Programmation parallèle et programmation fonctionnelle : propositions pour un langage

Maria del Pilar Cisneros Gascon

THESE

présentée à

l' Institut National Polytechnique de Grenoble

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

par

Maria del Pilar CISNEROS GASCON

PROGRAMMATION PARALLELE
ET
PROGRAMMATION FONCTIONNELLE:

PROPOSITIONS POUR UN LANGAGE.

Thèse soutenue le 31 octobre 1984 devant la commission d'examen.

J. MOSSIERE	Président
D. BERT Ph. JORRAND T. MUNTEAN J. VOIRON	Examinateurs

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

ECOLE NATIONALE SUPERIEURE DES MINES DE SAINT-ETIENNE

Directeur	: Monsieur M. MERMET
Directeur des Etudes et de la formation	: Monsieur J. LEVASSEUR
Directeur des recherches	: Monsieur J. LEVY
Secrétaire Général	: Mademoiselle M. CLERGUE

Professeurs de 1ère Catégorie

COINDE	Alexandre	Gestion
GOUX	Claude	Métallurgie
LEVY	Jacques	Métallurgie
LOWYS	Jean-Pierre	Physique
MATHON	Albert	Gestion
RIEU	Jean	Mécanique - Résistance des matériaux
SOUSTELLE	Michel	Chimie
FORMERY	Philippe	Mathématiques Appliquées

Professeurs de 2ème catégorie

HABIB	Michel	Informatique
PERRIN	Michel	Géologie
VERCHERY	Georges	Matériaux
TOUCHARD	Bernard	Physique Industrielle

Directeur de recherche

LESBATS	Pierre	Métallurgie

Maîtres de recherche

BISCONDI	Michel	Métallurgie
DAVOINE	Philippe	Géologie
FOURDEUX	Angeline	Métallurgie
LOBYLANSKI	André	Métallurgie
MALAUZE	René	Chimie
MANCELOT	Francis	Chimie
DE COZE	Jean	Métallurgie
THEVENOT	François	Chimie
TRAN MINH	Canh	Chimie

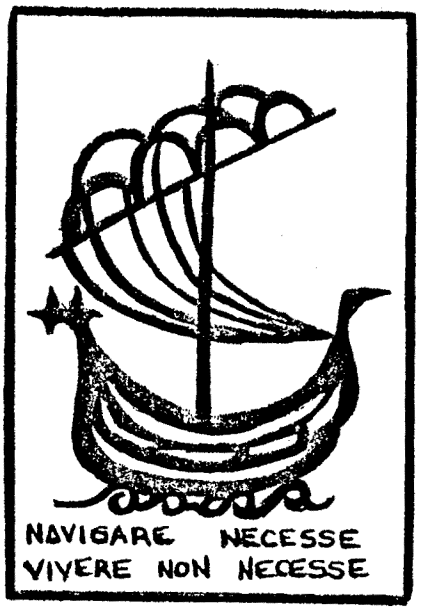
Personnalités habilitées à diriger des travaux de recherche

BRIVER	Julian	Métallurgie
VUILHOT	Bernard	Chimie
THOMAS	Gérard	Chimie

Professeur à l'UER de Sciences de Saint-Etienne

VERGNAUD	Jean-Maurice	Chimie des Matériaux & chimie industrielle

****



A mis papás
con todo cariño y gratitud

Je voudrais remercier,

Monsieur Jacques Mossière, Professeur à l'Institut National Polytechnique de Grenoble d'avoir accepté de présider le jury de cette thèse.

Monsieur Didier Bert, Chargé de Recherche au CNRS, pour avoir bien voulu juger mon travail en me faisant des critiques très utiles.

Monsieur Traian Muntean et Monsieur Jacques Voiron, Maîtres Assistants à l'USMG, de leur participation à mon jury.

Je suis également très reconnaissante à tous les membres du jury pour l'intérêt qu'ils ont manifesté pour mon travail.

Monsieur Philippe Jorrand, Directeur de Recherche au CNRS, pour la confiance qu'il m'a témoignée en m'accueillant dans son équipe et pour le soutien professionnel et humain qu'il m'a prodigué. Le travail présenté dans cette thèse est le fruit de ses orientations et critiques toujours pertinentes; je voudrais lui exprimer toute ma gratitude.

Je tiens aussi à montrer ma reconnaissance à Jorge Vidart, Professeur à l'Université "Simon Bolivar" de Caracas (Venezuela), qui a dirigé mes travaux antérieurs. Je lui dois particulièrement l'orientation vers l'équipe où cette thèse a été réalisée et un encouragement constant pendant les premières années de mon travail. Il a travaillé pendant son année sabbatique avec J.M. Pereira, Ph. Jorrand et moi même dans les premières définitions du langage proposé dans cette thèse.

Cette thèse a profité de l'expérience et des observations de mes collègues d'équipe J.M. Pereira, E. Arkaxhiu et P. Lagnier. Parmi mes collègues, je tiens à remercier tout particulièrement Sylvie Rogé avec laquelle j'ai entretenu de longues et riches discussions; j'ai un seul regret cependant: l'avoir connue un peu tard car elle arrivait quand je m'en allais.

Je ne peux pas oublier, au chapitre des remerciements, Orieta Santana qui avec beaucoup de patience et une grande efficacité a réalisé la frappe et la mise au point de cette thèse.

Ainsi de même tous ceux qui ont contribué à l'amelioration "littéraire" de mon travail: PH. Jorrand, D. Keita, R. Caferra et S. Rogé.

Enfin, je remercie tous mes camarades du troisième étage du bâtiment D pour cette atmosphère quotidienne de sympathie.

Cette recherche a été effectuée pendant que j'étais boursière du Centre Régional des Oeuvres Universitaires et Scolaires (CROUS France) et du Ministerio de Education (Venezuela); sans cette aide financière, ce travail n'aurait pu être accompli.

TABLE DES MATIERES

-ooo-

page

INTRODUCTION 5

I - LANGAGES DE PROGRAMMATION: LA NOUVELLE GENERATION 7

1. PROGRAMMATION FONCTIONNELLE 11
   1.1. Evolution Historique des langages: les langages applicatifs. 11
   1.2. Quelques langages applicatifs 13
      1.2.1. Les "fondateurs"
      1.2.2. FP: Style Fonctionnel contre Style Von Neumann.
      1.2.3. Les plus récents.
   1.3. Conclusion 19

2. PROGRAMMATION PARALLELE 21
   2.1. L'introduction de "la communication" dans les langages de programmation: motivation. 21
   2.2. Quelques langages de programmation parallèle. 23
      2.2.1. Communicating Sequential Processes (CSP).
      2.2.2. Introduire la communication ne suffit pas: Systèmes Parallèles Communicants.
      2.2.3. A Calculus of Communicating Systems (CCS).
   2.3. Conclusion 32

3. PROGRAMMATION PARALLELE ET PROGRAMMATION FONCTIONNELLE: est ce possible? 33

page

II- DEUX SYSTEMES SOURCE: SPARC ET LPG 37

1. SPECIFICATION ALGEBRIQUE ET QUELQUES DEFINITIONS 40

2. SPARC : Systèmes PARallèles Communicants 48
2.1. Le projet SPARC 48
2.1.1. Qu'est-ce que SPARC?
2.1.2. SPARC : Etat de l'art.
2.1.3. Caractéristiques de SPARC.
2.2. Description d'un processus communicant. 52
2.2.1. Exemples
2.3. Construction de réseaux de processus 61
2.3.1. L'opérateur d'union
2.3.2. L'opérateur de connexion
2.3.3. L'opérateur d'abstraction
2.4. Description formelle. 71
2.4.1. Les processus comme algèbres de termes.
2.4.1.1 Définitions Préliminaires
2.4.1.2 Termes Fonctionnels
2.4.1.3 Termes d'Etat
2.4.1.4 Termes d'événement
2.4.1.5 Termes de transition
2.4.1.6 Syntaxe et Sémantique de Processus
2.4.2. Algèbre de processus
2.4.2.1 Définitions préliminaires
2.4.2.2 Union de processus
2.4.2.3 Connexion de processus
2.4.2.4 Abstraction de processus
2.4.3. Généricité.
2.5. SPARC par rapport à d'autres projets. Conclusion. 82

3. LPG : Langage de Programmation Générique 84
3.1. Présentation Générale du Langage 84
3.1.1. Définition d'un type abstrait
3.1.2. Définition de fonction (enrichissement)

page

3.1.3. Définition de Propriété
3.1.4. Unités Génériques
3.1.5. Conclusion
3.2. Description algébrique des types et propriétés: syntaxe et sémantique. 94

III FP2: UN LANGAGE POUR LES PROCESSUS COMMUNICANTS 99

1. INTEGRATION DE LPG A FP2 102

2. PRESENTATION DU LANGAGE FP2 104
2.1. Généralités 104
2.2. Définition du Langage 105
2.2.1. Généricité de processus
2.2.2. Spécification d'un processus
2.2.3. Réseaux de processus : Expressions
2.3. Carte Syntaxique de FP2. 114

3. EXEMPLES 117
3.1. Exemple No.1: FONCTION MULTIPLICATION 117
3.2. Exemple No.2: TRADUCTEUR GENERIQUE 119
3.3. Exemple No.3: PARTAGE D'UNE RESSOURCE 126
3.4. Exemple No.4: RESEAUX SYSTOLIQUES. Calcul en parallèle d'une suite vectorielle. 134

4. EXTENSIONS A FP2 143
4.1. Formes Fonctionnelles 143
4.2. Les processus comme paramètres: Processus Formels.

CONCLUSION 155
ANNEXE 1 : CARTE SYNTAXIQUE DE LPG 159
BIBLIOGRAPHIE 165

INTRODUCTION

La nécessité d'unifier le traitement de la communication et le souhait d'un gain du temps en mettant en parallèle des entités indépendantes, ont conduit les travaux de recherche en logiciel vers la programmation parallèle. Ce style de programmation considère donc les notions d'entrée, de sortie, de communication, de parallélisme, de synchronisation et les intègre aux langages de programmation afin de permettre la description des comportements de systèmes de processus parallèles communicants.

