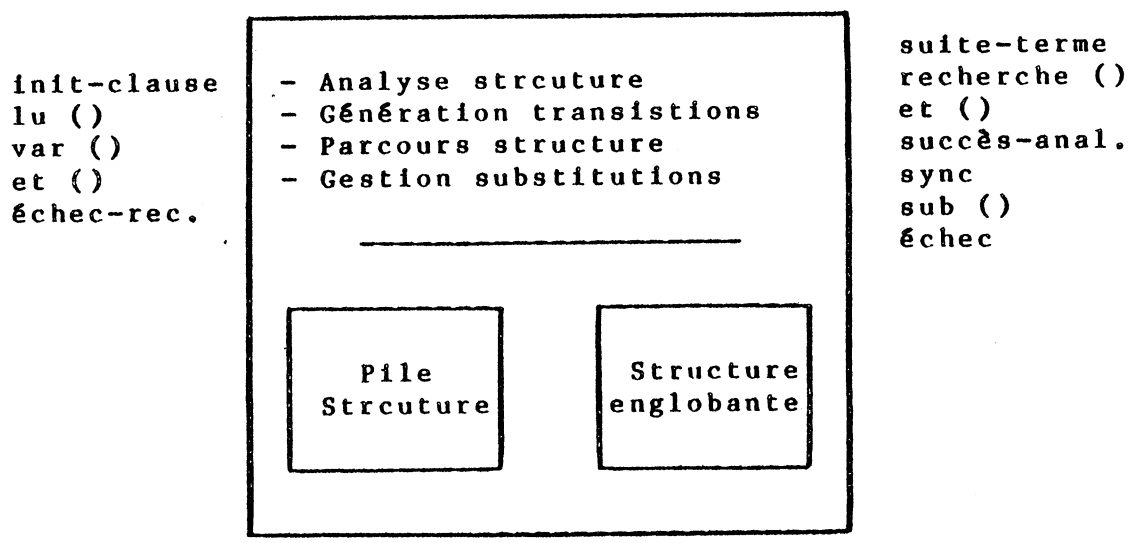
Le message "échec" lui signale l'échec de la préunification, ce qui l'amène également à rechercher la fin de la clause en cours d'analyse, puis à émettre un message "init-clause".

Notons que la recherche du délimiteur de fin de clause est une opération interne au module, ce qui assure sa rapidité d'exécution, point important vis à vis des performances.

VIII.3.2. Module analyse structure

Ce module est de loin le plus complexe de tous (bien qu'élémentaire suivant les définitions fournies jusqu'ici). Il assure différentes tâches, certaines imbriquées les unes aux autres:

- L'analyse de la structure du terme courant et la génération des transitions.
- Le déplacement dans la structure englobante.
- La gestion des substitutions.



Module analyse

Pour ce faire il possède une pile pour stocker la structure du terme courant. Plus précisément, cette pile mémorise l'arité de chaque symbole non "zéro-aire", ainsi que l'adresse des noeuds correspondants de la structure englobante dans le cas où ce noeud est générateur d'un début de substitution.
De plus il intègre un automate d'états finis permettant de se déplacer dans la structure englobante en fonction de la transition générée.

Sur la réception du message "lu(symbole)" il émet de suite le message "recherche(début-liste,longueur)", qui déclenchera la recherche de ce symbole dans la liste de valeurs associée au noeud

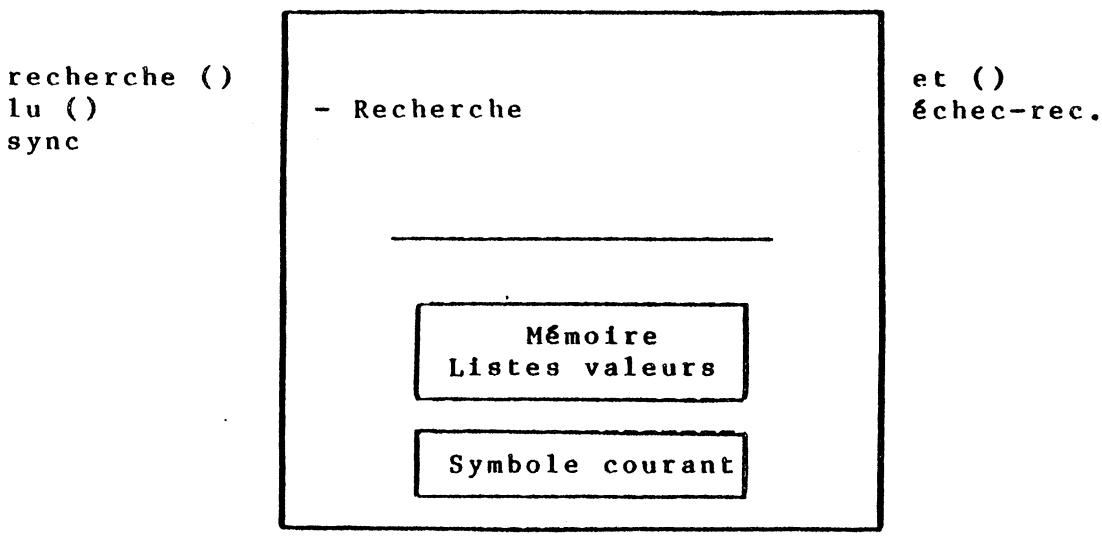
courant, et génère éventuellement un message de début de substitution (le début et la fin de la liste de valeurs sont issus du noeud courant). A la suite de quoi le module va générer la ou les transitions nécessaires et assurer le ou les déplacements dans la structure englobante. Au fur et à mesure il gère les débuts et fins de substitutions en émettant le message "sub(code,position)" où "code" spécifie un début, une suite ou une fin de substitution, et "position" fournit la position du noeud générateur, dans la structure englobante. Il émet alors un signal de synchronisation "sync", à l'attention du module de recherche, évitant le blocage dû à l'attente du signal de fin de recherche, peut-être déjà émis.

La réception du message "et(adr-chaine)" l'autorise à émettre, soit un message "succès-analyse" dans le cas où sa pile de structure est vide, soit un message "suite-terme" en cas contraire.

A l'inverse, la réception de "échec-recherche" lui demande soit de rechercher la fin d'une substitution, si l'une est en cours, soit de signaler un échec définitif de la préunification de la clause courante par l'envoi de "échec".
En cas de recherche d'une fin de substitution, il émettra le message "et(adr-chaine)", où 'adr-chaine' indique l'adresse de la chaîne de bits attachée à la variable substituée. Cette adresse est retrouvée grâce à la mémorisation, dans la pile de structure, de la position du noeud générateur de la substitution.

VIII.3.3. Module de recherche

Le module de recherche assure une seule fonction:
- La recherche du symbole courant dans la liste courante.



Module recherche

Il possède une mémoire locale où sont stockées les listes de valeurs associées à chaque noeud de la structure englobante.

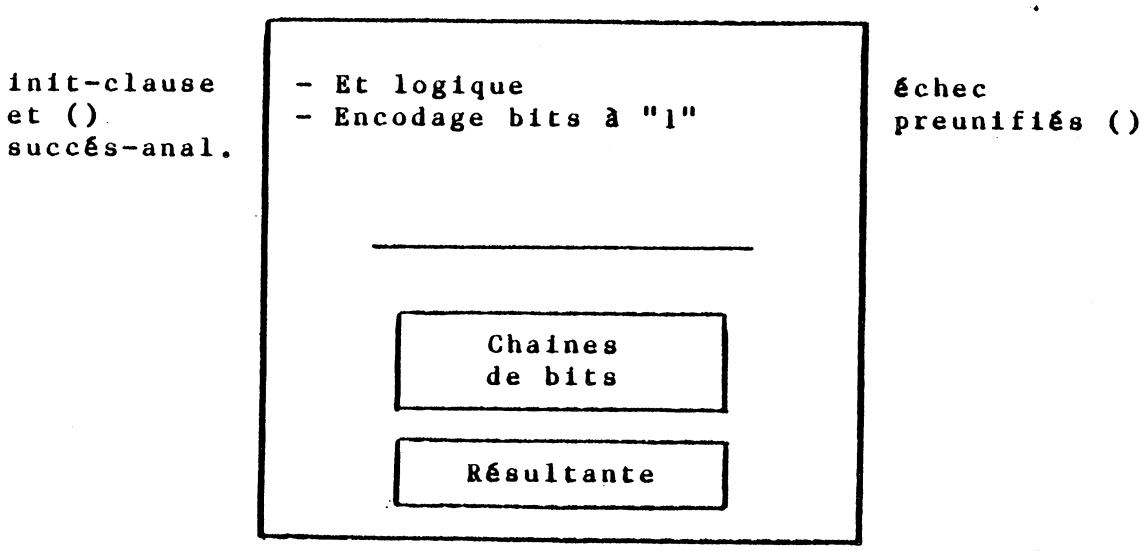
A la réception du message "lu(symbole)", le paramètre 'symbole' est stocké dans le registre de comparaison du module. Le message "recherche(début-liste,longeur)" déclenchera alors la recherche à partir de l'adresse fournie en premier paramètre du message, sur une longueur précisée par le deuxième paramètre.

En cas de succès de la recherche, le module émet "et(adr-chaîne)" où le paramètre fournit l'adresse de la chaîne de bits attachée à la valeur reconnue. En cas d'échec il émet "échec-recherche".

VIII.3.4. Module et

Ce module assure deux tâches:

- La gestion des buts encore reconnus.
- L'encodage des bits à "1" en fin de préunification réussie.



Module et

Pour ce faire il possède une mémoire locale dans laquelle sont mémorisées les chaînes de bits attachées aux différentes valeurs, et une chaîne de bits 'résultante' où se fait le cumul des différentes intersections. Pour l'encodage des bits de la résultante il utilise un encodeur câblé.

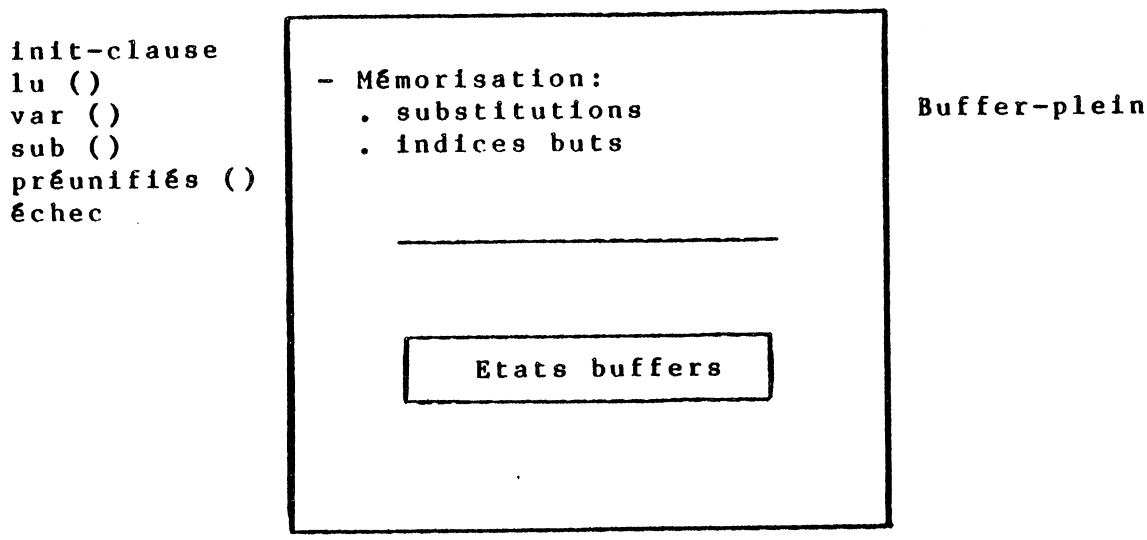
Le message "init-clause" lui indique le début d'une nouvelle analyse et déclenche la mise à "1" de tous les bits de la résultante, traduisant le fait que tous les buts sont initialement candidats à la préunification.

Sur la réception du message "et(adr-chaine)", il effectue le "et" logique entre la chaîne adressée et la résultante. Si la liste devient vide il émet "échec", la préunification échouant. Il accepte également le message "succès-analyse", et en réponse encode les bits à "1" de la résultante, fournissant ainsi les indices des buts préunifiés par le message "préunif(indices)".

VIII.3.5. Module sortie

Ce module assure la mémorisationt:

- Les substitutions générées lors des préunifications.
- Les indices des buts atteints, en cas de succès.



Module sortie

Il utilise deux buffers de sortie, en signalant au filtre que l'un des deux est plein, par le message "buffer-plein". Ainsi le filtre pourra transférer ces résultats vers le micro-processeur, pour terminer l'unification, sans arrêter le filtrage.

Le message "sub(code,position)" lui signale les débuts et fins de substitutions, conditionnant la mémorisation des symboles (ou variables) issus des messages "lu(symbole)" ou "var(id-var)". Le module incrémente et décrémente un indicateur de substitutions respectivement à chaque apparition d'un code de début et de fin, ce qui lui permet de connaître pour tout symbole lu s'il fait ou non partie d'une substitution.

Le message "préunifiés(indices)" lui demande de stocker les indices des buts préunifiés, à la suite des substitutions générées. Inversement, le message échec provoque l'annulation des substitutions générées par l'analyse de l'en-tête courant.

VIII.4. Fréquence des messages

Nous avons précédemment affirmé que les durées d'émission des messages seraient négligeables devant les temps de traitements liés à ces messages. Avant de justifier cette affirmation, nous proposons une synthèse à l'aide du tableau ci-dessous, des différents émetteurs et récepteurs. Chaque colonne du tableau correspond à un module, et chaque ligne spécifie un message par:

- Son nom et son émetteur, précisé par la colonne dans laquelle il se trouve.
- Ses récepteurs, signalés par des 'R' dans les colonnes correspondantes aux modules concernés.