D'autre part, la complexité croissante des langages de programmation mène à une remise en question de la machine traditionnelle: l'ordinateur de Von Neumann, en proposant la conception d'une architecture différente. Cette voie de recherche conduit à la programmation applicative.

Le sujet de cette thèse est la proposition d'un langage de programmation parallèle-applicatif. Le langage proposé, FP2, est issu d'une part du système SPARC (Systèmes PARallèles Communicants), permettant ainsi d'exprimer tout ce que concerne la communication, mais aussi du langage LPG, permettant ainsi de définir des types abstraits et des fonctions éventuellement génériques.

Dans notre premier chapitre, nous faisons un survol de l'histoire des langages applicatifs et parallèles, en suivant leur évolution jusqu'à arriver à ceux qui sont les plus importants à l'heure actuelle. Nous présentons aussi une discussion sur la possibilité de réunir ces deux styles de programmation parallèle et applicative sans trop de sacrifice de part et d'autre.

Le deuxième chapitre est consacré à l'exposé des deux systèmes à partir desquels est né le langage FP2. Le premier SPARC, dont on parle d'abord d'une façon intuitive, puis d'une façon formelle. Le deuxième est LPG (Langage de Programmation Générique) qui est un langage applicatif pour la spécification de types abstraits et de fonctions permettant la généricité. On montre les interprétations sémantiques de ces deux systèmes qui sont parfaitement compatibles: tous les deux sont fondés sur une sémantique algébrique.

Dans notre troisième et dernière chapitre, nous présentons le langage parallèle-applicatif FP2. Il est issu des concepts de SPARC et on intègre le langage LPG pour obtenir la puissance des descriptions de types et de fonctions. Après une description de la syntaxe, nous présentons quatre exemples programmés en FP2 à travers desquels nous montrons les différentes caractéristiques du langage et sa puissance quant au calcul des structures communicantes. Pour finir, nous suggérons deux extensions possibles à faire au langage FP2, qui une fois implémentées, enrichiront beaucoup les possibilités d'expression et de construction du langage.

CHAPITRE I

LANGAGES DE PROGRAMMATION: LA NOUVELLE GENERATION

CHAPITRE I

LANGAGES DE PROGRAMMATION: LA NOUVELLE GENERATION

Le développement des langages de programmation à partir des années 60 a été riche. Selon les divers besoins et objectifs, l'évolution des langages prend différentes directions.

Ainsi, certains se proposent d'introduire la notion de communication dans les langages : ils définissent des entités communicantes avec entrées, sorties, mécanismes d'échanges... Des notions comme le non déterminisme, la simultaneité, le parallélisme, la synchronisation sont considérées, étudiées et mises en oeuvre.

D'autres constatent que la complexité croissante des langages existants entraîne un coût de logiciel qui croît démesurément par rapport au coût du matériel : ils consacrent, devant ce fait, leurs efforts à la création de langages qui cernent davantage les spécifications formelles du problème et qui font abstraction du fonctionnement interne de la machine au moment de l'exécution. Le développement d'un tel type de langage suscite à son tour la conception d'architectures hardware appropriées, profitant des possibilités de VLSI et spécifiquement agencées pour exécuter ces langages.

Le besoin de méthodes simples et efficaces de vérification de programmes d'une part, et la nécessité d'outils pour "raisonner" sur les programmes d'autre part, poussent à concevoir des langages fondés sur la logique mathématique.

Une contrainte au moment de la programmation est de ne pouvoir manipuler que des objets de types déjà prédéfinis, ou

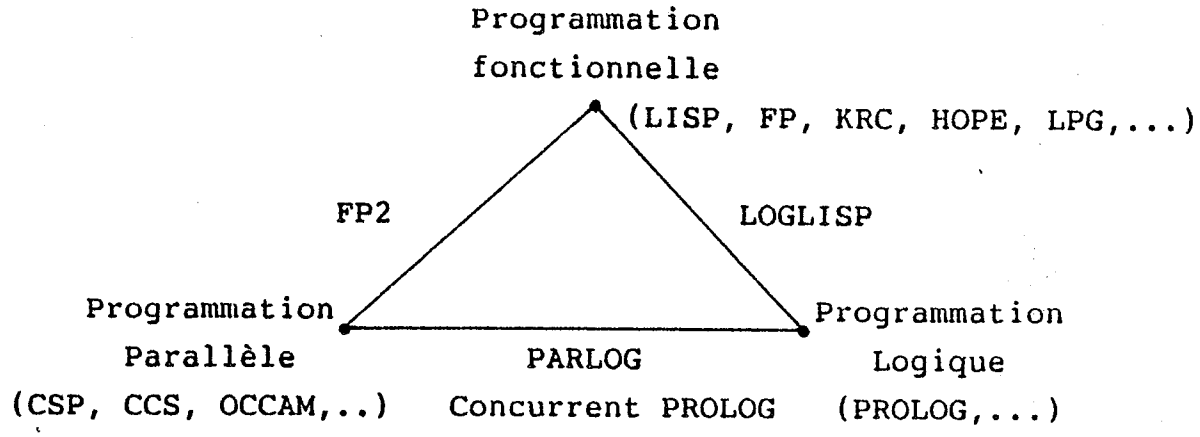
bien, si la possibilité de définir les types est offerte, il faut alors définir leur représentation. C'est l'introduction dans les langages d'une possibilité de définir des types abstraits, de les rendre génériques, qui constitue l'objectif de certains chercheurs.

La nouvelle génération de langages regroupe tous les langages développés en réponse aux questions que nous venons d'évoquer. Trois styles de programmation résument ces nouveaux langages et sont actuellement considérés comme les plus prometteurs :

- la programmation parallèle : CSP <Hoa 78>, CCS <Mil 80>, OCCAM <May 83> et beaucoup d'autres.
- la programmation fonctionnelle : FP <Bac 78>, KRC <Tur 81a, 81b, 81c, 81d>, HOPE <Bur-Mac 81> et beaucoup d'autres.
- la programmation logique : PROLOG <Col-Kan 83> et pas beaucoup d'autres.

Il existe aussi des langages mixtes regroupant dans un même cadre cohérent les caractéristiques d'au moins deux de ces styles "purs". Les langages PARLOG <Cla-Gre 83> et Concurrent PROLOG <Sha 83> par exemple, réunissent la programmation logique et la programmation parallèle et, le langage LOGLISP <Rob-Sib 82>, la programmation logique et la programmation fonctionnelle.

Le langage proposé dans cette thèse, FP2, est un langage fonctionnel pour décrire la communication et le parallélisme. Dans <Jor 84b>, cette classification de styles de programmation est exprimée graphiquement :



1. PROGRAMMATION FONCTIONNELLE

1.1. Evolution Historique des langages : les langages applicatifs.

L'architecture de l'ordinateur digital actuel a été conçue par John Von Neumann dans les années 1940, et cette configuration est restée inalterée jusqu'à nos jours. Elle consiste en une mémoire grande et passive, associée à un seul processeur qui agit sur la mémoire en changeant le contenu d'un mot à la fois. Le programme d'un tel ordinateur consiste en de longues séries de telles actualisations qui doivent être ordonnées dans le temps pour leur exécution.

Les langages de programmation conventionnels reflètent cette architecture. L'élément de base de programmation dans ces langages est l'affectation, instruction qui correspond à l'opération d'actualisation de l'ordinateur de Von Neumann, sur lequel ces langages vont être implémentés. Ainsi, il existe une structure de contrôle qui ordonne les instructions d'affectation dans le temps pour construire un programme. Cela fait des langages séquentiels impératifs où l'action de base est l'affectation.

Au milieu des années 1950 émergent des langages de "haut niveau": FORTRAN, COBOL, PL/1,..., l'évolution continue ..., PASCAL, ALGOL68 mais tous ont comme constante celle d'être des versions complexes de la même machine. Même si les différences entre FORTRAN et ALGOL68 sont considérables, le problème est qu'ils sont tous fondés sur l'ordinateur de Von Neumann.

Il y a deux raisons qui font penser à un style différent de programmation :

1) La crise du logiciel par rapport au matériel : le coût de production et de maintenance du logiciel d'un ordinateur est de plus en plus élevé, surtout quand on le compare à la baisse du coût du matériel. Dans cette situation, il devient attractif de considérer une classe de langages qui économisent le temps de programmation en faisant des programmes plus courts et plus transparents, même en détriment du temps d'exécution.

2) Le développement de la technologie VLSI et les nouvelles possibilités que cela implique au niveau du matériel. Aujourd'hui, processeur et mémoire peuvent être construits avec la même technologie : VLSI-"chips". Ainsi, au lieu d'avoir un seul processeur avec une grande mémoire passive, on pourrait réaliser une architecture qui supporte une grande collection de processeurs, et si de plus on les met à travailler en parallèle, la limitation de vitesse imposée par le "bottleneck" de la machine de Von Neumann pourrait être dépassée. Des recherches dans ce sens sont en cours <Den 79>, <Gur-Wat 80>, <Cla-Gla 80>, <Ack-Den 79>, <Arv-Gos 78>.

Les langages qui appartiennent à un tel style de programmation doivent :
- avoir des fondements qui permettent des descriptions mathématiques élégantes et concises.
- ne pas avoir l'affectation comme instruction (pas d'états), ni

un contrôle nécessairement séquentiel.
- avoir une sémantique de réduction à la place de transition d'états.
- pouvoir constuire des programmes qui expriment clairement l'expression d'un processus ou d'un calcul.

Parmi ces langages, on distingue essentiellement les langages fonctionnels ou applicatifs où les programmes consistent en un ensemble d'équations qui définissent une ou plusieurs fonctions et/ou structures de données que l'on veut calculer.

1.2. Quelques langages applicatifs

1.2.1. Les "fondateurs"

Les premiers langages répondant à un modèle applicatif ont été le Lambda Calcul de Church <Chu 41>, le Système de Combinateurs de Curry <Cur 58> et LISP pur <Mcc 60>. Tous les trois sont caracterisés par :

leurs fondements : concis et "utiles".
leur mémoire : pas de mémoire, pas d'affectations.
leur sémantique : pas d'états, sémantique par réduction.
leur clarté de programmation : programmes conceptuellement simples.

Le Lambda Calcul et le Système de Combinateurs de Curry sont des systèmes mathématiques pour représenter la notion d'application de fonctions. La plupart des systèmes applicatifs conçus par la suite se servent de l'opération de substitution du Lambda Calcul comme opération de base.

LISP est un langage conçu par John McCarthy à la suite de son article : "Recursive Functions of Symbolic Expressions and their Computation by Machine" <Mcc 60>. C'est un langage formel mathématique dont les principales caractéristiques, qui le

différencient des autres langages de son époque, peuvent se ramener à trois :

1. En LISP, toutes les données sont exprimées au moyen d'expressions symboliques (S-expressions) de longueur indéfinie ayant une structure d'arbre qui permet d'isoler rapidement les sous expressions significatives.
2. LISP, lui même, est le langage source pour spécifier la façon de traiter les S-expressions.
3. LISP interprète et exécute des programmes écrits en forme de S-expressions. Ainsi, comme le langage machine, il peut être utilisé pour générer des programmes.

Parmi ces trois langages, LISP est le seul à être utilisé comme langage de programmation et implémenté sur un ordinateur <Mcc 62>. Mais, ces "fondateurs" bien que extrêmement puissants quant à la théorie, présentent des inconvénients <Bac 78> :

- les règles de substitution dont dépend la sémantique du Lambda Calcul sont simples à définir, mais leurs implications sont difficiles à saisir pleinement. La complexité de ces règles n'est pas largement reconnue, par contre, les différentes preuves faites par des logiciens du théorème de Church-Rosser échouent au moment d'expliquer certaines de ces complexités. Le théorème de Church-Rosser ou la preuve de Scott de l'existence d'un modèle <Sco 72> sont nécessaires pour montrer que le Lambda Calcul a une sémantique consistente. La définition de LISP pur a même eu une erreur de ce genre : le problème de "funarg".
- L'implémentation de l'opération de substitution du Lambda Calcul entraine de grandes difficultés. Quand on essaie d'introduire la mémoire et d'améliorer l'efficacité, les systèmes applicatifs commencent à plonger dans le système de Von Neumann. C'est pourquoi LISP pur est fréquemment dénaturé au moment de l'implémenter dans la machine de Von Neumann.

1.2.2. FP: Style Fonctionnel contre Style Von Neumann.

En 1978, John Backus publie un article <Bac 78> qui a un triple propos :

1. Déclarer publiquement l'infériorité, du point de vu de la programmation abstraite, des langages soumis à une architecture Von Neumann. Même si dix huit ans en arrière on travaillait déjà dans un style fonctionnel, c'est dans cet article, au moyen d'une argumentation détaillée, qu'on proclame la nécessité d'une rupture avec une configuration d'ordinateur perimée.

2. Proposer un style de programmation fonctionnelle fondé sur l'utilisation de "formes fonctionnelles" pour créer des programmes. Les programmes fonctionnels traitent des données structurées, ils sont souvent non répétitifs et non récursifs, ils sont hiérarchiquement construits, ils ne nomment pas leurs arguments et ils n'ont pas besoin de la machinerie complexe de déclarations de procédures pour être généralement applicable.
Les formes fonctionnelles peuvent partir de programmes de haut niveau pour construire d'autres programmes d'encore plus haut niveau, ce qui les langages conventionnels, ne fournissent pas d'une manière aussi propre.

3. Associé à un style fonctionnel de programmation, il propose une algèbre de programmes dont la portée des variables est le programme, et dont les opérations sont les formes combinées. Cette algèbre peut être utilisée pour faire des transformations de programmes et pour résoudre des équations où les inconnues sont des programmes. Ces transformations sont définies par des lois algébriques. Les formes combinées sont introduites non seulement en raison de la puissance qu'elles rendent à la programmation, mais aussi pour la puissance des lois algébriques associées. Les théorèmes généraux de cette algèbre expriment le comportement détaillé et les conditions de terminaison d'une

large classe de programmes.

Les inconvénients des langages "fondateurs" exposés au paragraphe 1.2.1 sont surmontés par FFP (version formelle des systèmes FP) car il n'y a pas des variables et on n'a qu'une seule règle de substitution élémentaire (une fonction par son nom). D'autre part, on peut prouver la consistence de sa sémantique au moyen d'un argument de point fixe relativement simple suivant les lignes développées par Dana Scott et Manna <Man 73>. Pour cette preuve consulter <Mcj 75>.

1.2.3. Les plus récents.

Parmi les langages applicatifs les plus récents, il y en a deux sur lesquels l'attention des chercheurs se concentre actuellement, en raison de leur puissance, propreté et élégance : HOPE <Bur-Mac 81> et KRC <Tur 81a> et <Tur 81b>. Un programme en HOPE ou KRC est un ensemble d'équations éventuellement récursives.

1. HOPE est un langage applicatif expérimental qui a comme but, pour l'instant, de tester certaines idées de méthodologie de la programmation plus que d'être un langage de programmation. L'idée était de concevoir un langage qui soit très simple et qui incite à la clarté et à la manipulation de programmes. On cherche à réunir la puissance de LISP avec une discipline de modularité et un typage fort.
Ses principales caractéristiques sont :

a) Pas d'affectation : le fait de travailler en terme d'expressions et de leurs valeurs, et d'utiliser la récursion à la place de boucles, est plus clair et moins susceptible d'erreurs.
b) Utilisation maximale de types définis par l'utilisateur : il doit pouvoir définir ses propres types et la machine doit en

vérifier l'usage. Le langage permet des variables de type de façon à ce qu'on puisse construire des "listes de alphas" à la place de "listes de nombres", où alpha est une variable de type qui peut être instanciée avec un type quelconque. C'est donc, la notion de généricité.

c) Analyse exhaustive par cas : chaque type de données aura un ensemble de sous types disjoints, chacun avec un opérateur constructeur différent ; par exemple, les listes sont construites à partir de cons et de nil. Cela facilite l'analyse par cas en ce qui concerne ces constructeurs et facilite aussi la vérification du compilateur, si l'analyse est exhaustive. De cette façon, on évitera les erreurs du genre "oublier la vérification de la liste vide".

d) Eviter des noms en trop : on doit éviter l'introduction de noms différents pour les fonctions constructrices (par ex. cons), pour tester quel est le sous type (par ex. le prédicat null), ou pour sélectionner des composantes (par ex. car et cdr). Il sera suffisant d'utiliser les constructeurs et une syntaxe d'unification.

e) Eviter dans la mesure du possible la récursion explicite : l'utilisation indisciplinée de la récursivité n'est pas désirable. Il est mieux de se servir des constructions standards qui ont rapport avec les structures de données. Par exemple, $\{x^2 | x \in S\}$ à la place de la récursion sur les éléments de S.

Si le langage permet des fonctions d'ordre supérieur, c'est à dire, qui acceptent des fonctions comme paramètres, on peut facilement décrire des fonctions d'ordre supérieur avec une récursion implicite ; par exemple, mapcar en LISP.

f) Les séquences comme objets : dans la programmation impérative les séquences sont parfois représentées comme des valeurs successives d'une variable ou comme un vecteur ou une liste. Mais l'inconvénient de ces deux dernières représentations est que tous les éléments de la séquence doivent être en mémoire en même temps. Une idée, l'évaluation paresseuse

<Hen-Mor 76>, nous permet de traiter les séquences comme des listes, mais en évaluant un élément chaque fois qu'il est nécessaire. Cette technique permet de travailler avec des structures infinies.

g) Modularité et abstraction. La clé pour faire des grands programmes en réduisant la probabilité d'erreurs est de les construire par petits morceaux qui communiquent entre eux qui d'une façon explicite et décrite formellement. En ce qui concerne la description de données, on cherche à définir types de données par leurs opérateurs et non par la représentation qu'ils peuvent avoir. C'est ce qu'on appelle les "types abstraits".

2. KRC (Kent Recursive Calculator) est un système de programmation applicatif qui a été implémenté à l'Université de Kent avec un propos d'enseignement. Il est fondé sur un langage antérieur, SASL <Tur 76>, mais on lui a ajouté une facilité, l'abstraction d'ensembles. KRC a des ressemblances avec HOPE en ce qui concerne :

- l'absence d'affectations et la transparence,
- la non-séquentialité,
- la syntaxe d'unification,
- l'abstraction d'ensembles, fonctions d'ordre supérieur,
- "évaluation paresseuse" pour le traitement de structures qui peuvent être infinies,
- modularité,
- .....

La différence fondamentale est que HOPE introduit la possibilité de définir des types abstraits, tandis que KRC travaille avec deux types de base : nombres et chaînes ; et deux types d'objets structurés : listes et fonctions.

Quant à l'analyse par cas, KRC se sert des commandes gardées (HOPE a des formes d'expression conditionnelles).