Emetteurs et récepteurs

Entrée	Analyse	Recherche	et	Sortie
lu	R	R		
var	R			R
init-cl.	R		R	R
R	Suite-ter.	-	-	R
-	Recherche	R	-	-
-	et	-	R	
R	Succès		R	
	Sync	R		
	Sub			R
R	Echec			R
	R	et		
	R	Echec-rec		
R			Echec	R
			Préunifie	R

Bien entendu, le nombre de messages émis lors de l'analyse d'une clause varie de l'une à l'autre, en fonction des substitutions générées, du succès ou de l'échec de la préunification, des variables disques ... Nous proposons ici l'évaluation du nombre de messages émis dans le pire cas, pour l'analyse d'un symbole. Ce cas se présente quand une substitution se commence et se termine sur le même symbole, qui de surcroît n'appartient pas à la liste de valeurs courante. Les messages émis sont alors les suivants (l'ordre n'étant pas unique):

Messages	Emetteurs
lu (symbole)	Module entrée
recherche ()	Module analyse
sub ()	Module analyse
sync	Module analyse
echec-recherche	Module recherche
et ()	Module analyse
suite-terme	Module analyse

Pour apprécier l'aspect temporel, il faut se rappeler que le goulot d'étranglement de l'algorithme se situe au niveau de la recherche. Dès lors il s'agit d'allouer le plus de temps possible à ce processus. De ce point de vue, les messages sync et sub sont émis pendant la recherche et ne la pénalisent donc pas.

Il ne reste plus que cinq messages émis par symbole lu (on néglige ceux liés à l'ensemble de la clause), ce uniquement en cas de succès de la préunification. En effet dès qu'un échec est détecté, les symboles terminant la clause ne nécessitent plus aucune émission, la recherche du délimiteur étant interne au module d'entrée.

Ceci posé, on peut affirmer que le nombre moyen de messages émis par symbole sera inférieur à deux pour la majeure partie des clauses (dans le cas où les clauses ne sont pas unaires, ce nombre tombe en dessous de l'unité). Devant le parallélisme mis en oeuvre, le coût de ces communications devient négligeable, autorisant une solution aussi simple que celle proposée.

VIII.5. Conclusions

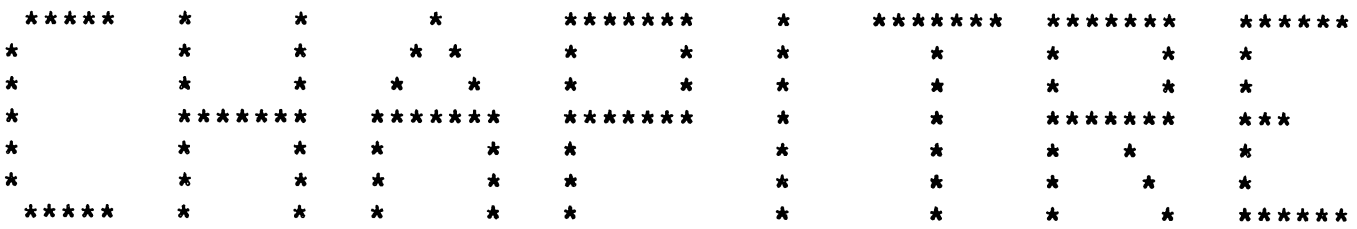
Au cours de ce chapitre nous avons défini l'architecture du filtre en fonction des concepts énoncés au chapitre précédent. Ainsi nous avons abouti aux spécifications fonctionnelles de cinq modules élémentaires, c'est à dire pouvant être vus comme des machines classiques avec une partie contrôle et une partie opérative. Nous pouvons déjà extraire de cette décomposition modulaire, deux avantages:

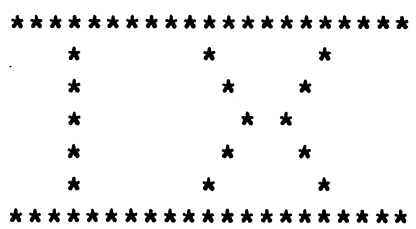
- La complexité de chaque module élémentaire est réduite, assurant une conception peu coûteuse.

- L'exploitation du parallélisme est maximale, critère vital pour une application devant respecter des contraintes de rapidité.

Ces points établis, il reste à préciser l'architecture de chaque module, en regard de leurs spécifications. Pour une maquette nous envisageons de réaliser cinq "chips" séparés, d'une part pour des raisons de complexité, et d'autre part au vu de la dimension des mémoires locales de chaque module.

Mais avant d'aborder cette ultime phase, il faut valider le modèle de gestion de messages, ainsi que les conséquences temporelles, quant au déroulement de l'algorithme de préunification, entraînées par l'exploitation du parallélisme.





SIMULATION ET REALISATION

CHAPITRE IX

RESUME

Ce dernier chapitre aborde l'aspect matériel. Nous exposons tout d'abord une simulation fonctionnelle du filtre "modulaire" décrit au chapitre précédent, où l'on fait intervenir la notion de temps. Nous proposons alors d'effectuer la conception et la réalisation matérielle à l'aide d'un "compilateur de silicium". Pour ce faire nous commençons par décrire l'ensemble des outils développés dans le cadre de la méthodologie CAPRI. Nous fournissons, pour terminer, les résultats obtenus par la "compilation" de la description d'un module.

Chapitre IX

SIMULATION ET REALISATION

Malgré son titre, ce chapitre n'a pas la prétention de fournir les masques d'un circuit de préunification. En effet à ce stade de l'étude, nous arrivons à la frontière des deux disciplines encore séparées que sont l'informatique et la micro-électronique, la conception d'un circuit ne pouvant pour l'heure se faire sans une connaissance suffisante de la deuxième. Pourtant cette frontière devient de plus en plus floue, tout au moins vis à vis de l'aspect conception, grâce à l'apparition d'outils de plus en plus sophistiqués, aidant à la réalisation de certaines étapes (placement, routage), voire en assumant la charge complète ("macrocell systems", Génération automatique de PLA, ...). L'état de l'art en la matière réside en l'élaboration de compilateurs de silicium qui, à partir d'une description algorithmique de haut niveau du circuit, devraient permettre d'aboutir automatiquement au dessin des masques. Malheureusement nombre de ces outils sont encore du domaine de la recherche, point sur lequel nous reviendrons au cours du chapitre.

Nous entendons ici simuler l'architecture modulaire du filtre, un module étant vu comme une boîte effectuant un certain nombre de fonctions. Plus précisément nous voulons valider le système de communications élaboré au chapitre précédent, ainsi que le parallélisme introduit par l'activation simultanée de plusieurs modules. Pour ce qui est de la simulation architecturale d'un module, elle demande tout d'abord sa description physique, ce qui nous ramène au premier point.

IX.1. Simulation du filtre "modulaire"

Si la version logicielle de l'algorithme de préunification tient déjà compte d'une décomposition matérielle en plusieurs primitives, nous n'avons alors, ni introduit la notion de message, ni exploité le parallélisme lié à l'activation de plusieurs primitives. Nous proposons d'intégrer ces deux points dans une simulation plus proche du matériel, ce qui devrait permettre:

- De valider sous un aspect temporel la coopération des différents modules, et par la même les solutions proposées quant aux communications.

- De décrire chaque module en se rapprochant le plus possible d'une description architecturale, qui pourra servir de base à l'étape de conception.

Nous commencerons par la simulation des communications par messages avant de voir l'implantation de la gestion de ces messages au niveau de chaque module.

IX.1.1. Le système

La simulation des envois de messages va reprendre la description architecturale effectuée au paragraphe VIII.2.3. Ainsi le système logiciel est composé d'un module allocateur, corps de la simulation, un bus logiciel, et autant de modules logiciels que de modules matériels élémentaires. La boucle principale de la simulation (le module allocataire) est la suivante:

```
Tantque (messages en attente) faire
   Pour M_courant = 1 jusqua Nombre_de_modules
   Début
      bus = invite-émission (M_courant)
      si (émet (module(M_courant))) alors
      /* M_courant a reconnu une invite le concernant
         et a placé un message sur le bus */
         for i = 1 jusqua Nombre_de_modules
            activation (module(i))
         /* Activation de tous les modules */
   Fin_pour
Fin_tq
```

Pour que cet algorithme réponde aux spécifications du VIII.2.3. il faut que les fonctions "émet()" et "activation()" prennent en compte l'aspect temporel du modèle. Plus précisément il faut que:

- émet (module(i)) assure que le module émetteur est chronologiquement adéquat, c'est à dire qu'aucun autre module ne doit émettre avant lui (en vérifiant au préalable que le module courant est en attente d'émission).

- activation (module(i)) vérifie que le module activé n'est ni en exécution ni en attente d'émission, auxquels cas il ne faudrait pas l'activer (ou le réactiver).

Ces nécessités nous ont amenés à introduire explicitement la notion de temps et d'état d'un module, comme nous allons le voir ci-aprés.

IX.1.2. Temps et état

Le point clé de toute simulation de parallélisme est l'introduction de la notion de temps. Nous l'avons introduit en dotant chaque module d'un temps local propre, se référant à un temps externe, à l'inverse commun à tous les modules.

Le temps externe est accessible à tous, mais seul le corps de la simulation peut le modifier. Il représente le temps réel lié à

un déroulement parallèle. Son incrément aura lieu après chaque invite d'émission, du temps nécessaire à l'émission de l'invite elle-même et, le cas échéant, du temps d'émission du module invité.

A l'inverse chaque module possède son temps local accédé et incrémenté par lui seul. Ce temps le situe dans le "passé" ou le "futur", par rapport au temps externe, et par ce biais par rapport aux autres modules.

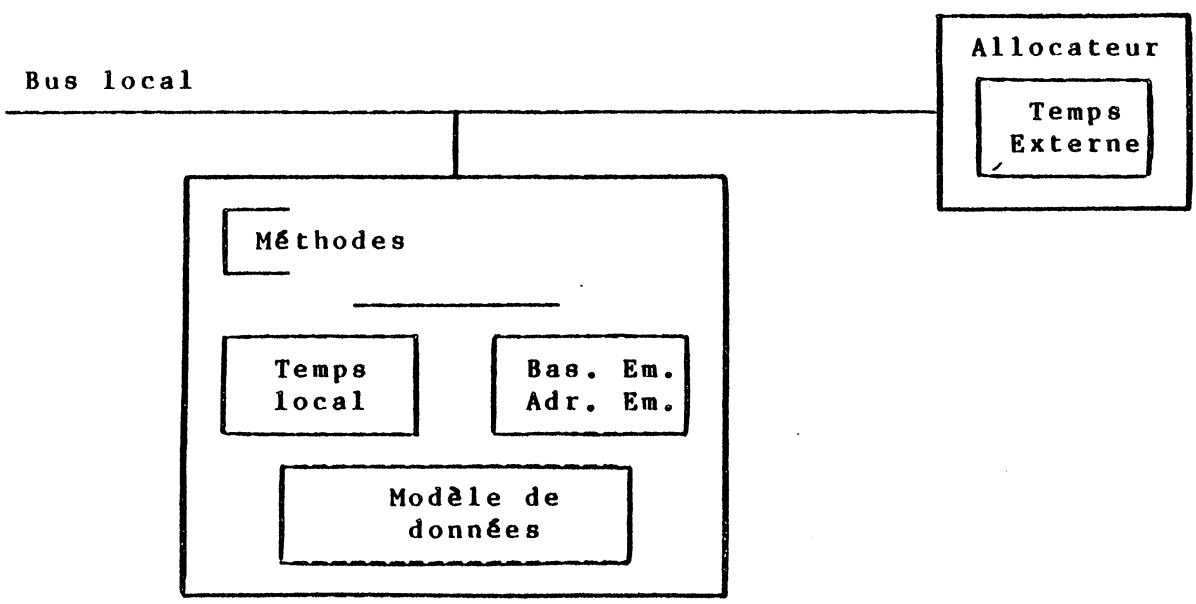


Figure IX.1

Le dernier élément utilisé pour la simulation est une bascule d'état implantée dans chaque module, indiquant si le module est ou non en attente d'émission. A cette bascule est associée un registre mémorisant "l'adresse" d'émission sur laquelle est bloquée le module (Fig IX.1.).

Ceci posé, les fonctions "émet()" et "activation()" du corps de la simulation, vont se résumer à un simple appel des modules logiciels. En effet, les spécifications du protocole de communications seront assumées au niveau de chaque module, grâce à une fonction de réception, exposée ci-dessous:

- La première action entreprise par un module activé et de comparer son temps local au temps externe. Si son temps local est supérieur au temps externe cela signifie que le module est dans le "futur", ou en d'autres termes que ce module se trouve en **état d'exécution** lors de l'émission du message. Ainsi il ne peut recevoir le message émis et se désactive sans autre action.

- Si le module n'est pas en exécution (temps local inférieur ou égal au temps externe), alors il reste deux états possibles pour ce module:

  . **attente émission**

  Si le message n'est pas une invite d'émission il se désactive sans autre action.

  Sinon il remet à zéro sa bascule d'émission, émet son message et continue le traitement stoppé par cette attente. Ce traitement terminé, il met à jour son temps local avec la valeur du temps externe incrémenté: du temps de l'invite, du temps d'émission et du temps de traitement (éventuellement nul).

  Notons que ce traitement peut se terminer sur un blocage en une autre attente d'émission.