Voyons quelques exemples programmés en KRC qui montrent ses caractéristiques les plus intéressantes :

1) Faire l'addition de tous les éléments d'une liste :

```
sum[ ]=0
sum(a:x)=a+sum x
```

syntaxe d'unification

2) Donner la liste des dix premiers carrés parfaits:

```
{n*n; n<-[1,...,10]}
```

Utilisation de l'abstraction d'ensembles. Le { est lu "liste de tous", le point virgule "tel que" et le <- "pris de". La structure [1,...,10] sera traitée par "évaluation paresseuse" et interprétée comme un intervalle avec ses limites supérieure et inférieure. Nous pouvons demander le même calcul pour tous les nombres Naturels :

```
{n*n; n<-[1,...]}
```

intervalle infini

et obtenir comme résultat :

```
[1,4,9,16,...etc
```

3) Fonction puissance :

```
puiss 0 x = 1
puiss n x = x*puiss(n-1)x
```

Maintenant on peut écrire :

```
sq   = puiss 2
cube= puiss 3
```

La fonction puiss est utilisée ici comme fonction d'ordre supérieur. KRC offre la paramétrisation partielle, c'est à dire, que les fonctions de plus d'un argument peuvent toujours être traitées comme fonctions d'ordre supérieur ce qui amène à avoir un style de programmation extrêmement condensé. C'est peut être cela la plus intéressante caractéristique des programmes applicatifs.

1.3. Conclusion

Un objectif très important pour la conception de langages

est de vérifier le fait qu'un programme réalise une spécification donnée. Dans ce sens, les langages applicatifs offrent des avantages considérables. L'absence d'instructions d'affectation et le remplacement de l'itération par la récursion donne aux programmes une forme simple et facile pour l'analyse. Des systèmes de vérification pour les langages applicatifs ont été développés par <Boy-Moo 77> et par Aubin <Aub 77>. Dans <Tur 81c> est présentée une proposition pour la vérification des langages fonctionnels fondée sur l'utilisation des propriétés de l'égalité (complètement substitutive en raison d'une sémantique non stricte), et sur l'induction structurelle. Les avantages de cette méthode par rapport à celle des "assertions inductives" pour les programmes impératifs sont :

1. Il y a une théorie de "correction" totale.
2. Les programmes et les preuves se font essentiellement dans le même langage : équations d'expressions applicatives.
3. Le degré de formalisme est plus élévé, ce qui rend ce mécanisme proche de ce qui est nécessaire pour faire une vérification automatique.

Une preuve formelle complète pour la vérification d'un petit compilateur, en utilisant la méthode d'induction structurée, se trouve en <Tur 81d>.

Un autre avantage des langages applicatifs est que les programmes se prêtent eux mêmes très bien à la technique de transformation de programmes <Bur-Dar 77> : un programme simple mais inefficace peut être transformé systématiquement en un programme acceptable et efficace pour des opérations élémentaires qui préservent sa "correctitude".

Les langages applicatifs n'ont pas la notion de séquentialité des langages impératifs. La valeur de la fonction e0(e1,..,en) est toujours indépendante de l'ordre d'évaluation des expressions eo,..,en ; cela est garanti par l'absence de l'affectation. Si une machine parallèle est disponible,

e0,..,en seront évaluées simultanément et si e0,..,en sont des applications de fonctions, cette décomposition parallèle se répetera.

Les programmes écrits dans un langage applicatif, comme KRC, sont beaucoup plus courts que les mêmes programmes écrits en Pascal par exemple.

La compréhension d'un programme écrit dans un langage applicatif est très simplifiée grâce à l'absence complète d'effets de bord.

Ces techniques applicatives servent aussi dans des cas où l'on a de grands programmes. La clé a été le développement de l'évaluation paresseuse qui rend possible le traitement des problèmes d'entrée-sortie et d'intercommunication des processus dans un cadre purement applicatif. Il est possible d'écrire un système d'exploitation complet dans un langage purement fonctionnel <Dar-Her 81>.

La possibilité d'avoir à la fois des équations récursives et l'abstraction d'ensembles produit une combinaison qui, comme le dit <Tur 81a>, permet "to think very big thoughts".

2. PROGRAMMATION PARALLELE

2.1. L'introduction de "la communication" dans les langages de programmation: motivation.

Vers les années 70 certains concepts de base des langages de programmation étaient courants et bien compris. Par exemple l'affectation : un changement de l'état interne d'une machine exécutant un programme peut être réalisé par l'affectation d'une nouvelle valeur à une partie variable de cette machine. Par

contre, les opérations d'entrée et de sortie qui affectent l'environnement externe d'une machine sont loin d'être aussi bien intégrées dans les langages. Elles sont en général ajoutées aux langages de programmation avec une réflexion a posteriori.

Parmi les méthodes de programmation structurée, trois constructeurs ont reçu une grande acceptation et utilisation : le constructeur répétitif (par ex. tantque), le constructeur alternatif (par ex. si alors sinon) et le constructeur de composition séquentielle (par ex. ;). Mais il n'y a pas eu d'accord en ce qui concerne d'autres structures importantes de programmes et beaucoup de suggestions ont été faites : "subroutines" en FORTRAN, "procedures" en ALGOL60, "entries" en PL/1, "coroutines" en C, "classes" en SIMULA, "clusters" en CLU, "forms" en ALPHARD, "processus et moniteurs" en PASCAL concurrent, ...

En même temps, le souhait d'atteindre de plus grandes vitesses conduit à penser au parallélisme. La technologie des processeurs suggère par exemple, un modèle de machine construite à partir d'un nombre de processeurs tous semblables, chacun avec sa propre mémoire. Cette machine supporterait le parallélisme d'une façon plus puissante et moins chère. Mais pour pouvoir utiliser une telle machine efficacement pour l'accomplissement d'une seule tâche, les processeurs composants doivent être capables de communiquer entre eux, et de se synchroniser.

Plusieurs méthodes ont été développées à propos de la communication. Une qui a été largement adoptée est la méthode par inspection et actualisation d'une mémoire commune (Algol68, PL/1,...), mais elle peut entrainer de grandes difficultés pour la construction de programmes corrects.

En ce qui concerne la synchronisation, divers méthodes ont été aussi proposées : les "sémaphores" de Dijkstra, les "events"

en PL/1, les "régions critiques" conditionnelles de Hoare, les "moniteurs et queues" de Pascal Concurrent,... la plupart d'entre elles sont très convenables pour leur propos, mais il n'y a pas de critère généralement reconnu pour faire un choix.

Enfin, la nécessité d'introduire la "communication" dans les langages de programmation d'une façon propre et complète, paraît évidente. Cette façon doit contenir essentiellement:
- la description de l'entité communicant avec les entrées et les sorties,
- un mécanisme de synchronisation,
- la définition d'un opérateur de mise en relation des entités communicantes.

2.2. Quelques langages de programmation parallèle.

2.2.1. Communicating Sequential Processes (CSP).

En 1978 C.A.R. Hoare <Hoa 78> propose un langage de programmation concurrent, CSP, qui intègre à un langage impératif traditionnel la notion de communication entre entités, le parallélisme, le non determinisme et la synchronisation par rendez vous. Les entrées, sorties, la concurrence sont regardées comme primitives simples d'un langage, ce qui garantit que cette inclusion est logiquement compatible avec le reste du langage.

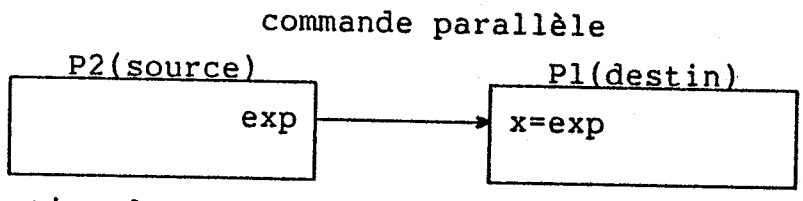
Parmi ses caractéristiques, les plus importantes sont :

- L'utilisation des commandes gardées, assez proches de celles de Dijkstra, pour introduire le non déterminisme controlé et le contrôle séquentiel de structures. Il définit une commande alternative comme une suite de commandes gardées où seulement l'une d'elles sera exécutée. Si toutes échouent, alors la

commande alternative échoue, mais si par contre plus d'une réussit, un choix arbitraire parmi celles ci sera fait pour sélectionner et exécuter.

- Les entités communicantes, processus, sont des programmes faits à partir de commandes séquentielles. Une commande parallèle permet de spécifier l'exécution concurrente de tous les processus concernés. Tous les processus commencent simultanément et la commande parallèle se termine quand tous les processus sont terminés. Les variables globales à plusieurs processus ne sont par permises.

- Deux autres commandes sont introduites :
  commande d'entrée : P?x où P est le processus source et x est la variable recevant la valeur lue de P.
  commande de sortie : P!y où P est le processus qui reçoit et y l'expression qui va être lue par P.
  Ces deux commandes sont utilisées pour établir la communication entre processus concurrents.
  La communication se fait par rendez vous : quand un processus se réfère à un autre comme destination d'une sortie, et en meme temps cet autre se réfère au premier comme source, la communication a lieu et la valeur de la sortie est affectée à la variable d'entrée.
  Exemple: P1::P2?x ‖ P2::P1!exp



  L'exécution de cette commande sera accomplie quand P2 sera prêt à envoyer et P1 prêt à recevoir ; l'attente possible est invisible par l'environnement et c'est pour cela qu'on dit que la communication est simultanée.

- Les commandes d'entrée peuvent apparaître dans les gardes des

commandes gardées. Une commande gardée avec une entrée comme garde sera choisie seulement si le processus source est prêt à exécuter la commande de sortie correspondante. Comme on l'a dit auparavant, s'il y a plusieurs processus prets, un choix non déterministe sera fait.

- Une syntaxe d'unification sera utilisée pour discriminer la structure d'un message d'entrée et pour avoir accès à ses composantes d'une façon sûre. On interdira l'entrée de messages qui ne s'unifient pas avec le modèle spécifié.

Les programmes en CSP peuvent être implémentés soit dans une machine conventionnelle (une seule mémoire principale), soit dans une machine composée d'un nombre fixe de processeurs connectés par des canaux d'entrée et de sortie. En conséquence, c'est un langage assez statique : le texte d'un programme spécifie le nombre maximum des processus se comportant en parallèle et il n'y a pas de récursion dans la construction des processus.