  . **réception**

  Le module sélectionne ou non le message en fonction de ses points d'entrée, et exécute le traitement propice. Son comportement est alors identique à celui décrit ci-dessus après l'émission, si ce n'est que le temps d'émission est maintenant nul.

Le corps de la simulation se réécrit comme suit:

```
/* Initialisations des temps locaux et externe à zéro */
Tantque (messages à émettre) faire
Début
   Pour Mod_invite = 1 jusqua Nombre_de_modules
   Début
      bus = invite (Mod_invite)
      dt_émission = activation (module(i))
      Si (dt_émission != 0) alors
         Pour mod_actif = 1 jusqua Nombre_de_modules
               activation (module(mod_actif))
      t_externe = t_externe + T_INVITE + dt_émission
   Fin_pour
Fin_tq
```

La fonction de réception et le corps d'un module sont structurés comme suit:

```
si (t_local > t_externe) alors                    /* Etat exécution */
   retourne (0)                                   /* Pas de réception */
si (att_émission) alors                           /* Etat émission */
   si (bus != invite_personnelle) alors           /* Pas d'émission */
      retourne ()
   sinon début                                    /* Emission */
      dt_émet = émet (adr_émission)
      dt_exécute = exécute (suite_exécution)      /* Suite exécution */
      t_local = t_externe + T_INVITE +            /* MAJ temps local */
                dt_émet + dt_exécute
      retourne (dt_émet)
   fin_sinon                                      /* Etat réception */
si ( (adr = point_entree(bus)) != 0) alors        /* Sélection */
début                                             /* Exécution */
   dt_exécute = exécute (adr)
   t_local = t_externe + T_INVITE +               /* MAJ temps local */
             dt_exécute
   retourne (0)
fin_si
retourne (0)                                      /* Pas de sélection */
```

Ceci étant, nous allons voir pourquoi et comment nous pensons matérialiser chaque module logiciel en un "chip", les assemblant pour aboutir à une maquette du futur "circuit". Nous commencerons par la présentation de la chaîne de "compilation de silicium" que l'on utilisera par la suite.

IX.2. Outils de conception: CAPRI

A ce stade de l'étude, il est nécessaire de prendre en compte l'aspect physique des modules. En effet les algorithmes à implanter ont été spécifiés, validés puis "simulés" en fonction de l'architecture cible. Celle-ci est composée de "modules", vu par cette simulation comme des boîtes réalisant un certain nombre de fonctions. Pour avancer il faut alors "entrer" dans ces modules pour les décrire en vue de leur réalisation physique. On aborde ainsi le domaine de la conception dépassant les seules spécifications. Pourtant nous allons essayer de pousser l'étude, ce grâce à la possibilité d'utiliser différents outils de élaborés au sein de l'équipe, dans le contexte de la méthodologie CAPRI [ANC 83].

Elle a pour objectif l'élaboration d'une chaîne de traitement qui, partant d'une description de haut niveau d'un circuit, assure sa conception et son intégration automatiques. La mise en oeuvre de cette méthodologie a débuté par la définition du langage IRENE (langage de type "transfert de registre"), utilisé comme support de la description comportementale et structurale d'un circuit VLSI. Partant de ce langage de description deux chaînes d'outils sont actuellement en cours de développement:

- Celle permettant la simulation comportementale et structurale du circuit décrit.

- Celle assurant l'intégration VLSI du circuit.

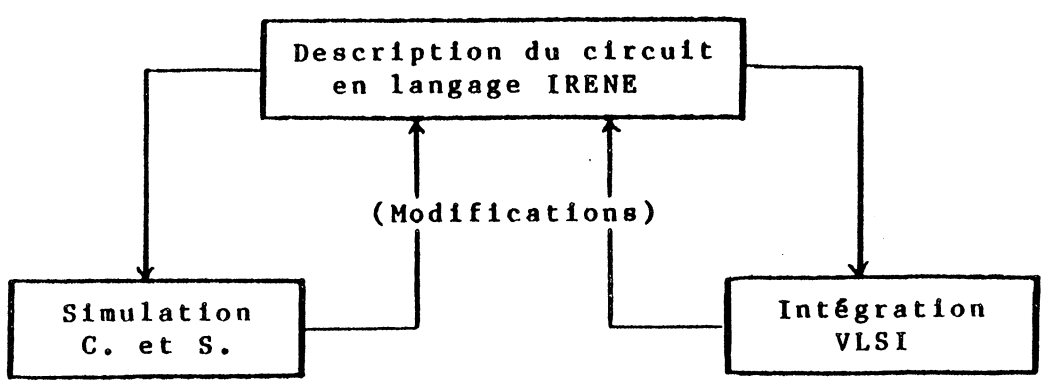


Figure IX.2

Dès lors l'élaboration d'un circuit s'effectue par raffinements successifs. La première étape consiste en la description initiale du circuit et les autres apportent les modifications induites par les résultats obtenus lors des phases de simulation et d'intégration.

Nous présenterons rapidement l'avancement actuel de tous ces outils avant de fournir la description d'un des modules définis au chapitre précédent (le module de recherche). Nous verrons alors les résultats obtenus par la "compilation" du circuit ainsi défini.

IX.2.1. Le langage: IRENE

Comme énoncé précédemment, IRENE est le langage développé dans le cadre de CAPRI, permettant de décrire le comportement et la structure d'un circuit VLSI. Nous n'en ferons ici qu'une description d'un point de vue utilisateur, ne nous intéressant aux considérations technologiques sous-jacentes que lors d'éventuelles incidences sur le premier aspect. Une discussion des objectifs du langage se trouve dans [MAR 83] et [MAR 85].

Nous exposons ci-après les points essentiels du langage, (pour une définition complète se référer à [BOU 84]), et ne verrons que la partie concernant la description comportementale d'un circuit. Pour celle-ci IRENE ressemble quasiment en tous points à un langage de haut niveau du type PASCAL. En effet un circuit est décrit sous forme d'une hiérarchie de blocs, appelés **modules** , à l'instar d'un algorithme écrit dans un langage structuré. Un module est constitué des parties suivantes:

- Un nom et des variables d'interface: celles-ci sont connues à l'intérieur du module lui-même ainsi que dans le module père. Elles permettent les communications entre modules père et fils et limitent les interconnections physiques entre les différents éléments.

- Les déclarations des variables: les variables peuvent être soit physiques soit algorithmiques. Les variables physiques sont locales au module. Si plusieurs modules doivent accéder à la même variable physique, celle-ci doit être placée dans un

module englobant les modules devant y accéder. Les variables algorithmiques n'ont pas de réalité physique, servent uniquement en début de description, et sont remplacées au fur et à mesure de l'affinement du circuit par des variables physiques. Les règles de visibilité des variables algorithmiques sont identiques à celles des langages de programmation structurée.

- Les spécifications de séquence: cette partie n'a pas de dualité dans les langages algorithmiques car elle a trait a des considérations physiques temporelles. Une spécification de séquence est constituée de listes imbriquées d'identificateurs. Les différents niveaux d'imbrication définissent autant de niveaux d'interprétation de l'algorithme, ou en d'autres termes de sous-machines:
  . La partie opérative constitue le premier niveau.
  . les niveaux supérieurs sont autant de partie contrôle, pilotant les niveaux inférieurs.
  Un état de la machine est ainsi identifié par une liste d'étiquettes de séquencement. Enfin signalons que différents types de séquencement sont possibles: algorithmiques, séquenceur, mixte.

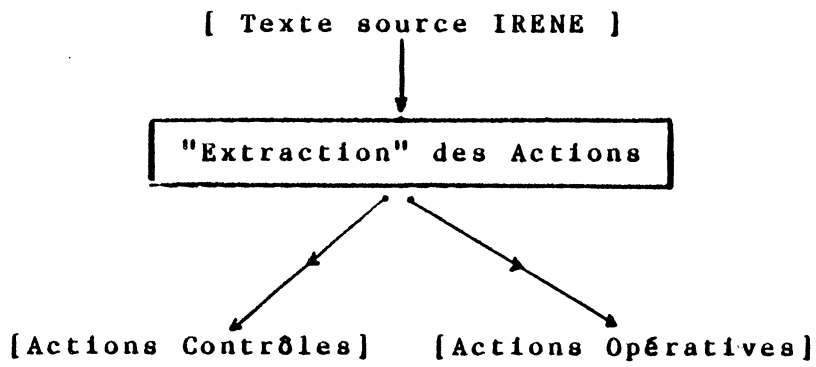
- Le corps: c'est une liste d'actions parallèles, soit opératives, soit de séquencement que l'on retrouve sous leur forme habituelle: affectations et opérations pour les premières, tests et sauts pour les secondes. Notons encore la possibilité de définir des routines que l'on peut considérer comme des "macros", utilisées par le corps à l'aide d'une instruction "call". Une action peut également consister en l'activation d'un module imbriqué.

Si actuellement le simulateur accepte l'ensemble des notions du langage, la chaîne de compilation (que l'on appellera dorénavant "compilateur") n'autorise pour l'heure que l'utilisation d'un sous-ensemble. Ainsi la notion de module imbriqué n'est pas encore implantée, ni celle de routine, pour diverses raisons que nous expliciterons au cours de la description du compilateur.

IX.2.2. Le compilateur

Même si IRENE est un langage "technologiquement dépendant", ou en d'autres termes définis en fonction des mécanismes matériels pouvant être intégrés en VLSI, la traduction d'un texte source en termes de masques d'un circuit demeure une opération complexe. Aussi la traduction a été décomposée en une succession d'étapes, définissant ainsi une "chaîne de compilation". Les traitements sont assurés par autant d'outils que nous allons rapidement présenter.

Le premier outil [MART 85] accepte pour entrée le texte source IRENE et extrait de cette description les informations nécessaires aux réalisations des parties contrôles et de la partie opérative



Rappelons, pour justifier cette étape, que l'intégration d'un module s'appuie sur un modèle architectural du type "PC-PO". En fait, comme nous l'avons indiqué dans la présentation du langage, une description définit un certain nombre de "sous-machines" imbriquées les unes dans les autres suivant les spécifications de séquencement de l'algorithme. La phase d'extraction consiste à regrouper les actions en fonction des niveaux d'interprétation ("sous-machines") auxquels elles appartiennent:

- Les actions opératives (transferts de registre, opérations de calcul unaires ou binaires) appartiennent au niveau d'interprétation le plus bas: la PO. Ces informations seront traitées par APOLLON, outils générant les masques de la PO.

- Les actions de séquencement (tests, branchements) vont appartenir aux différents niveaux d'interprétation supérieurs, ce en fonction des étiquettes auxquelles elles font référence. Un autre outil doit générer les différentes couches de PC, chacune pilotant la sous-machine lui étant immédiatement inférieure.

Parmi les notions du langage non encore prises en compte par ces différents outils, on trouve notamment les définitions de routine et de module imbriqué. Le problème lié à un appel de routine provient de la rupture de séquence qu'il induit. Pour ce qui est de la notion de module imbriqué plusieurs difficultés sont à résoudre:

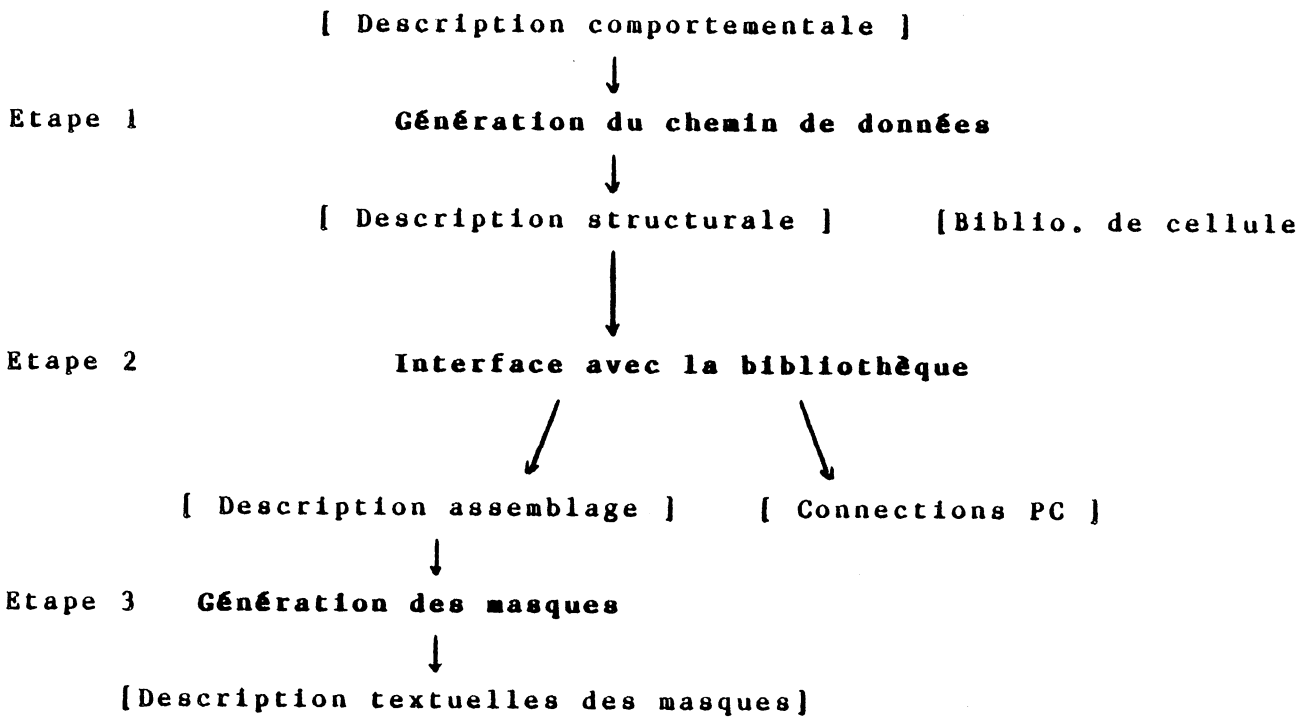
- La représentation physique des paramètres de communication (pour un seul module ils sont matérialisés par les plots d'entrée-sortie).
- Le partage des variables communes.
- L'imbrication des niveaux de séquencement.

Pour le reste, les premières utilisations d'IRENE ont soulevé divers problèmes. Ainsi il paraît nécessaire, dans un premier temps, de compléter la sémantique algorithmique de chaque instruction par une sématique "physique", précisant les conséquences matérielles de leur utilisation (ou de certaine forme de leur utilisation).

IX.2.2.1. Générateur de la PO: APOLLON

L'ensemble des actions opératives constitue l'entrée du générateur de la PO (APOLLON), et représente, pour lui, la description comportementale du circuit. APOLLON va traduire cette description comportementale en une description textuelle des masques d'une PO, capable d'exécuter l'ensemble des actions opératives spécifiées en entrée. Pour ce faire le générateur utilise un modèle de chemin de données, en l'occurrence celui mis en oeuvre pour le microprocesseur MC 68000, que l'on estime être un "bon" modèle. Il se compose de deux bus pouvant être coupés en différents points, définissant ainsi des sous parties opératives sur lesquelles peuvent se dérouler simultanément des actions différentes.

Le traitement est décomposé en trois étapes, organisées comme suit:



Les différentes étapes assument les fonctions suivantes, leur définition plus complète, ainsi que leur implantation étant exposées dans [JAM 85]:

- La première définit les différents éléments du circuit (registres, ALUs éléments (stucture du circuit). Simultanément elle génère la trace de l'exécution, sur cette PO, des différentes actions opératives.

- Ces résultats sont utilisés par la seconde pour fournir, en fonction d'une bibliothèque de cellules, les spécifications d'assemblage ainsi que des informations permettant de réaliser l'interface avec la partie contrôle.

- Enfin la dernière étape produit une description textuelle des masques de la PO, en un langage de description de masques (LUCIE [GUY 81]), en utilisant un assembleur de silicium (LUBRICK [SCH 83]).

Notons que les trois niveaux de description (comportementale,

- Enfin la dernière étape produit une description textuelle des masques de la PO, en un langage de description de masques (LUCIE [GUY 81]), en utilisant un assembleur de silicium (LUBRICK [SCH 83]).

Notons que les trois niveaux de description (comportementale, structurale, topologique) sont accessibles à l'utilisateur, permettant soit des modifications manuelles (optimisations, ajouts), soit d'entrer à n'importe quel niveau du traitement.

IX.2.2.2. Générateur de la PC

A l'instar de la génération de la PO, la génération de la PC s'appuie sur la définition d'un modèle. Celui choisi découle naturellement du langage de description en ce sens qu'il consiste en un empilement de différentes "couches" de partie contrôle, correspondant aux niveaux de séquencement spécifiés dans la description du circuit.

Un des problèmes de l'implantation réside en les connections entre les différents "étages" de partie contrôle. En effet ceux-ci peuvent entraîner un "coût en surface" rapidement prohibitif. Les réflexions actuelles [JER 85] amènent à définir un certain nombre de règles quant aux relations "inter-étages":

- Un étage de niveau quelconque ne peut recevoir des commandes que de l'étage strictement supérieur et des informations issues des plots de contrôle et par conséquent ne peut générer des commandes que pour l'étage lui étant strictement inférieur.

- Un étage de niveau quelconque ne peut tester que des "flags" (registres à un seul bit) de la partie opérative. En d'autres termes cela signifie que seuls un nombre réduit de fils "remontera" de la partie opérative vers les couches de partie contrôle.

Pour implanter un étage on utilisera, dans un premier temps, une organisation mono-PLA. En effet si différentes

possibilités sont envisageables [OBR 82] (PC câblée, PC à PLA unique, PC à plusieurs PLAs, PC microprogrammée), la mise ne oeuvre d'une PC à PLA unique demeure simple. D'autre part divers outils sont disponibles pour générer et optimiser de tels PLAs [CHU 84]. Une évolution pourra se faire vers l'utilisation des autres organisations, voire leur mélange, ce en fonction de la complexité de l'étage à réaliser. Notons qu'en fin de traitement on retrouve les outils LUCIE et LUBRICK.

IX.2.3. Le simulateur

Le simulateur n'accepte actuellement [BOU 84] que la description comportementale d'un circuit. La mise en oeuvre de la simulation repose sur la compilation du texte source en une forme intermédiaire qui sera utilisée par un interpréteur. Cette phase de compilation assure une analyse statique:

- de la syntaxe
- de la sématique: parallélisme local, mauvaise temporisation des couples condition-action, ...

Si aucune erreur n'est détectée, une forme intermédiaire est générée. L'interpétation de cette forme consiste alors à effectuer pour chaque état de la machine:

- L'évaluation de toutes les actions parallèles liées à l'état courant de la machine.
- La vérification des erreurs dynamiques: débordement, émissions simultanées sur un bus commun, ...

Un langage de commande permet une simulation interactive (pour la phase d'interprétation). Notons la possibilité de définir des "macros" contenant un ensemble de commandes.

IX.3. Maquette matérielle

Pour l'élaboration d'une maquette, nous pensons intégrer chacun des modules logiciels en autant de "chips" en regard de l'état actuel des outils dont nous disposons. En effet d'une part la notion de modules n'est pas implantée, et d'autre part, au vu de considérations technologiques (surface et densité d'intégration d'un "chip"), l'intégration en un seul "chip" des mémoires locales de chaque module, du bus de communications et finalement du protocole d'émission est pour l'instant hors de question avec la technologie à notre disposition. Quant à l'éventualité de reporter ces mémoires à l'extérieur d'un "chip" unique, elle est irréalisable de part le nombre limité de broches d'un boîtier, insuffisant pour supporter l'ensemble des fils d'adresses et de données nécessaires. Le partage de ces broches (à l'aide d'une technique de multiplexage), entre plusieurs modules, est également interdit sous peine de supprimer la plupart du parallélisme. Qui plus est, ceci entraînerait un accroissement de la complexité de l'algorithme.

Un autre argument, toujours relatif au développement des outils, réside en leur orientation actuelle vers la réalisation de circuits de "style" microprocesseur, c'est à dire ayant une partie contrôle et une partie opérative. Ceci n'est plus vrai pour notre application de par la décentralisation du contrôle sur chaque module au travers des émissions et réceptions de messages. Si la notion de module existe dans le langage IRENE, l'activation d'un module ne peut se faire que par un module englobant mais non entre deux modules de même niveau. De plus cette notion n'est pas encore implantée dans le sous-ensemble "compilable" du langage et semble soulever pour l'heure des problèmes délicats.

La dernière raison, justifiant également l'utilisation de la chaîne de compilation présentée pour aboutir à une maquette, est d'ordre économique. En effet sans outils il est impensable d'imaginer même la phase de réalisation, sans un support financier conséquent, de par le temps de conception manuelle. A l'inverse, l'utilisation de la chaîne de compilation laisse espérer atteindre de premiers résultats pour un coût dérisoire

vis à vis d'une démarche manuelle.

De plus deux raisons plus techniques permettent d'envisager une utilisation satisfaisante de ces outils dans leur état actuel:

- Les algorithmes des différents modules sont simples ce qui doit permettre une réalisation automatique satisfaisante vis à vis d'une réalisation manuelle.

- Si les performances constituent un critère décisif quant à la raison d'être du filtre, il suffit qu'elles permettent de suivre le débit du disque. En fonction des différentes évaluations effectuées jusqu'ici, il est clair que cette condition ne requiert que des performances très modestes.

Nous proposons pour la suite, non la conception de tous les modules, mais l'ébauche de l'un d'entre eux qui servira d'exemple. Pour ce faire nous avons choisi le module de recherche, estimant qu'il est représentatif des difficultés rencontrées et des solutions alors envisageables.

IX.4. Exemple: module de recherche

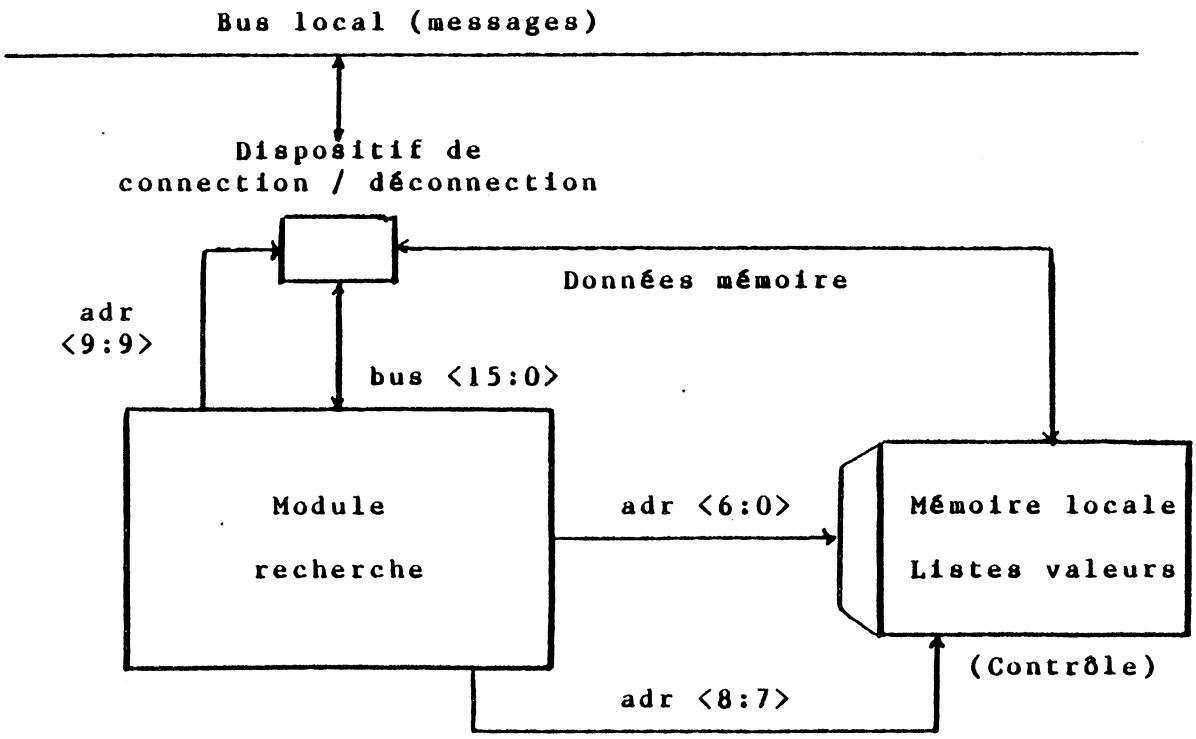
Le module de recherche est décrit au chapitre VIII. Rappelons ses fonctionnalités:

- Recherche du symbole courant dans la liste de valeurs associée.
- En cas de succès, émission de l'adresse de la chaîne de bits liée à cette valeur.

Avant de fournir la description IRENE du module, nous allons présenter son environnement, permettant de justifier certains points de la description.

IX.4.1. Environnement

Comme tous les modules élémentaires du filtre, le module de recherche est relié au bus local de l'assemblage. Ce bus sera connecté aux broches données du module. Pour des considérations liées au nombre limité de broches d'un boîtier, il est nécessaire de réutiliser ces broches de données pour les données issues de la mémoire. Aussi un dispositif externe permet de connecter et déconnecter le module du bus local, l'autorisant ainsi soit à recevoir des messages soit à accéder à sa mémoire.



Les bits 7 à 9 du bus adresse sont en fait des fils de contrôle, mais ce "découpage manuel" est dû aux limitations actuelles des outils (uniformité des tailles des différents registres, ce qui entraînerait d'utiliser un registre de 16 bits pour quelques fils de contrôle).