L'importance de CSP est très grande : la définition d'un processus séquentiel communicant fournit une méthode pour exprimer des solutions aux problèmes couramment utilisés pour illustrer la puissance d'un langage. Nous pouvons conclure qu'un processus CSP constitue une bonne synthèse d'un nombre d'idées de programmation conventionnelles et nouvelles.

Par contre en <Hoa 78> deux inconvénients sont exposés :
- Aucune proposition de méthodes de preuves de développement et vérification de programmes n'est faite.
- Aucune attention n'a été prêtée aux problèmes d'efficacité d'implémentation, ce qui peut être grave quand il s'agit d'un ordinateur séquentiel traditionnel.

Certaines solutions possibles sont envisagées dans

<Hoa 78>, mais tel qu'il est présenté, CSP ne doit pas être regardé comme un langage de programmation ni abstraite ni concrète ; il est simplement la présentation d'une bonne solution aux problèmes ennocés dans la motivation (cf. 2.1).

Peu de temps après, un modèle et une théorie pour les processus séquentiels communicants ont été présentés <Hoa 80>, <Hoa 81a>, ainsi qu'un calcul de "correction" totale pour les processus communicants <Hoa 81b>.

OCCAM

D'après CSP et le modèle mathématique proposé comme base d'une méthode de preuve de correction, on trouve une proposition d'un langage de programmation parallèle en <May 83>. Ce langage est OCCAM, un nouveau langage parallèle issu de CSP qui offre un formalisme de construction de logiciel pour l'implémentation des systèmes distribués et concurrents. Ses caractéristiques sont :

- Un processus est une entité de base du langage. Il peut être soit une instruction unique (affectation (:=), entrée (C?x), sortie (C!exp) et attente (wait)), soit le résultat des opérateurs constructeurs (op. séquentiel (SEQ), op. répétitif (WHILE), op. parallèle (PAR), op. alternatif (ALT), op. conditionnel (IF), op. multiplexeur (FOR)).

- Un canal, est une autre entité de base du langage qui peut être considérée comme un composant élémentaire de synchronisation entre deux processus, un d'entrée et l'autre de sortie.

- La communication est définie comme un couple entrée sortie synchrone sur le même canal (mecanisme de rendez vous).

- OCCAM peut exprimer à l'aide d'un constructeur de base, la structure hiérarchique et modulaire d'un système par encapsulation d'un ensemble de processus communicants permettant à l'environnement de le voir comme un seul processus.

- En OCCAM les constructeurs du langage permettent l'exécution parallèle ou séquentielle d'un ou plusieurs processus, ainsi qu'une sélection non déterministe d'un parmi un ensemble de processus, en fonction de conditions logiques.

- En OCCAM un ensemble de règles de transformation peuvent être systématiquement utilisées pour la construction correcte et l'optimisation des programmes. Cela est dû au fait que la sémantique formelle de CSP permet la lecture d'un programme OCCAM comme un prédicat formé d'assertions dans une extension du calcul des prédicats où les règles de déduction peuvent être appliquées.

On peut conclure d'OCCAM que c'est une version améliorée et implémentable de CSP. OCCAM est un vrai langage de programmation parallèle. Un bon exposé d'OCCAM qui suggère aussi une façon adaptée de considérer les machines ("transinateurs") se trouve en <Mun 83>.

2.2.2. Introduire la communication ne suffit pas: Systèmes Parallèles Communicants.

Un langage qui introduit la notion de communication, parmi les multiples et importants moyens qu'il met en oeuvre pour répondre à une grande variété d'applications, est le langage ADA <ADA 82>. Dans ce langage, l'entité communicante est la tâche.

- Le nombre des tâches d'une application n'est pas fixé

statiquement.

- La synchronisation est assurée par un mécanisme de rendez vous voisin de celui de CSP.

- La communication est réalisée par la transmission de paramètres typés, dans les deux sens, à l'occasion des rendez vous.

- La désignation des tâches dans les rendez vous est asymétrique: une tâche T1 doit désigner la tâche T2 avec laquelle elle demande un rendez vous, alors que la tâche T2 accepte des rendez vous avec toute tâche susceptible de la désigner (à la différence de CSP).

- ADA offre des constructions dérivées des commandes gardées qui traduisent le non déterminisme des événements. Ces constructions sont cependant réservées à l'expression des communications entre tâches (instruction select), le langage offrant par ailleurs les instructions de contrôle classiques (for séquentiel, case déterministe, etc.).

Le langage ADA permet d'exprimer facilement les problèmes de synchronisation et de communication que l'on rencontre habituellement en programmation système. Mais, le langage ADA ne permet pas de tout faire. Il s'agit surtout d'un langage de programmation système, et non de calcul parallèle. Aucun mécanisme ne permet, en particulier, d'exprimer un parallélisme synchrone fin.

On peut en tirer comme conséquence que l'introduction de la communication dans les langages va au delà du fait d'ajouter les notions inhérentes: entrées, sorties, processus, non déterminisme, communication, etc aux langages de programmation.

Un système parallèle communicant, en plus d'un langage qui implémente toutes ces notions, doit définir un calcul des systèmes concurrents. Un tel calcul doit avoir un très haut niveau d'articulation et cette faculté d'articulation implique non seulement une richesse d'expression mais aussi une grande souplesse de manipulation. Un calcul doit nous permettre de décrire les systèmes existants, de spécifier et programmer des nouveaux systèmes, de raisonner mathématiquement sur les systèmes et tout cela sans sortir du contexte notationnel du calcul. Tous ces critères réunis sont difficiles à obtenir déjà pour les systèmes concurrents dans le domaine informatique et beaucoup plus, peut être, pour les systèmes en général.

Des essais nombreux et importants ont été faits:

- Théorie de Réseaux (Net Theorie) <Pet 80>, <Gen 80> sur laquelle sont fondées un grand nombre de techniques d'analyse.
- Des modèles fondés sur la communication non synchronisée: Hewitt's Actors Systems <Hal 79>, Coroutines an Networks of Parallel Processes <Kah 77>.
- Un style différent pour présenter les systèmes concurrents se trouve aussi dans les Expressions de Chemins <Cam 74>. A travers ce modèle une analyse mathématique peut être faite.
- L'élégant travail de "description en negatif" <Mag 79> où on décrit le comportement à partir des états qui ne peuvent jamais être atteints.

2.2.3. A Calculus of Communicating Systems (CCS).

En 1980, Milner présente un calcul pour les systèmes concurrents. Ce calcul présente un grand intérêt, justement parce qu'il réunit en grande partie toutes les caractéristiques souhaitables d'un calcul telles qu'on les a ennoncées dans le paragraphe précedent. Les mêmes notations sont utilisées pour définir et raisonner sur les systèmes. Le calcul est applicable

non seulement aux programmes mais aussi aux structures de données et, à un certain niveau d'abstraction, aux architectures materielles. En plus d'un bon niveau d'articulation, il existe toute une théorie fondée sur une petite collection d'idées de base bien reliées entre elles au moyen de laquelle on justifie les manipulations du calcul.

Le calcul de Milner est fondé sur deux idées centrales:

1. L'observation : un système concurrent doit être décrit de façon à ce qu'il soit possible de déterminer exactement quel comportement sera vu ou experimenté par un observateur externe. Le calcul définit une relation d'équivalence observationnelle et recherche ses propriétés.

2. La communication synchronisée : chaque système concurrent est construit à partir d'entités indépendantes qui communiquent. Une communication entre deux entités indépendantes est une action indivisible ; la composition concurrente est l'opérateur central de l'algèbre des systèmes et sa fonction est de composer deux entités indépendantes en permettant la communication. Cette notion est aussi importante en CCS que la notion de séquence dans la programmation séquentielle. En CCS un programme ou description de système est juste un terme du calcul et, en conséquence, la structure du système est reflétée dans la structure du terme. Les manipulations consistent en général, à transformer un terme en derivant un autre terme de structure différente mais du même comportement.

Ces deux idées centrales sont réellement une seule : la seule façon d'observer un système est de communiquer avec lui, ce qui fait du système et de l'observateur un plus grand système.

Les termes de CCS représentent les comportements de

systèmes et ils sont soumis à des lois équationnelles. Cela donne une algèbre, et c'est un tel cadre algébrique qui paraît convenir le mieux à une composition sémantique.

Une grande variété de systèmes peuvent être exprimés et discutés en CCS : controleurs, calculs "data flow", algorithmes numériques concurrents, dispositifs de mémoires, structures de données, descriptions sémantiques de langages de programmation parallèle. On trouve dans <Mil 79>, la modélisation et vérification d'un lecteur de carte utilisant une première version des idées présentées en CCS.

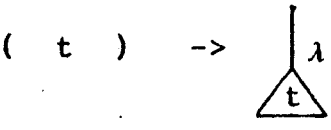
En CCS le passage de valeurs d'une entité à une autre est permis ce qui rend le calcul d'une grande puissance expressive.

CCS développe une algèbre sur les comportements. Une équivalence observationnelle des lois algébriques est définie et sert de base à toute discussion sur les comportements. Ces lois ont été prouvées "complètes" pour un cas simplifié de comportements finis. (Dans ce cas, "complète" veut dire que si deux expressions de comportement sont équivalentes observationnellement dans tous les contextes, l'égalité entre ces comportements sera prouvée par les lois).

Dans CCS le comportement d'une entité est représenté par un arbre (arbre de synchronisation, ST). Les principaux opérateurs de l'algèbre sont (exposition intuitive) :
NIL (opérateur nul) représenté par un point : .