IX.4.2. Description IRENE

On expose ci-dessous la description IRENE du module. On peut remarquer que le circuit décrit a deux niveaux de séquencement ce qui entraînera deux niveaux de partie contrôle.

```
    module  recherche (in  ;
                       out adr <15:0> ;
                       bi buf <15:0> ) ;

  reg ptl <15:0> , long <15:0> ,
      symb <15:0> , res <15:0>, elt <15:0>,
      fsup, finf, ffin, flong, bsuc ;

  sequence main : $scrute, $initchgt, $chgtcode,
                  $lusymbole, $initrec, $recherche, $sync,
                  $echec, $succes ; general
    sequence phase : $t1, $t2, $t3;  general
    endseq
  endseq

  begin
```

```
          ! Selection des points d'entree !
$scrute   : $t1: adr <9:9> := 1 /          !Connecte a l'exterieur!
                 case buf <3:0> of
                    8: goto $lusymbole,
                    9: goto $initrec,
                   10: goto $sync,
                   11: goto $initchgt
                    : goto $scrute
                 endcase ;

          ! Reception du symbole courant !
$lusymbole: $t1: symb <15:0> <- buf <15:0> /
                 goto $scrute  ;

          ! Initialisation de la recherche !
$initrec  : $t1: ptl <15:0> <- buf <6:0> /    !Debut liste!
                 long <15:0> <- buf <12:7> ; !Longueur liste!
            $t2: adr <9:9> := 0 /             !Deconnecte exterie
                 adr <8:7> := 3 /             !Memoire CS et R!
                 adr <6:0> := ptl <6:0> /     !Adresse memoire!
                 long <15:0> := long <15:0> - 1 ;
            $t3: elt <15:0> <- buf <15:0> /   !Donnee memoire!
                 ffin <- long <15:0> >> 0 /
                 goto $recherche ;

          ! Boucle de recherche !
$recherche : $t1: fsup <- symb <15:0> >> elt <15:0> /
                 ptl <15:0> <- ptl <15:0> - 1 ;
            $t2: finf <- symb <15:0> << elt <15:0> /
                 long <15:0> <- long <15:0> - 1 /
                 adr <6:0> := ptl <6:0> /
                 adr <8:7> := 3 ;             !Memoire CS et R!
            $t3: if finf                      !symbole inferieur!
                   then
                     if ffin                  !fin de la liste!
                       then goto $echec
                       else goto $recherche !element suivant!
                     endif
                   else                       !superieur ou egal!
                     if fsup                  !superieur!
                       then goto $echec
                       else goto $succes      !egal!
                     endif
                 endif /
                 ffin <- long <15:0> >> 0 /
                 elt <15:0> <- buf <15:0> ; !Elt. suivant!

           ! Emission message d'echec !
$echec    : $t1: bsuc <- 0 /
                 adr <9:9> := 1 ;           !Reconnecte exterieur!
            $t2: if buf <3:0> <> 3          !Attente invite!
                   then goto $t2
                 endif ;
            $t3: buf <3:0> := 14 /          !Emission echec!
                 goto $scrute ;

           ! Emission message succes !
$succes   : $t1: bsuc <- 1 /
                 adr <9:9> := 1 ;           !Reconnecte exterieur
```

```
                $t2: if buf <3:0> <> 3              !Attente invite!
                       then goto $t2
                     endif ;
                $t3: buf <3:0> := 15 /              !Emission succes!
                     buf <10:4> := ptl <6:0> /      !Adr. chaine de bits!
                     goto $scrute ;

                 ! Emission message synchronisation !
$sync      : $t1: if buf <3:0> <> 3                 !Attente invite!
                       then goto $t1
                     endif ;
                $t2: if bsuc
                        then buf <3:0> := 14        !Emission echec!
                        else buf <3:0> := 15        !Emission succes!
                     endif /
                     goto $scrute ;

                 ! Initialisations chargement !
$initchgt  : $t1: long <15:0> <- buf <15:0> /       !Longueur code!
                     ptl <15:0> <- 256 /            !Fin memoire!
                     goto $chgtcode ;

                 ! Chargement des listes !
$chgtcode  : $t1: adr <6:0> := ptl <6:0> /
                     adr <8:7> := 2 /               !Memoire CS et WI
                     long <15:0> <- long <15:0> - 1 ;
             $t2: ptl <15:0> <- ptl <15:0> - 1 /
                     ffin <- long <15:0> >> 0 ;
             $t3: if ffin
                        then goto $scrute
                        else goto $chgtcode
                     endif ;

end. !fin de recherche!
```

Cette description met en évidence la simplicité du système de communications, grâce aux notions d'émissions et réceptions de message. En effet aucun dispositif de synchronisation n'est nécessaire pour la coopération des différents modules, devant simplement partager une horloge commune.

IX.4.3. Partie Opérative générée

Nous exposons dans l'annexe 7 l'ensemble des actions opératives et de contrôle extraites de la description en IRENE. Nous ne présentons ci-dessous que le schèma logique de la partie opérative. Rappelons qu'APOLLON utilise un modèle architectural composé de deux bus auxquels sont connectés les différents éléments. Suivant les actions à effectuer en parallèle il générera plusieurs sous-parties opératives (deux pour l'exemple, figure IX.3).

La trace permet de visualiser pour chaque étape (se déroulant sur un microcycle), les actions exécutées, les opérations effectuées sur les différentes sous-parties opératives et enfin les transferts réalisés sur les phases "phi1" et "phi2". Nous la fournissons dans l'annexe 8.
Notons que le découpage en champ n'est pas encore implanté.

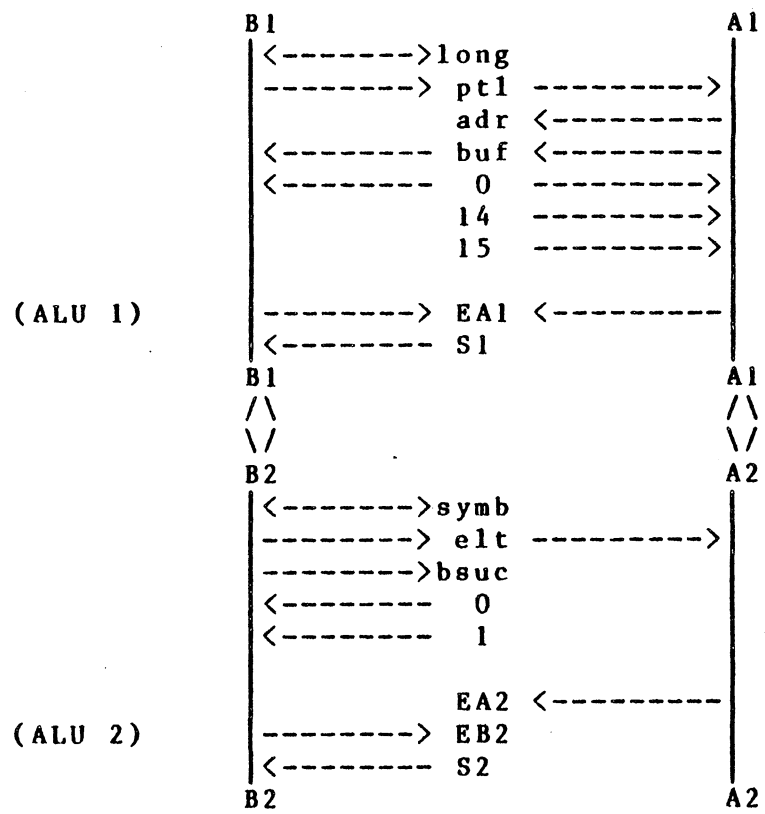


Figure IX.3

L'étape suivante consisterait en la génération du fichier de description des masques, en langage LUCIE à l'aide de l'assembleur LUBRICK. Nous n'effectuerons pas cette étape car quelques problèmes restent à résoudre quant à la possibilité de compiler certaines opérations: accès par champ, réalisation des entrées/sorties. Toutefois précisons que la génération des masques, à partir de ce dernier niveau de description, est actuellement automatique.

IX.5. Conclusions

Par le développement de cet exemple, on espère avoir montré que l'élaboration d'une maquette matérielle pouvait d'ores et déjà s'envisager. Pour ce faire deux points sont à développer:

- La simulation au plus bas niveau matériel.

- L'intégration de chaque module.

Pour la simulation matérielle on peut, soit reprendre l'outil développé pour la simulation fonctionnelle en lui intégrant les notions matérielles nécessaires (notamment signaux de contrôle), soit utiliser un langage supportant le parallélisme (par exemple OCCAM).

Pour l'intégration de chaque module, l'effort à fournir est essentiellement lié à la réalisation de la partie contrôle. On peut espérer que les outils permettant une conception automatique, au moins pour des partie contrôles simples et à peu de niveaux, seront sous peu disponibles.

Notons que l'utilisation de la chaîne de compilation confirme les espérances que l'on peut placer en un tel outil. Espérons que les travaux actuellement en cours pourront être menés à terme, pour prouver que pour bien des applications, un circuit aux performances modestes suffit là où l'implantation logicielle de l'algorithme s'avérait par contre insuffisante. Qui plus est, la complexité des architectures devient telle qu'il faut pouvoir "sous-traiter" à des outils de conception des opérations de plus en sophistiquées. Il devient donc impératif de développer ces outils et de les utiliser, même si dans un premier temps leurs résultats peuvent paraître médiocres, afin de les faire évoluer en des éléments de conception qui deviendront indispensables.

CONCLUSIONS

CONCLUSIONS

Au terme de cette thèse, nous pensons que l'algorithme d'unification développé peut devenir un des éléments essentiels d'une machine base de connaissances PROLOG. Pour cela il reste à implanter la stratégie de recherche adéquate (interprétation par ensemble de solutions) permettant d'exploiter au mieux les possibilités de cet algorithme. En effet rappelons que les mesures sur l'implantation logicielle ont montré un gain proche d'un facteur 10, ce sur une version non optimale du code, car orientée vers une simulation matérielle. Les performances, que l'on peut espérer d'une version matérielle, permettent de répondre favorablement quant au bien fondé de la réalisation d'un "circuit d'unification".

Une extension de ces travaux consisterait à concevoir une machine PROLOG basée sur ce mécanisme d'unification. Pourtant l'utilisation du filtre telle que nous l'avons définie n'est pas immédiatement envisageable, d'une part car les hypothèses liées au contexte base de connaissances ne sont plus vérifiées pour des programmes plus généraux et d'autre part le temps de compilation annulerait le gain obtenu par l'utilisation du filtre. L'idée, présentée et développée par Gilles Berger-Sabbatel dans [BERb 85] (chapitre XI), est "d'inverser" le processus, c'est à dire de compiler les en-têtes des clauses constituant les programmes et filtrer les buts en attente de résolution. Dès lors le temps de compilation est supprimé, puisque la compilation devient une phase précédant celle d'exécution, et les mesures effectuées sur différents programmes PROLOG [BERb 85] montrent que la préunification dans ce sens assure une sélectivité importante. Notons que cette approche d'une machine PROLOG nécessite néanmoins de développer les autres primitives de l'interprétation, l'unification ne représentant qu'une partie du temps total d'exécution.

Pour ce qui est de la suite du projet, la conception d'un "chip" assumant les fonctions du filtre logiciel est le premier objectif. Nous avons abordé cettte phase de réalisation avec la

volonté d'exploiter au mieux le parallélisme exprimé dans l'algorithme d'unification, jugeant cette démarche nécessaire pour atteindre les objectifs de rapidité.

Une première étape doit aboutir à la réalisation d'une maquette matérielle composée d'autant de circuits que de primitives définies dans l'algorithme d'unification. Cette maquette permettra de valider l'architecture proposée et plus particulièrement le protocole de communications et la mise en oeuvre d'une coopération par envois de messages.

Une deuxième étape pourra alors débuter devant aboutir sur l'intégration en un seul circuit des différents éléments constituants la maquette. Nous serons alors confrontés aux problèmes posés par l'introduction de parallélisme à l'intérieur d'un même circuit. Pour l'heure, seules des formes extrêmes de parallélisme sont exploitées:

- L'une, que l'on qualifiera de "bas niveau", englobe les différents niveaux de pipe-line ainsi que le déroulement simultané des actions de la partie opérative.

- A l'opposé, un parallélisme de "haut niveau", introduit récemment dans les circuits, vise l'intégration en un même circuit non seulement de l'unité centrale mais également de plusieurs fonctions jusqu'alors réalisées par des boitiers externes, voire par du logiciel: canaux d'entrées-sorties, contrôleur de DMA, modes d'adressage...

Par contre une hiérarchisation de différents éléments permettant la mise en oeuvre d'un parallélisme "intermédiaire" n'est pas encore maîtrisée. Un axe de recherche serait de développer l'esquisse de méthodologie amorcée au chapitre VII, non seulement pour l'appliquer au filtre (l'exemple étant relativement simple, toutes les difficultés ne sont pas mises à jour), mais pour construire une théorie plus complète, un exemple plus complexe pouvant être la conception d'une machine orientée objet.

ANNEXES

ANNEXE 1

Base de connaissances généalogique

Un exemple d'aplication où se mélangent étroitement clauses unaires et "non-unaires", est celui de la représentation et utilisation d'un arbre généalogique. En effet choisissons de définir les relations de base (celles exprimées en extension) "père" et "mère"

```
Père ( _ , _ ) -> ;          Mère ( _ , _ ) -> ;
        .                            .
```

On peut alors définir tout un ensemble de liens familiaux:

```
Famille (X,Y) -> Père (X,Y) ;
Famille (X,Y) -> Mère (X,Y) ;
Famille (X,Y) -> Frère (X,Y) ;
Famille (X,Y) -> Soeur (X,Y) ;
Famille (X,Y) -> Oncle (X,Y) ;
Famille (X,Y) -> Tante (X,Y) ;
Famille (X,Y) -> Cousin (X,Y) ;
Famille (X,Y) -> Cousine (X,Y) ;
      .                .

Frère (X,Y) -> Homme (X) Parent (Z,X) Parent (Z,Y) ;
Soeur (X,Y) -> Femme (X) Parent (Z,X) Parent (Z,Y) ;
      .                .

Avec:
Parent (X,Y) -> Père (X,Y) ;
Parent (X,Y) -> Mère (X,Y) ;
```

Toutes ces relations peuvent se réécrire en appels aux relations de base (le sexe d'une personne pouvant être associé à son nom).

Par contre la définition générale de la descendance se fait à l'aide du prédicat ancêtre:

```
Ancêtre (X,Y) -> Parent (X,Y) ;
Ancêtre (X,Y) -> Parent (X,Y) Ancêtre (X,Y) ;
```

Notons que la définition de cette relation est récursive, ce qui peut entraîner des difficultés quant à son évaluation dans notre contexte, si l'on ne prend pas de précautions particulières.

ANNEXE 2

Représentation interne des clauses

On rappelle (cf chapitre II, paragraphe 3) qu'un symbole est codé sous format interne à l'interpréteur, ce par l'intermédiaire d'un dictionnaire. L'adresse de l'item décrivant le symbole sert de nom interne. Cet item est constitué des champs suivants:

- Sens qui précise le type du symbole:
  - . atome => nil
  - . prédicat utilisateur => valeur > 0
  - . prédicat évaluable : valeur < -1000
  - . variable globale : -1000 < valeur < 0
- Modèle fournit un pointeur vers la strcuture du terme, structure stockée en fin de table des clauses.
- Hcode contient le hcode du symbole, calculé sur le nom externe de ce symbole, permettant d'accélerer l'interface nom externe -> nom interne.
- Nom externe est un ponteur sur l'espace des chaînes permettant d'accéder au nom externe du symbole.

Les clauses sont stockées dans la table des clauses sous forme de listes. Le symbole prédéfini "." a deux arguments sert de constructeur de liste. Pour une représentation plus claire on prendra les conventions suivantes:

- Une clause est représent[e sur une ligne.
- Le premier élément de chaque ligne est un pointeur sur l'alternative suivante d'un paquet de clause (NIL si la dernière).
- Le deuxième élément est le nombre de varaibles de la clause.
- Ensuite vient le constructeur de liste que l'on notera ".", alors que pour les autres symboles on indiquera leur arité (octet de type), suivi du pointeur vers le dictionnaire. Les variables sont identifiées par un octet de type négatif.

Exemple

Grand-père (X,Y) -> Père (X,Z) Père (Z,Y) ;

Grand-père (X,Y) -> Père (X,Z) Mère (Z,Y) ;

Père (Michel,Véronique) -> ;

Père (Colette,Véronique) -> ;

Mère (Véronique,Sylvain) -> ;

	Sens	Modèle	Hcode	Nom_ext		Espace chaîne
1	1	101	-	---	-->	Grand-père
2	3	102	-	---	-->	Père
3	4	103	-	---	-->	Mère
4	NIL	NIL	-	---	-->	Michel
5	NIL	NIL	-	---	-->	Véronique
6	NIL	NIL	-	---	-->	Colette
7	NIL	NIL	-	---	-->	Sylvain

Table des clauses

1	2	3	.	v -1	v -2	2 2	v -1	v -3	.	2	2
				v -3	v -2	NIL					
2	NIL	3	.	v -1	v -2	2 2	v -1	v -3	.	2	2
				v -3	v -2	NIL					
3	NIL	0	.	2 2	0 4	0 5	NIL				
4	5	0	.	2 3	0 6	0 5	NIL				
5	NIL	0	.	2 3	0 5	0 7	NIL				

101	NIL	2	.	2 1	v -1	v -2	NIL
102	NIL	2	.	2 2	v -1	v -2	NIL
103	NIL	2	.	2 3	v -1	v -2	NIL

ANNEXE 3

Construction de la structure englobante

On fournit ci-dessous le corps de l'algorithme. Précisons qu'un noeud de la structure englobante est constitué des adresses des noeuds fils, frère, père, de l'adresse et de la longueur de la liste de valeurs associée à ce noeud. De plus un indicateur se substitution signale l'éventuel début d'une substitution.

```
Racine_struct = NIL , NIL , NIL , NIL ;
Racine_cible = Racine de la 1ere cible ;
indice_cible = 0 ;

tantque ( Racine_cible != NIL  &&  filtre non plein ) faire
début
  noeud_struct = Racine_struct ;
  noeud_cible = Racine_cible ;

  faire
     pt_val = stocke_valeur ( Noeud_cible , Noeud_struct -> ptr_liste )
     maj_indice ( pt_val , indice_cible_cour ) ;

     faire
        transition = éval_trans_préfixé ( Noeud_cible ) ;
        si ( noeud_struct -> transition == NIL )
           creer ( transition , noeud_struct ) ;
        noeud_cible = parcours ( transition , noeud_cible ) ;
        noeud_struct = parcours ( transition , noeud_struct ) ;
     tantque ( noeud_cible marqué  &&  noeud_cible != Racine_cible ) ;

     marque ( noeud_cible ) ;
  tantque ( noeud_cible != Racine_cible ) ;

  Racine_cible = cible_suivante ( ++ indice_cible ) ;
fin tantque
```

ANNEXE 4

Génération des transitions

La génération des transitions, issues du parcours de la structure de l'en-tête de clause en cours de préunification, s'effectue en analysant l'arité de chaque symbole constituant cet en-tête (en-tête qui est un terme). Deux cas se présentent:

- Soit le symbole courant est symbole fonctionnel "non-zéroaire". Alors on empile son arité, mémorisant ainsi le nombre d'arguments à lire pour terminer l'analyse de ce terme, et on génère une transition fils permettant de se déplacer sur le noeud de la structure englobante matérialisant le premier argument.

- Dans tous les autres cas on est en présence d'un atome. Son arité étant nulle, on ne l'empile pas. Cet atome représente un argument du terme englobant. On décrémente alors le sommet de pile, mémorisant à tout instant de l'algorithme le nombre d'argument à reconnaître pour terminer l'analyse du terme englobant. Dès lors, un des deux cas suivants se présentent:
  . soit la valeyr du sommet de pile ainsi décrémentée est différente de zéro. Alors on génère une transition frère car d'autres arguments du terme englobant sont à reconnaître.
  . soit cette valeur de sommet de pile devient nulle. Celà signifie que le dernier argument du symbole englobant vient d'être lu. Alors on génère une transition père signifiant qu'il faut remonter à ce noeud englobant. On dépile alors ce sommet nul et on se retrouve dans le cas de la lecture d'un atome, le terme englobant constituant pour son propre père un argument atomique.

  La boucle de génération des transitions "père" s'arrête soit sur la génération d'une transition "frère", soit sur le fait que la pile devienne vide, ceci signifiant que l'on a reconnu tous les arguments du terme racine.

Une écriture "algorithmique" est la suivante:

```
          /* En début d'analyse la pile est vide */
Si (arité != 0)                    /* Arité du symbole courant */
   empiler (arité) ;
   génére (TRANS_FILS) ;           /* Génération FILS */
   retourne (non_racine) ;
Fin_si
          /* Tous les autres cas */
Tanque ( ! pile_vide )             /* Boucle pour PERE */
   val (sommet_pile) -- ;          /* Décrément du sommet */
   si ( val (sommet_pile) != 0 )   /* Encore des frères */
      génére (TRANS_FRERE) ;       /* Génération FRERE */
      retourne (non_racine) ;
   Fin_si
          /* Remonte au père */
   génére (PERE) ;
   dépiler () ;                    /* Sommet nul */
Fin_tq
```

ANNEXE 5

Exemple de compilation

Cibles à compiler

```
cible ( type1 , arg1 , arg2 ( a ) ) -> ;
cible ( type1 , arg1 , arg2 ( b ) ) -> ;
cible ( type2 , niveau1 ( *x ) , arg2 ( *x ) ) -> ;
cible ( type2 , niveau1 ( niveau2 ( *x )) , arg2 ( a )) -> ;
cible ( type2 , niveau1 ( niveau2 ( niveau3 ( *x ))) , arg2 ( c ))->;
cible ( type2 , niveau1 ( niveau2 ( niveau3 ( nil ))), arg2 ( d ))->;
```

Automate structure

	FILS	FRERE	PERE	SUB	LG	DEB
0	-1	1	-1	0	2	0
1	4	2	-1	0	2	2
2	3	-1	-2	0	1	10
3	-1	-1	2	1	5	11
4	5	-1	1	1	2	4
5	6	-1	4	1	2	6
6	-1	-1	5	1	2	8

listes contigues

	termes
0	type1
1	type2
2	arg1
3	niveau1
4	*x0
5	niveau2
6	*x0
7	niveau3
8	*x0
9	nil
10	arg2
11	*x0
12	a
13	b
14	c
15	d

chaines de bits

	indices
0	110000
1	001111
2	110000
3	001111
4	001000
5	001111
6	001100
7	001111
8	001110
9	001111
10	111111
11	001000
12	101100
13	011000
14	001010
15	001001

Debut pos

Cible	Debut	Unif
0	0	0
1	4	0
2	8	1
3	13	0
4	19	0
5	26	0

tableau positions

	pos	terme
0	0	typel
1	1	argl
2	2	arg2
3	3	a
	-	-------
4	0	typel
5	1	argl
6	2	arg2
7	3	b
	-	-------
8	0	type2
9	1	niveaul
10	4	v_dis *x0
11	2	arg2
12	3	v_dis *x0
	-	-------
13	0	type2
14	1	niveaul
15	4	niveau2
16	5	v_dis *x0
17	2	arg2
18	3	a
	-	-------
19	0	type2
20	1	niveaul
21	4	niveau2
22	5	niveau3
23	6	v_dis *x0
24	2	arg2
25	3	c
	-	-------
26	0	type2
27	1	niveaul
28	4	niveau2
29	5	niveau3
30	6	nil
31	2	arg2
32	3	d

ANNEXE 6

Mesures de compilation

Cette annexe présente les mesures effectuées sur trois cas de compilation mettant en évidence les différents paramètres importants. Pour chaque ensembles de cibles une boucle de 10000 compilations est effectuée. Pour chaque boucle on fournira d'une part le détail des mesures relevées, et d'autre part le pourcentage global des temps d'exécutions des procédures générant les codes :

- PARCOURS : compil_partiel + comp_p_arg
  (Création structure)

- RECHERCHE : symb_stocke + (transfert / 2)
  (Listes de valeurs)

- ET : maj_indice + ldiv + lrem +
  (Matrice de bits) (transfert / 2)

Les fonctions re_init_compil et jointure sont utilisées pour réinitialiser la boucle de compilation et on peut donc les décompter du temps total.
D'autre part les fonctions nargbytes et mcount sont également à décompter car elles servent aux mesures et non à la compilation.
On appellera temps réel total le temps de traitement obtenu aprés les décomptes cités ci-dessus (les fonctions n'atteignant pas 1% du temps d'exécution ont été supprimées des listings mais pris en compte pour le temps réel total) .

Remarque: la fonction symb_stocke, corps de la création des listes de valeurs, a été optimisée ce qui a permis de diviser par deux son temps d'exécution. Une telle amélioration est envisageable pour les autres fonctions et doit être prise en compte quant à la comparaison des temps des diverses fonctions.

Génération d'une liste de 60 symboles

Structure des cibles : t (*x , *y) ;

Avec : *x = ai 0 < i <= 60
(distribution aléatoire des ai)
*y libre.

%time	cumsecs	#call	ms/call	name
29.1	132.58	600000	0.22	_symb_stocke
17.8	213.91	10000	8.13	_compil_partiel
16.5	289.34	600000	0.13	_comp_p_arg
7.0	321.23	600000	0.05	_maj_indice
5.6	346.99			ldiv
5.3	371.40			lrem
4.9	393.54	20000	1.11	_transfert
3.7	410.64	10000	1.71	_re_init_compil
2.6	422.72	600000	0.02	_jointure
2.4	433.65			_nargbytes
1.8	441.89			mcount
0.7	445.31	30000	0.11	_insere_tete
0.6	447.93			lmul
0.5	450.39			_var_stocke
0.3	451.95			_exit
0.2	452.91	10000	0.10	_fin_compil
0.1	454.71	10000	0.05	_code_noeud

Temps total réel : 40.8 ms. (100 %)

PARCOURS : 15.7 ms (38 %)
RECHERCHE : 14.2 ms (34 %)
ET : 8.2 ms (20 %)

Divers : 2.6 ms (8 %)

Génération de 2 listes de 60 symboles

Structure des cibles : t (*x0 , *x1 , *y) ;

Avec : *x0 = ai 0 < i <= 60
*x1 = bj 0 < j <= 60
(distribution aleatoire des ai et bj)
*y libre

%time	cumsecs	#call	ms/call	name
28.5	192.21	1200000	0.16	_symb_stocke
22.2	342.19	1200000	0.12	_comp_p_arg
10.6	413.92	10000	7.17	_compil_partiel
9.2	475.97	1200000	0.05	_maj_indice
7.3	524.96			ldiv
6.8	570.95			lrem
3.2	592.27	20000	1.07	_transfert
2.8	610.91			_nargbytes
2.5	628.00	10000	1.71	_re_init_compil
2.0	641.28			mcount
1.8	653.38	600000	0.02	_jointure
0.9	659.48	70000	0.09	_insere_tete
0.7	664.04			_var_stocke
0.7	668.46			lmul
0.2	669.86			_exit
0.1	670.76	10000	0.09	_fin_compil

Temps total réel : 61.3 ms (100 %)

PARCOURS : 22.1 ms (35 %)
RECHERCHE : 20.4 ms (33 %)
ET : 16.8 ms (27 %)

Divers : 3.0 ms (5 %)

Création d'une matrice de bits complexe

Cibles compilées :

```
plus ( *x , a , b ) ;
moins ( *x , arg1 , arg2 ) ;
div ( *x , dividende , diviseur ) ;
mul ( *x , plicande , plicateur ) ;
exp ( *x , base , exposant ) ;
log ( *x , base , exposant ) ;
plus ( div ( mult ( a,b) , c) , moins ( c,d) , *x ) ;
opl ( *z , exp ( *x , basel , expl ) , log ( *y , base2 , exp2)) ;
operation ( plus ( exp ( a,b,c ) , d ) , moins ( log ( a,b,c ) , e )) ;
racine ( mul ( a,b,c ) , div ( c,d,e ) ) ;
sinus ( *x , exp ( *x , plus ( *y,b,c ) , mul ( *z,e,f )) ) ;
```