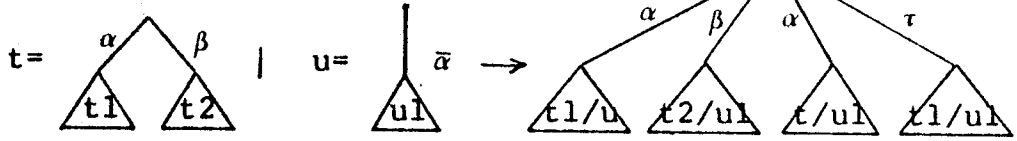
+ :

λ :

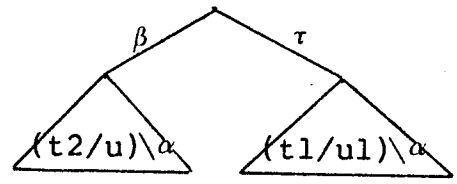


où est un "nom d'événement"

| :



$\backslash$: $(t/u)\backslash\alpha$ $\longrightarrow$



En 1982, Milner présente un nouveau travail dans la même ligne : "Calculi for Synchrony and Asynchrony" <Mil 82>. Ce calcul qu'il propose, modélise le calcul synchrone et assynchrone. Il montre que le calcul présenté en <Mil 80> peut être dérivé à partir de son nouveau calcul.

2.3. Conclusion

Définir des outils pour la programmation parallèle consiste à introduire dans les langages de programmation des moyens pour pouvoir bien exprimer toutes les notions inhérentes à la communication d'entités indépendantes: entrées, sorties, un opérateur de mise en parallèle, un mecanismme de synchronisation, un mecanisme de communications, le non déterminisme. Parmi les travaux réalisés dans ce sens, qui sont très nombreux, on peut considérer que les apports essentiels ont été faits par deux d'entre eux :

1. CSP, dont on peut dire qu'il a été le premier à regrouper et à résoudre d'une façon simple dans les langages les notions et problèmes posés par la communication. Cela a été fait dans un langage impératif dont la sémantique formelle est décrite d'une façon opérationnelle.
Une autre qùalité à lui reconnaitre est le fait d'avoir rassemblé de façon cohérente les éléments essentiels du problème :
- la synchronisation par rendez vous
- l'implémentation du non déterminisme
- la définition de l'opérateur parallèle.

2. CCS, dans le calcul de Milner l'objectif est plus ambitieux et la démarche différente. Il propose un système mathématique pour étudier les systèmes communicants et son apport essentiel est de définir une algèbre des processus. CCS travaille dans un style applicatif (pas d'affectations seulement des expressions). Par contre, le traitement de la communication est semblable à celui de CSP.

3. PROGRAMMATION PARALLELE ET PROGRAMMATION FONCTIONNELLE: est ce possible?

La reponse à cette question, dans un premier temps est non. Dans la programmation applicative, la séquentialité explicite est exclue, il n'y a pas d'affectations ni de notion d'états ; on arrive à s'exprimer en termes de fonctions et d'une algèbre sur ces fonctions. Dans la programmation parallèle, dont le but est d'exprimer la communication, on se voit obligé à dire des choses comme :

si P reçoit x par I, puis y par J, alors P envoie x+y par O et si en même temps, Pl reçoit z par K alors P et Pl communiquent, puis Pl envoie z par Q, etc..., où :

P,Pl sont des processus
I,J,K sont "portes d'entrée"
O,Q sont "portes de sortie"

Dans cette phrase, un rangement d'événements dans le temps est explicite : "avant", "après", "en même temps". La notion de séquentialité est nécessaire pour exprimer des comportements, ce qui implique la notion d'état. C'est la raison pour laquelle la première reponse est non : si l'on veut conserver un style applicatif pur, l'absence de la notion d'état empêche d'exprimer ce qui est nécessaire pour la programmation parallèle.

Cependant, on peut imaginer un style de programmation

mixte, très proche de la programmation fonctionnelle, où la violation soit legère et proprement réalisée. Cela consiste à introduire un état qui serait actualisé une fois par événement (ensemble de communications possibles). Cet état, quelle que soit sa structure interne, serait actualisé applicativement comme un tout : tous les calculs internes se feraient en parallèle, il n'y aura pas d'affectations (syntaxe d'unification) et la portée de variables serait locale à chaque actualisation, pas d'effets de bord.

Ainsi, un programme dans notre proposition devient un ensemble non vide des règles de transition Ti de la forme :

Ti : (Ci,eveni) -> Fi, où :

Ci est une condition sur l'état courant,

eveni est événement et

Fi est une fonction des états dans les états.

L'initialisation du programme est de la forme :

Tinit : (vrai, $\mathcal{E}$) -> Finit, où Finit est une constante

Une transition Ti peut être appliquée si, dans l'état E, Ci(E) est vrai et si toutes les communications de eveni sont prêtes à avoir lieu. Si Ti est appliquée, le nouvel état courant est Fi(E). Or, on peut organiser la définition des fonctions d'une façon tout à fait analogue, comme c'est le cas dans les langages applicatifs, avec définition équationnelle de fonctions. Par exemple, deux règles peuvent décrire la fonction Fact(x) :

```
Fact(x) : x=0 -> 1
Fact(x) : x>0 -> x*Fact(x-1)
```

On peut donc constater une grande analogie d'expression entre, d'une part, la définition du système de transition nécessaire au parallélisme et à la communication et, d'autre part, la définition de fonctions.

Dans cette programmation mixte, tout ce qui est calcul, ainsi que le contrôle, sont exprimés sous une forme analogue.

Dans <Bac 78> on trouve la présentation d'une classe des systèmes dans lesquels la programmation applicative est associée à une certaine sensitivité historique (états), sans qu'il s'agisse de systèmes de Von Neumann. Ces systèmes s'appelent Systèmes de Transition d'Etats Applicatifs ("AST Systems"), et leur structure est définie à partir de trois éléments :

- Un sous système applicatif.
- Un état (ensemble de définitions de sous système applicatif).
- Un ensemble de règles de transition qui décrivent comment l'état peut évoluer.

La ressemblance avec notre proposition est grande, ce qui nous permet de dire que toutes les justifications, comparaisons avantageusses et remarques finales qui se trouvent en <Bac 78> pour caractériser un Système de Transition Applicatif sont valables pour notre modèle.

Nous allons donc conclure ce paragraphe en donnant un oui comme reponse à la question posée dans le titre, reponse beaucoup plus réflechie que celle qui venait d'abord à l'esprit. Le langage FP2 présenté dans le chapitre III de cette thèse en apporte la confirmation car il s'agit d'un langage de programmation parallèle et fonctionnelle.

CHAPITRE II

DEUX SYSTEMES SOURCE: SPARC ET LPG

CHAPITRE II

DEUX SYSTEMES SOURCE: SPARC ET LPG

Ce chapitre est consacré à l'exposé de deux systèmes.

- Le premier, SPARC (Systèmes PARallèles Communicants), est un projet développé au sein du LIFIA. Il a comme objectif d'intégrer dans un cadre cohérent les spécifications de fonctions, de types et de processus communicants et de fournir au concepteur de systèmes les outils correspondants d'analyse et de validation.

- Le deuxième, LPG (Langage de Programmation Générique) développé aussi au LIFIA, est un langage applicatif permettant la spécification et la programmation de types abstraits et des fonctions génériques.

SPARC est le projet sur lequel mon travail de recherche a été concentré dès le début. Le travail, que j'expose dans cette thèse a abouti à la création d'un langage pour les processus communicants, FP2, dont la sémantique est décrite au moyen d'un formalisme algébrique.

D'autre part, nous nous sommes servis du langage LPG, où sont définis les concepts de types abstraits et fonctions. Comme cela constitue aussi un de nos propos, nous avons pris LPG pour l'intégrer à FP2. Cette opération a pu se réaliser d'une façon naturelle car les deux systèmes ont des cadres formels semblables : technique de spécification algébrique.

1. SPECIFICATION ALGEBRIQUE ET QUELQUES DEFINITIONS

Le mot spécification est employé pour définir une espèce ou une chose, pour déterminer ses caractères <Le Petit Robert>. Ce terme souvent utilisé dans la vie courante, apparaît comme la façon de communiquer une description souhaitée. Pour ce faire, le demandeur se sert de conventions, selon le domaine concerné, qui ont un caractère suffisamment objectif pour que la description soit bien comprise par le réalisateur.

En informatique on parle aussi de spécification. En effet, la spécification est le pas intermédiaire entre le concept (ce que l'on veut exprimer, calculer, etc) et le programme. Les spécifications peuvent être informelles comme par exemple l'exposition du problème en langue naturelle ou bien formelles ; c'est à dire, basées sur des fondements logico mathématiques. Entre les deux, il existe une gamme de façons de spécifier. Cependant, la tendance actuelle est de spécifier formellement, et les raisons pour le faire sont fortes. D'abord, le fait de décrire mathématiquement un concept, évite déjà en bonne partie les mauvaises interprétations dues à l'ambiguité, la rhétorique, la non clarté du problème, etc. D'autre part, le formalisme mathématique peut être un outil important pour la vérification de programmes. La spécification nous ayant fourni une description mathématique du concept, la vérification d'un programme consiste à prouver une relation d'implémentation correcte entre le programme et la spécification (P<=S).

Enfin, une spécification formelle nous propose un cadre pour raisonner sur les programmes, afin, par exemple, de déterminer l'équivalence de deux programmes ou de trouver les modifications d'un programme correspondants aux changements d'un concept.

Sur la vérification, équivalence, terminaison, etc., voir

par exemple <Gog 74>.

Ainsi, les caractéristiques souhaitables d'une spécification sont :

1. Formalisme, qui permet de faire de raisonnements mathématiques et éventuellement un traitement automatique.
2. Simplicité de Construction, pour le demandeur.
3. Simplicité de Comprehension, pour le réalisateur.
4. Minimalité, qui permet une description précise et sans ambiguité.
5. Extensibilité, qui permet des modifications sur la spécification du même ordre de complexité que celles du concept.

Actuellement, les informaticiens sont à la recherche de "langages" qui soient aussi des outils de spécification de programmes, qui soient à la fois précis, concis, lisibles et qui de plus facilitent la programmation, la preuve et la maintenance. Dans ce domaine, on distingue plusieurs techniques de spécification formelle que l'on peut regrouper en cinq catégories <Lis-Zil 77> :

1. Utilisation d'un domaine fixe d'objets formels : ensembles, graphes, etc.
2. Utilisation d'un domaine formel approprié mais arbitraire.
3. Utilisation d'un modèle de machine à états.
4. Utilisation d'une définition implicite en termes d'axiomes.
5. Utilisation d'une définition implicite en termes de relations algébriques.

Nous nous intéressons à cette dernière technique : la spécification algébrique. Les deux langages de spécification qui seront exposés dans ce chapitre sont décrits à travers un formalisme algébrique.

Pour commencer nous présentons quelques définitions dont

l'ensemble constitue le cadre algébrique qui sert de base aux langages SPARC et LPG. La plupart de ces définitions ont été prises dans <Gog-Bur 81> et <Bur-Gog 81>. Voir aussi <Bur 81>.

Signature (Σ)

La notion de signature correspond, en programmation à un paquet de déclarations de types, de constantes et de procédures. On utilise les mots "sorte" au lieu de "nom de type" et "opérateur" au lieu de "nom de fonction".

Une signature décrit une famille d'ensembles d'opérateurs indexés par leur arité, c'est à dire, avec leur séquence de "sortes d'entrée" et leur "sorte de sortie".

Définition. Une signature, Σ, est une paire $<S,\Omega>$ où S est un ensemble de sortes et Ω est une famille d'ensembles d'opérateurs indexée par S^*xS.

Σ-Algèbre

Les signatures représentent une syntaxe sans sémantique. Si l'on donne un sens à cette syntaxe on obtient une algèbre. Une algèbre correspond à une signature par une fonction qui associe des ensembles aux sortes et des fonctions aux opérateurs. On appelle "porteur" de l'algèbre A, $|A|$, la famille d'ensembles associée aux sortes.

Définition. Soit Σ une signature. Une Σ-algèbre, $<A,\alpha>$, est une famille d'ensembles S-indexée, appelée le porteur de A ($|A|$) avec une famille (S^*xS)-indexée d'applications

$$\alpha_{US} : \Omega_{US} \rightarrow (A_U \rightarrow A_S)$$

où $u \in S^*$, $s \in S$ et $A_{u_1,\ldots,A_{u_n}} = A_{u_1} \times \ldots \times A_{u_n}$.

Notation. Si $w \in \Omega$ et $\langle a_1,\ldots,a_n \rangle \in A$, on écrira $w(a_1,\ldots,a_n)$ à la place de $\alpha_{us}(w)(a_1,\ldots,a_n)$ sans danger d'ambiguité.

Homomorphisme

Définition. Si $\Sigma=\langle S,\Omega \rangle$ alors un Σ-homomorphisme d'une Σ-algèbre $\langle A,\alpha \rangle$ vers une Σ-algèbre $\langle A',\alpha' \rangle$ est une application $f:A \to A'$ tel que pour chaque $w \in \Sigma$ et chaque $a_1 \in A_{s1}, \ldots, a_n \in A_{sn}$ on a :

$f_s(w(a_1,\ldots,a_n)) = w(f_{s1}(a_1),\ldots,f_{sn}(a_n))$

Algèbre de Termes, Algèbre Initiale et Algèbre Libre

Pour toute signature il existe une algèbre particulièrement intéressante appelée l'algèbre de termes (word algebra). Ses éléments sont toutes les expressions construites avec les éléments de la signature. On peut voir ces expressions comme chaines de caractères et les opérateurs comme manipulateurs de chaines. Mais, nous pouvons définir cette algèbre d'une façon plus abstraite ; l'idée clé est que, pour n'importe quelle interprétation des opérateurs on obtient une valeur unique pour chaque expression. Mais une interprétation des opérateurs est une autre algèbre A et l'évaluation des expressions est une homomorphisme à partir de l'algèbre de termes. De plus, c'est le seul homomorphisme vers A.

Définition. I est une Σ-algèbre initiale ssi pour toute Σ-algèbre A, il existe un homomorphisme unique $f:I \to A$

Fait. Si I et I' sont Σ-algèbres initiales, alors elles sont

isomorphes.

Fait. L'algèbre de termes est initiale.

Un concept un peu plus général est l'algèbre de termes définie sur une signature Σ et sur un ensemble X de variables. L'ensemble d'éléments de cette algèbre est l'ensemble d'expressions utilisant ces variables. On note cette algèbre : $W_\Sigma(X)$ et on l'appelle algèbre libre sur X.

Définition. Une algèbre libre sur X est une Σ-algèbre F(X) avec une fonction η_X : X-> F(X) telle que pour chaque Σ-algèbre A, n'importe quelle fonction f: X->|A| peut s'étendre de façon unique à un homomorphisme $f^\#$: F(X)->A. $f^\#$ étends f veut dire que $|f^\#| \circ \eta_X = f$. L'algèbre de termes sur X est une algèbre libre sur X. Cette algèbre de termes correspond à ce que l'on appelle l'univers d'Herbrand.

Equation

Définition. Une Σ-équation e est un triplet <X,t1,t2> où X est un ensemble S-indexé de variables et t1 et t2 $\in |W_\Sigma(X)|$ sont des termes sur X de la même sorte.
Les variables sont quantifiées universellement, c'est à dire, qu'une équation se lit : $\forall$X, t1=t2.

Satisfaire une équation

Définition. Une Σ-algèbre A satisfait une Σ-équation <X,t1,t2> ssi pour toutes les applications f : X->|A|, $f^\#(t1)=f^\#(t2)$. On écrira A$\models$e pour "A satisfait e".

Définition. Soit E un ensemble d'équations. Une algèbre est un

modèle de E si elle satisfait chacune des équations de E. On écrira E^* pour l'ensemble de tous les modèles de E.

Fermeture d'un ensemble d'équations

Soit E^* l'ensemble de tous les modèles de E. Soit M un ensemble de Σ-algèbres et M^* l'ensemble de toutes les Σ-équations qui sont satisfaites par chaque algèbre de M.

La fermeture d'un ensemble E de Σ-équations est l'ensemble E^{**} que l'on écrira $\bar{E}$. On dit que l'ensemble E est fermé si $E=\bar{E}$.

Présentation

Définition. Une présentation est une paire $<\Sigma,E>$ où Σ est une signature et E est un ensemble d'équations.

Théorie

Définition. Une théorie est une présentation $<\Sigma,E>$ telle que E est fermé ; c'est à dire $<\Sigma,\bar{E}>$.

Algèbre Initiale d'une théorie

Définition. I est une algèbre initiale d'une théorie ssi pour chaque algèbre A de la théorie, il existe un homomorphisme unique f : I->A.

Une façon de trouver l'algèbre initiale d'une théorie est de prendre l'algèbre des termes sur sa signature et de former

les classes d'équivalence correspondants aux équations de la théorie.

Morphisme de Signatures σ

Définition. Un morphisme de signature σ d'une signature $\Sigma 1 = <S1,\Omega 1>$ vers une signature $\Sigma 2 = <S2,\Omega 2>$ est une paire $<f,g>$ où f est une application $f : S1 \to S2$ et g est une famille d'applications

$g_{us} : \Omega 1_{us} \to \Omega 2_{f^*(u)f(s)}$

où $f^* : S1^* \to S2^*$ est l'extension de f aux chaînes.

On écrira $\sigma : \Sigma 1 \to \Sigma 2$.

Fonction $\underline{\sigma}$ associée a σ

Définition. Si $\sigma : \Sigma 1 \to \Sigma 2$ est un morphisme de signature et A est une $\Sigma 2$-algèbre $=<|A|,a>$; alors on définit la fonction

$\underline{\sigma}$: $\Sigma 2$-algèbres $\to$ $\Sigma 1$-algèbres comme

$\underline{\sigma}(A) = A'$ où $|A'|_s = |A|_{\sigma(s)}$

et $w(a_1,\ldots,a_n)$ en A' est $\sigma(w)(a_1,\ldots,a_n)$ en A.

Morphisme de Théorie

Définition. Si T1 et T2 sont des théories, $T1=<\Sigma 1,E1>$, $T2=<\Sigma 1,E2>$, un morphisme de théorie σ: $T1 \to T2$ est un morphisme de signature $\sigma : \Sigma 1 \to \Sigma 2$ tel que pour chaque T2-algèbre A, $\underline{\sigma}(A)$ est une T1-algèbre.

Catégorie. Notation de <Fer 83>

Définition. Une catégorie C est la donnée de deux classes :

- Une classe d'objets notée obj(C)
- Une classe de morphismes notée Morph(C). Un morphisme f est la donnée de deux objets dom(f), codom(f).

Ces deux classes vérifient les axiomes suivants :

- Composition de g∘f est définie ssi dom(g)=codom(f). Dans ce cas dom(g∘f)=dom(f) et codom(g∘f)=codom(g).
- La composition est associative

  (dom(h)=codom(g) et dom(g)=codom(f))
  ==> (f∘g)∘h=f∘(g∘h)
- Identité

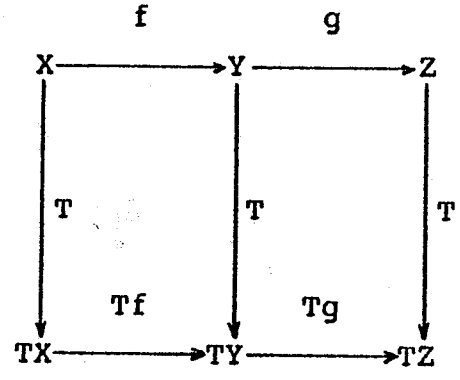
  pour tout X ∈ obj(C), il existe 1-X ∈ Morph(C).
  pour tout f,g codom(f)=X et codom(g)=X ==> g∘1-X=g et 1-X∘f=f.