%time	cumsecs	#call	ms/call	name
19.3	74.79	110000	0.68	_comp_p_arg
12.8	124.66	250000	0.20	_transfert
10.0	163.38	640000	0.06	_symb_stocke
9.8	201.63	780000	0.05	_maj_indice
8.5	234.50			ldiv
7.8	264.82			lrem
5.9	287.84	10000	2.30	_re_init_compil
5.1	307.63	370000	0.05	_insere_tete
3.2	319.91			_nargbytes
3.0	331.57	10000	1.17	_compil_partiel
2.7	341.93	10000	1.04	_fin_compil
2.6	351.91			mcount
1.7	358.65	140000	0.05	_rec_var_p
1.6	364.92	140000	0.04	_var_stocke
1.6	371.10	250000	0.02	_code_noeud
1.6	377.12	80000	0.08	_empile_bit
0.7	380.00	50000	0.06	_maj_pile_bit
0.7	382.78			lmul
0.7	385.34	110000	0.02	_jointure

Temps total réel	: 35.2	ms	(100 %)
PARCOURS	: 8.6	ms	(25 %)
RECHERCHE	: 8.8	ms	(26 %)
ET	: 13.6	ms	(39 %)
Divers	: 4.2	ms	(10 %)

ANNEXE 7

Résultats issus de la chaîne de compilation

On reprend la description du module de recherche exposée au chapitre IX pour montrer les résultats des différentes étapes de la génération de la partie opérative.

```
    module  recherche (in  ;
                    out adr <15:0> ;
                    bi buf <15:0> ) ;

  reg ptl <15:0> , long <15:0> ,
      symb <15:0> , res <15:0>, elt <15:0>,
      fsup, finf, ffin, flong, bsuc ;

  sequence main : $scrute, $initchgt, $chgtcode,
                  $lusymbole, $initrec, $recherche, $sync,
                  $echec, $succes ; general
    sequence phase : $t1, $t2, $t3;  general
    endseq
  endseq

  begin
            ! Selection des points d'entree !
  $scrute   : $t1: adr <9:9> := 1 /        !Connecte a l'exterieur!
                  case buf <3:0> of
                     8: goto $lusymbole,
                     9: goto $initrec,
                    10: goto $sync,
                    11: goto $initchgt
                      : goto $scrute
                  endcase ;

              ! Reception du symbole courant !
  $lusymbole: $t1: symb <15:0> <- buf <15:0> /
                  goto $scrute  ;

              ! Initialisation de la recherche !
  $initrec    : $t1: ptl <15:0> <- buf <6:0> /    !Debut liste!
                     long <15:0> <- buf <12:7> ; !Longueur liste!
                $t2: adr <9:9> := 0 /             !Deconnecte exterieur!
                     adr <8:7> := 3 /             !Memoire CS et R!
                     adr <6:0> := ptl <6:0> /     !Adresse memoire!
                     long <15:0> := long <15:0> - 1 ;
                $t3: elt <15:0> <- buf <15:0> /   !Donnee memoire!
                     ffin <- long <15:0> >> 0 /
                     goto $recherche ;

              ! Boucle de recherche !
  $recherche : $t1: fsup <- symb <15:0> >> elt <15:0> /
                    ptl <15:0> <- ptl <15:0> - 1 ;
               $t2: finf <- symb <15:0> << elt <15:0> /
                    long <15:0> <- long <15:0> - 1 /
```

```
                adr <6:0> := ptl <6:0> /
                adr <8:7> := 3 ;                !Memoire CS et R!
            $t3: if finf                        !symbole inferieur!
                   then
                     if ffin                    !fin de la liste!
                       then goto $echec
                       else goto $recherche     !element suivant!
                     endif
                   else                         !superieur ou egal!
                     if fsup                    !superieur!
                       then goto $echec
                       else goto $succes        !egal!
                     endif
                endif /
                ffin <- long <15:0> >> 0 /
                elt <15:0> <- buf <15:0> ; !Elt. suivant!

                 ! Emission message d'echec !
$echec      : $t1: bsuc <- 0 /
                adr <9:9> := 1 ;                !Reconnecte exterieur!
              $t2: if buf <3:0> <> 3            !Attente invite!
                     then goto $t2
                endif ;
              $t3: buf <3:0> := 14 /            !Emission echec!
                goto $scrute ;

                 ! Emission message succes !
$succes     : $t1: bsuc <- 1 /
                adr <9:9> := 1 ;                !Reconnecte exterieur!
              $t2: if buf <3:0> <> 3            !Attente invite!
                     then goto $t2
                endif ;
              $t3: buf <3:0> := 15 /            !Emission succes!
                buf <10:4> := ptl <6:0> /       !Adr. chaine de bits!
                goto $scrute ;

                 ! Emission message synchronisation !
$sync       : $t1: if buf <3:0> <> 3            !Attente invite!
                     then goto $t1
                endif ;
              $t2: if bsuc
                     then buf <3:0> := 14       !Emission echec!
                     else buf <3:0> := 15       !Emission succes!
                endif /
                goto $scrute ;

                 ! Initialisations chargement !
$initchgt   : $t1: long <15:0> <- buf <15:0> / !Longueur code!
                ptl <15:0> <- 256 /             !Fin memoire!
                goto $chgtcode ;
```

```
                ! Chargement des listes !
  $chgtcode  : $t1: adr <6:0> := ptl <6:0> /
                    adr <8:7> := 2 /              !Memoire CS et WI
                    long <15:0> <- long <15:0> - 1 ;
               $t2: ptl <15:0> <- ptl <15:0> - 1 /
                    ffin <- long <15:0> >> 0 ;
               $t3: if ffin
                       then goto $scrute
                       else goto $chgtcode
                    endif ;

end. !fin de recherche!
```

A partir de cette description, le premier outil génère les actions opératives liées aux différentes étapes de l'algorithme. Les conventions suivantes sont adoptées pour leur représentation:

- L'étiquette spécifie le numéro d'étape.
- (A1 | A2 | ... | An) signifie que **l'une** des actions sera à effectuer en cette étape.
- A1 / A2 / ... / An signifie que **toutes** les actions seront à effectuer en cette étape (en parallèle).
- (| | ... |) signifie que l'étape ne fait appel à aucune action opérative, mais seulement à des actions de contrôle.

```
    $e0.0 :  adr <9:9> := 1 / ( |  |  |  | ) ;
$e0.1 :  symb <15:0> <- buf <15:0> ;
$e0.2 :  ptl <15:0> <- buf <6:0> /
         long <15:0> <- buf <12:7> ;
$e0.3 :  adr <9:9> := 0 /
         adr <8:7> := 3 /
         adr <6:0> := ptl <6:0> /
         long <15:0> := long <15:0> - 1 ;
$e0.4 :  elt <15:0> <-  buf <15:0> /
         ffin <- long <15:0> >>0 ;
$e0.5 :  fsup <- symb <15:0> >> elt <15:0> /
         ptl <15:0> <-  ptl <15:0> - 1 ;
$e0.6 :  finf <- symb <15:0> << elt <15:0> /
         long <15:0> <- long <15:0> - 1 /
         adr <6:0> := ptl <6:0> /
         adr <8:7> := 3 ;
$e0.7 :  ( ( | ) |  ( | ) ) /
         ffin <- long <15:0> >> 0 /
         elt <15:0> <- buf <15:0> ;
$e0.8 :  bsuc <- 0 /
         adr <9:9> := 1 ;
$e0.9 :  ( | ) ;
$e0.10 :  buf <3:0> := 14 ;
$e0.11 :  bsuc <- 1 /
          adr <9:9> := 1 ;
$e0.12 :  ( | ) ;
$e0.13 :  buf <3:0> := 15 /
          buf <10:4> := ptl <6:0>  ;
$e0.14 :  ( | ) ;
```

```
$e0.15 :  ( buf <3:0> := 14 | buf <3:0> := 15) ;
$e0.16 :  long <15:0> <- buf <15:0> /
          ptl <15:0> <- 256 ;
$e0.17 :  adr <6:0> := ptl <6:0> /
          adr <8:7> := 2 /
          long <15:0> <- long <15:0> - 1 ;
$e0.18 :  ptl <15:0> <- ptl <15:0> - 1 /
          ffin <- long <15:0> >> 0 ;
$e0.19 :  ( | ) ;
```

Deux niveaux d'actions de contrôle sont issus de l'algorithme, correspondant aux deux niveaux de séquencement déclarés en spécifications.
Le niveau le plus bas, pilotant la partie opérative, est décrit par l'ensemble d'actions suivantes où:

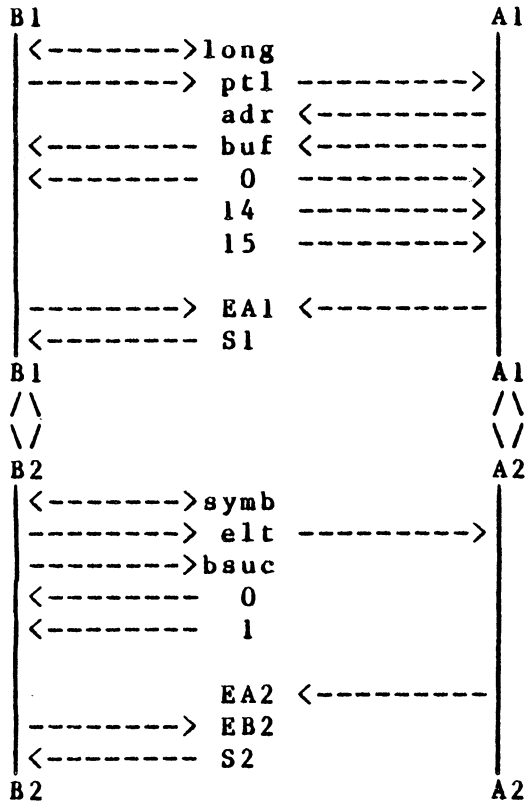
- "(C1 | C2 ... | Cn : A1 | A2 ... | An)" signifie que l'action correspondant à la condition évaluée à vrai sera réalisée.
- "goto étiquette" signifie un séquencement.
- "activate étiquette" spécifie l'activation d'une étape du niveau inférieur.
- "stop" indique, pour le niveau supérieur, la fin d'une séquence d'étapes.

```
    $scrute.t1    :  stop ;

$lusymbol.t1 :  activate $e0.1 / stop ;

$initrec.t1  :  activate $e0.2 ;
$initrec.t2  :  activate $e0.3 ;
$initrec.t3  :  activate $e0.4 / stop ;

$recherch.t1 :  activate $e0.5  ;
$recherch.t2 :  activate $e0.6  ;
$recherch.t3 :  activate $e0.7 | : / stop ;

$echec.t1    :  activate $e0.8 ;
$echec.t2    :  (buf <3:0> <> 3 | :  goto $echec.t2 | ) ;
$echec.t3    :  activate $e0.10 / stop ;

$succes.t1   :  activate $e0.11 ;
$succes.t2   :  (buf <3:0> <> 3 | :  goto $succes.t2 | ) ;
$succes.t3   :  activate $e0.13 / stop ;

$sync.t1     :  (buf <3:0> <> 3 | :  goto $sync.t1 | ) ;
$sync.t2     :  (bsuc | :  activate $e0.15 | ) /
                activate $e0.15 / stop ;

$initchgt.t1 :  activate $e0.16 / stop ;

$chgtcode.t1 :  activate $e0.17 ;
$chgtcode.t2 :  activate $e0.18 ;
$chgtcode.t3 :  stop ;
```

Le niveau de séquencement supérieur est décrit en terme d'activations d étapes de la machine précédemment décrite (les mêmes conventions de notation sont reprises):

```
$scrute : ( buf <3:0> = 8 | 9 | 10 | 11 | :
          goto $lusymbol |
          goto $initrec |
          goto $sync |
          goto $initchgt |
          goto $scrute) ;
$lusymbol  : goto $scrute / activate $lusymbol.tl ;
$initrec   : activate $initrec.tl / goto $recherch ;
$recherch  : activate $recherch.tl /
             (finf | :
                 (ffin | :  goto $echec | goto $recherch) |
                 (fsup | :  goto $echec |  goto $succes)) ;
$echec     : activate $echec.tl / goto $scrute ;
$succes    : activate $succes.tl / goto $scrute ;
$sync      : activate $sync.tl / goto $scrute ;
$initchgt  : goto $chgtcode / activate $initchgt.tl  ;
$chgtcode  : activate $chgtcode.tl /
              (ffin | :  goto $scrute | goto $chgtcode) ;
```

A partir des actions opératives, APOLLON génère la partie opérative exposée ci dessous, ainsi que la trace des différentes actions (une action se déroule en un microcycle constitué de deux phases notées "phi1" et "phi2"). Rappelons qu'APOLLON utilise un modèle architectural composé de deux bus auxquels sont connectés les différents éléments. Suivant les actions à effectuer en parallèle il générera plusieurs sous-parties opératives (deux pour l'exemple).



La trace permet de visualiser pour chaque étape (se déroulant sur un microcycle), les actions exécutées, les opérations effectuées sur les différentes sous-parties opératives et enfin les transferts réalisés sur les phases "phi1" et "phi2".
Notons que le découpage en champ n'est pas encore implanté.

```
    etape No 1 :   adr <-- 1
   phi2          1   --->  A1 --->  adr

etape No 2 :   symb <-- buf
   phi2         buf  --->  B1
   phi2         B1   --->  B2 ---> symb

etape No 3 :   ptl <-- buf     long <-- buf
   phi2         buf  --->  B1 ---> long    ptl

etape No 4 :   adr <-- ptl     long <-- decr long s-pop 1 : decr
   phi1        long  --->  B1 --->  EA1
   phi2         ptl  --->  A1 --->  adr
   phi2         S1   --->  B1 ---> long

etape No 5 :   elt <-- buf     >0 long s-pop 1 : >0
   phi1        long  --->  B1 --->  EA1
   phi2         buf  --->  B1
   phi2         B1   --->  B2 --->  elt

etape No 6 :    symb > elt     ptl <-- decr ptl s-pop 1 : decr
```

```
s-pop 2 : >
   phi1          pt1   --->  A1 --->  EA1
   phi1          elt   --->  A2 --->  EA2
   phi1         symb   --->  B2 --->  EB2
   phi2          S1    --->  B1 --->  pt1

etape No 7 :   symb < elt     long <-- decr long     adr <-- pt1 s-
pop 1 : decr     s-pop 2 : <
   phi1         long   --->  B1 --->  EA1
   phi1          elt   --->  A2 --->  EA2
   phi1         symb   --->  B2 --->  EB2
   phi2          pt1   --->  A1 --->  adr
   phi2          S1    --->  B1 ---> long

etape No 8 :   bsuc <-- 0     adr <-- 1
   phi2           0    --->  A1 --->  adr
   phi2           0    --->  B2 ---> bsuc

etape No 9 :   buf <-- 14
   phi2          14    --->  A1 --->  buf

etape No 10 :   bsuc <-- 1     adr <-- 1
   phi2           0    --->  A1 --->  adr
   phi2           1    --->  B2 ---> bsuc

etape No 11 :   buf <-- 15
   phi2          15    --->  A1 --->  buf

etape No 12 :   buf <-- pt1
   phi2          pt1   --->  A1 --->  buf

etape No 13 :   long <-- buf     pt1 <-- 0
   phi1          buf   --->  B1 ---> long
   phi2           0    --->  B1 --->  pt1

etape No 14 :   pt1 <-- decr pt1     >0 long s-pop 1 : >0      s-pop
2 : decr
   phi1          pt1   --->  A1
   phi1         long   --->  B1 --->  EA1
   phi1          A1    --->  A2 --->  EA2
   phi2          B2    --->  B1 --->  pt1
   phi2          S2    --->  B2
```

Pour ce qui est de la génération de la partie contrôle, seul un "maillon" de la chaîne de traitement reste à terminer pour aboutir à l'automatisation complète, à partir des informations extraites de la description IRENE.

BIBLIOGRAPHIE

BIBLIOGRAPHIE

[ANC 83] : F. Anceau
"CAPRI: a design methodology and a silicon compiler for VLSI circuits specified by algorithms"
Third CALTECH conference on VLSI, Mars 1983

[AVRa 81]: B. Avron, E.A. Feigenbaum
"The hanbook of artificial intelligence"
Volume 1, p160-171
Heuris Tech Press, California, 1981

[AVRb 81]: B. Avron, E.A. Feigenbaum
"The hanbook of artificial intelligence"
Volume 1, chapitre II : "Search"
Heuris Tech Press, California, 1981

[BAN 80] : F. Bancilhon, M. Scholl
"Le filtrage des données dans la MBD VERSO"
INRIA, Journées MBD Sophia Antipolis, 10-12 sept 1980

[BAN 81] : F. Bancilhon, P. Richard, M. Scholl
"The relationnal Database Machine VERSO: binary operations"
INRIA, 6ème Workshop, 9-11 juin 1981

[BD3 83] : Rapport du groupe BD3
"Bases de Données: nouvelles perspectives"
INRIA, Janvier 1983

[BEK 83] : Y. Bekkers, B. Canet, O. Ridoux, L. Ungaro
"Problèmes d'implementation du langage PROLOG en vue de la réalisation d'une machine PROLOG"
Rapport Final ATP, publi. interne Octobre 1983

[BER 82] : G. Berger-Sabbatel, G.T. Nguyen
"Motivations et principes pour une MBD PROLOG"
IMAG/INPG, RR no 339, Décembre 1982

[BER 84] : G. Berger-Sabbatel, JC. Ianeselli
"Un unificateur câblé pour la MBD PROLOG OPALE"
IMAG/TIM3 Rapport de Recherche No: Novembre 1984

[BERa 85]: G. Berger-Sabbatel, W. Dang, JC. Ianeselli
"Matériels et logiciels pour la cinquième g[nération"
Congrés AFCET, Paris, Mars 1985

[BERb 85]: G.Berger-Sabbatel
"Machines spécialisées et programmation logique"
INPG, Thése d'état (à paraître)

[BIG1 83]: 1 ère journées d'éetude sur les Langages Orientés Objets
Cap d'agde, Octobre 1983

[BIG2 84]: 2 ème journées d'étude sur les Langages Orientés Objets
Brest, Novembre 1984

[BIR 73] : G. Birtwistle, O. Dhal, B. Myrharig, K. Nygaard
SIMULA BEGIN Petrocelli/Charter
New-York, 1973

[BOU 84] : E. Bourcier
"Conception et réalisation du simulateur du langage de description de circuits intégrés IRENE-B"
Mémoire ingénieur CNAM, Université Grenoble, Octobre 1984

[BOW 82] : K.A. Bowen, R.A. Kowalski
"Amalgating language and metalanguage in logic programming"
Logic Programming 1982, Academic Press

[CHA 82] : S. Chakravarthy, J. Minker, D. Tran
"Interfacing predicate logic languages and relationnal DB"
10 th logic programming conférence 1982

[CHI 84] : T. Chikayama
"ESP référence manual"
ICOT TR 44, Février 1984

[CHU 84] : B. Chuquillanqui
"Une nouvelle approche pour l'optimisation topologique et l'autoamtisation de u dessin des masques de PLA complexes"
Thèse docteur/ingénieur, INPG, Octobre 1084

[CLA 81] : K.L. Clark, S. Gregory
"A relationnal language for parallel programming"
Communications ACM, Octobre 1981

[CLA 82] : K.L. Clark, F.G. McCabe, S. Gregory
"IC-PROLOG language features"
"Logic programming"
Academic Press Inc.

[CLA 83] : K.L. Clark, S. Gregory
PARLOG: a parallel logic programming language
Research Report DOC 83/5 (May 1983)
Londres, Imperial College of Science and Tecnology

[CLO 81] : W.F. Clocksin, C.S. Mellish
"Programming in PROLOG"
Springer Verlag 1981

[COD 70] : E.F. Codd
"A relationnal model for large shared data bases"
ACM 13, 6 Juin 1970

[COD 72] : E.F. Codd
"Data base system"
Prentice Hall 1972

[COL 79] : A.Colmerauer, H.Kanoui, M. Van Caneghen
"Etude et réalisation d'un système PROLOG"
Rapport IRIA/SESORI no: 77030
Rapport groupe Intelligence Artificielle
Université Aix Marseille II, 1979

[COL 83] : A.Colmerauer, H.Kanoui, M. Van Caneghen
PROLOG, Bases théoriques et développement actuels
TSI 1983, Vol. 1, No 4

[COMP 84] : Computer Architecture, Technical Comitte
Newsletter
Septembre 1984

[DAN 84] : W. Dang
Note interne OPALE-R No 2

[DAN 85] : W. Dang
Note interne

[DAH 82] : V. Dahl
"On database systems development trough logic"
ACM Mars 1982, Vol. 7, No 1

[DEF 73] : CR. De Fiore, PB. Berrer
"A Data Base Management System utilising an associative memory"
AFIPS, Vol 42 juin 1973

[DEL 78] : C. Delobel
"Normalization and hierarchical dependencies in the relationnal data model"
ACM Transactions Databases Systems, Vol 3, No 3, 1978

[DEL 82] : C. Delobel, M. Adiba
"Bases de donnéeset systèmes relationnels"
Dunod informatique, Bordas Paris 1982

[DELb 82]: "Chemins d'accès"
[DEL 82]

[DELc 82] : "organisations en B_arbre et exploitation"
[DEL 82]

[DIJ 76] : EW. Dijkstra
"A discipline of programming"
Prentice HAll 1976

[DIN 80] : M. Dincbas
"Le système de résolution de problèmes METALOG"
Rapport Final 3146/DERI 1980

[DON 84] : P. Donz
"D-PROLOG"

[FEI 82] : E.A. Feigenbhaum, P.M. Corduck
"La cinquième génération"
InterEditions, Paris 1982

[GAL 78] : H. Gallaire, J. Minker, J.M. Nicolas
"an overview and introduction to logic and data base"
Logic and Data Base, Aout 1978

[GAL 82] : H. Gallaire, C. Lasserre
"Metalevel control for logic programs"
Logic programming 1982, Academic Press

[GAL 83] : H. Gallaire, J. Minker, JM. Nicolas
"Logic ans database: a deductive approach"
ACM computing survey Vol 15, No 3, Septembre 1983

[GAR 80] : G. Gardarin, P. Valduriez
"Algorithme multiprocesseur et jointure de relations"
INRIA, Journées MBD, Sophia Antipolis, 10-12 Sept 1980

[GRO 83] : R.R. Gross
"silicon compilers: a critical survey"
Université de Caroline du Nord
Chapell Hill, Mai 1983

[GUY 81] : A.Guyot, A.A. Jerraya, J. Raymond
Présentation de LUCIE
IMAG Grenoble, Juin 81 RR

[ING 78] : D.H. Ingalls
"The SMALLTAK 76 programming system design and implementation"
5 éme symposium ACM sur la PPL
Tucson, Janvier 1978

[JAM 85] : R. Jamier, A.A. Jerraya
"APOLLON: a datapath silicon compiler"
Papier soumis à ICCD '85

[KOW 74] : R. Kowalski
"Predicate logic as programming language"
IFIP 74

[LAU 82] : J.L. Lauriere
"Représentation et utilisation de la connaissance"
TSI 1982, Vol. 1, No 1 et 2

[LEG 84] : B. Legeard
"Le langage PROLOG"
Microsystemes Juillet 1984, No 44

[LIN 76] : CS. Lin, DC. Smith, JM. Smith
"The design of rotative associative memory for relationnal database applications"
ACM Transactions on Data Base Systems
Vol 1, No 1, Mars 1976

[MAR 83] : S. Marine, F. Anceau, K. Jahidi
"IRENE: un langage de description de circuits intégrès logiques"
IMAG, Rapport de Recherche No 356, Mars 1983

[MAR 85] : S. Marine, E. Bourcier
"IRENE: a language for the description of VLSI digital hardware"
Papier soumis à ICCD '85

[MAR 85] : F. Martinez
"Utilisation du compilateur IRENE"
Note interne, Mars 1985

[MIN 78] : J. Minker
"An experimental relational data base system based on logic"
Logic and data base, Aout 1978

[NIC 78] : J.M. Nicolas, H. Gallaire
"Theory vs. Interpretation"
Logic and data base, Aout 1978

[OBR 82] : M. Obrebska
"Etude comparative de différentes méthodes de conception des Parties Contrôles de Microprocesseurs"
Thèse docteur/ingénieur, INPG, Juin 1982

[PER 79] : L.M. Pereira, A. Porto
"Intelligent backtracking and sidetracking in Horn clause programs: the theory"
Université de Lisbonne, Report 2/79

[PER 79] : L.M. Pereira, A. Porto
"Intelligent backtracking and sidetracking in Horn clause programs: the implementation"
Université de Lisbonne, Report 13/79

[REI 78] : R. Reiter
"Deduction question-answering on relational data Bases"
Logic and data base, Aout 1978

[REM 84] : C. Remy
"Les CI à l'Arsenure de Gallium"
Microsystèmes no: 44, Juin 84

[REM 85] : C. Remy
"L'ordinateur biologique"
Microsystèmes no: 49, Janvier 85

[RHO 80] : J. Rhomer
"Machines et langages pour traiter les ensemble de données"
INP Grenoble, Thèse état, Décembre 1980

[ROB 65] : J.A. Robinson
"A model oriented logic based on the resolution principle"
JACM 12, 1, Décembre 1965

[ROCa 84]: C. Roche
"EAQUE-LRO Génération de systèmes experts"
Thèse de 3ème cycle p63-75, INPG Juillet 1984

[ROU 79] : P. Roussel
"PROLOG: manuel de référence et d'utilisateur"
Rapport groupe Intelligence Artificielle
Université Aix Marseille II, 1979

[SCH 83] : J.P. Schoellkopf
"LUBRICK: a silicon assembler and its application
to data-path design for FISC"
VLSI 83, Trondheim Norway, Août 1983

[SCH 78] : SA. Schuster, MS. Nguyen, EA. Ozkaraher, KC. Smith
"RAP2 an associative processor for Database"
5th Symposium on computer architecture 1978

[SHA 83] : E. Shapiro
"A subset of Concurent PROLOG and its interpreter"
ICOT, TR003 Février 1983

[SWA1 77]: R.J. Swan, S.H. Fuller, D.P. Siewiorek
"Cm a modular multi-microprocessor"
National Computer Conference
1977

[SWA2 77]: R.J. Swan, A. Bechtolskeim, K.W. Lai, J.K. Ousterhout
"The implementation of the Cm multi-microprocessor"
National Computer Conference
1977

[TOH 84] : Tohu Moto-oka
"Les ordinateurs de la cinquième génération"
La Recherche Avril 1984, No: 154

[TRE 82] : P. Treleaven
"VLSI processor architectures"
IEEE Trans. Computer, June 1982

[YAZ 84] : K. Yazdanian
"Deductions dans les BD relationnelles, fondement logique et mise en oeuvre"
BD Sécurité et intelligence, CNAM
Paris 23 et 24 Octobre 84
Logic and data base, Aout 1978

[VAU 78] : B. Vauquois
"Calculabilité des langages"
IMAG, Cours polycopié, 1978

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974

VU le rapport de présentation de Monsieur F. ANCEAU, Professeur

Monsieur IANESELLI Jean-Christophe

est autorisé à présenter une thèse en soutenance en vue de l'obtention du titre de DOCTEUR de TROISIEME CYCLE, spécialité "Informatique".

Fait à Grenoble, le 6 mai 1985

Le Président de l'I.N.P.-G

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,