Foncteur

Définition. Soient C et C' deux catégories. Un foncteur (covariant) T : C->C' est tel que :

1) pour tout A ∈ obj(C), il existe TA ∈ obj(C')
2) T(g∘f) = Tg∘Tf
3) T1-X = 1-TX

Graphiquement :



2. SPARC : SYSTEMES PARALLELES COMMUNICANTS

2.1. Le projet SPARC

2.1.1. Qu'est-ce que SPARC?

SPARC propose un modèle pour spécifier des systèmes dans lesquels les notions de parallélisme et de communication interviennent.

- Le composant de base d'un tel système est le processus communicant. Un processus sera reconnu dans notre système avec les mêmes prerrogatives que des composants plus classiques comme les fonctions ou les types abstraits dans d'autres langages.

- Les processus peuvent évoluer de façon indépendante, ce qui introduit dans le système toutes les possibilités de parallèlisme, alors que la séquentialité était la règle dans le cas des systèmes classiques.

- Les processus peuvent être interconnectés. Les communications, qui transportent des messages de processus à processus, deviennent une autre façon de transmettre de l'information d'un environnement à un autre, en plus des passages de paramètres qui existent déjà dans les appels de fonctions.

L'objectif du projet SPARC est d'intégrer dans un cadre cohérent les spécifications de fonctions, de types et de processus communicants, ainsi que de fournir au concepteur des systèmes les outils correspondants d'analyse et de validation.

2.1.2. SPARC : Etat de l'art.

Les travaux menés dans le cadre de ce projet ont passé nécessairement par une étude à caractère fondamental et théorique du parallélisme et des processus communicants. Formaliser les objets et concepts essentiels des systèmes parallèles communicants, puis élaborer des théories sur les modèles choisis, c'est ce qui a constitué la première partie du travail entrepris dans le projet SPARC. Le modèle de spécification et de construction proposé sera exposé aux paragraphes 2.2, 2.3 et 2.4.

Une deuxième tâche a été de réaliser un poste de travail expérimental pour l'aide à la conception des systèmes repartis composés de réseaux de processus qui échangent des messages. Comme la façon la plus "naturelle" pour décrire des réseaux de processus communicants est de dessiner des assemblages de "boîtes" et de "flêches", où les boîtes représentent les processus et les flêches leurs interconnections, SPARC offre un environnement et un langage graphique pour implementer le poste experimental <Ark 84>.

Une troisième catégorie de travaux est en train de s'effectuer : des études sur l'analyse des systèmes. Les principes de base choisis pour la description de processus nous permettent d'avoir des interprétations sémantiques différentes pour analyser les proprietés comportamentales et fonctionnelles du système. Trois sémantiques diffèrents sont proposées en <Per 84>. Le fait de pouvoir utiliser des outils formels divers, nous permet d'envisager des mecanismes d'analyse plus puissants. Ainsi, si l'on considère la description d'un processus comme un système de transitions d'états, une approche opérationelle doit nous permettre d'étudier des proprietés telles que : invariants, trajectoires, terminaison, "dead lock", "liveness",...

En considérant la description d'un processus comme un système de réécriture de termes auquels on ajoute des événements, une étude sur le problème de la decidabilité de terminaison d'une règle de transition de termes a été faite <Per 84>. La solution d'un tel problème permet de décider entre autre, de la divergence dans une grande famille de processus.

La validation, c'est à dire, la fonction de comparaison entre deux descriptions de systèmes (une étant la spécification et l'autre une implementation), et la synthèse de processus sont également en cours d'étude.

Enfin, un langage formel pour SPARC est proposé ici : FP2. Dans ce langage sont intégrés des descriptions de fonctions, de types et de processus communicants avec la possibilité de paramètrer tout cela par des types.

2.1.3. Caractéristiques de SPARC.

SPARC est un système où l'on veut à la fois :
- Décrire, au moyen d'un langage.
- Analyser et valider, en s'appuyant sur des modèles sémantiques.

Le modèle proposé, SPARC, a des caractéristiques qui, pour la plupart, ne sont par offertes simultanément dans d'autres modèles :

Un système où interviennent le parallélisme et la communication sera décrit par un réseau de processus interconnectés. Ces processus fonctionnent indépendamment les uns des autres, et seront synchronisés par échanges de messages (communication par rendez-vouz).

Un processus sera décrit, indépendamment de tout autre processus, par la façon dont ses "utilisateurs" ont connaissance de ses capacités de communication et peuvent en faire usage. La description d'un processus jouera donc, le rôle d'un "mode d'emploi" pour ce processus et devra être exprimée sans faire mention du fonctionnement interne du processus.

Un processus isolé n'est qu'un cas particulier d'un réseau de processus : les capacités de communication d'un réseau, tant entre deux processus que le composent, qu'entre le réseau et l'extérieur, doivent être descriptibles dans les mêmes termes généraux que celles d'un processus isolé. De plus, ces capacités de communication ne dépendent que de celles des processus composants et des connexions qui ont été établies : la description du "mode d'emploi" d'un réseau, doit donc, pouvoir être obtenue directement à partir d'une description de sa construction.

Les capacités de communication d'un processus ou d'un réseau, présentent plusieurs aspects qui, tous, doivent faire partie du "mode d'emploi" :

- Types de messages reçus et envoyés,
- Ordonnancement des commmunications, et
- Fonctions réalisées en sortie.

D'autres aspects comme les délais entre commmunications et la reprise d'erreur, seront un jour intégrés à la description d'un processus.

La construction de réseaux, par assemblage et interconnexion de processus, devra donc, utiliser ces informations sur les processus composants pour élaborer systématiquement les informations correspondantes sur le réseau construit.

2.2. Description d'un processus communicant.

Cette section et la suivante, 2.3, sont basées sur <Jor 81a>, <Jor 81b> et <Jor 83>.

Un processus est une entité capable de communiquer, c'est à dire, d'envoyer et de recevoir des messages.

Un processus peut réaliser des fonctions : calculs sur les valeurs reçues, simulation de structures de données, execution de différentes formes de contrôle sur des séquences de valeurs, etc.

Grâce à ces deux caractéristiques, un processus est une entité extrêmement puissante. Nous pouvons distinguer deux parties dans la description d'un processus : une partie de déclarations expliquée ci dessous et une autre définissant le comportement du processus. Le comportement d'un processus est la donnée des séquences possibles de communication et des valeurs possibles qui sont communiqués. Le comportement est décrit par un ensemble de règles.

Declarations

Tout processus possède un certain nombre des portes, chaque porte ayant un nom.

Tout échange de message s'éffectue le long d'un connecteur, chaque connecteur reliant deux portes ou reliant une porte avec l'extérieur. Ainsi, si A et B sont des noms de portes, A.B, B.A, A.?, ?.B, ?.A et B.? sont des noms de connecteurs. A.B et B.A nomment des connecteurs internes et les autres des connecteurs externes. Le symbole "?" est utilisé pour désigner les portes anonymes de l'environnement ; ainsi, le

connecteur ?.A exprime la possibilité de recevoir des messages de l'extérieur à travers la porte A et B.? exprime la même chose mais pour envoyer des valeurs : ?.A est un connecteur d'entrée et B.? un connecteur de sortie.

Une même porte peut servir à la fois pour un connecteur d'entrée et pour un connecteur de sortie.

La première chose à déclarer quand on décrit un processus est l'ensemble des connecteurs qu'il possède avec le type des valeurs qui circuleront le long de chaque connecteur. Le reste des déclarations concerne les règles de comportement. Une règle définit une transition d'état au cours de laquelle un événement (ensemble de communications simultanées) a lieu.

Un état est composé d'un "terme d'état" de la forme $S(a_1,\ldots,a_n)$ où S est un symbole d'état et les arguments $a_1,\ldots,a_n$ sont des expressions construites à partir des noms de fonctions associées aux types abstraits et peuvent également utiliser des noms de variables : ce sont des termes, au sens des termes d'un univers de Herbrand.

Une deuxième partie des déclarations introduit donc, l'ensemble des symboles d'états apparaissant dans les règles, accompagnés du type de leurs argumments. Par exemple :
Q0:(), Q1,Q2:Nat Bool, Q3:Seq
sont des déclarations de symboles d'états.

Troisièmement, on définit l'état initial, c'est à dire, l'état à partir duquel le comportemment commencera. Dans les arguments de l'état initial il ne peut pas apparaître de variables.

Enfin, on déclare dans une dernière rubrique, l'ensemble de variables utilisées dans les règles avec leurs types. Ainsi,

pour tout processus, on doit déclarer :

1. Son nom
2. L'ensemble des connecteurs avec leur types associés.
3. L'ensemble des symboles d'état avec le type de leurs arguments.
4. L'état initial.
5. L'ensemble des variables avec leurs types associés.

Comportement

Un ensemble de règles définit les séquences possibles d'événements et les fonctions calculées par le processus. Ces règles ont la forme et l'interprétation de règles de réecriture. Leur membre gauche et leur membre droit sont des termes d'états contenant des variables.

Un événement est associé à chaque règle, c'est à dire, un ensemble de communications simultanées où chaque communication est décrite par une paire <c,m> : c est un connecteur et m est une expression qui définit la valeur du message qui circule le long du connecteur c. L'expression m est construite avec les fonctions de types abstraites et peut faire usage de variables.

Si G et D sont des termes d'états contenant des variables, et si e est un événement, une règle est donc un triplet (G,e,D). Si E est l'état courant et si s, fonction de substitution la plus générale telle que s(E)=s(G), la règle (G,e,D) est alors applicable, le nouvel état est s(D) et l'application de cette règle correspond à l'occurence de l'événement s(e).

Quand plusieurs règles sont applicables le processus a un comportement non-déterministe. Si E est l'état courant et si r1, r2 sont des règles applicables (s1(E)=s1(G1), s2(E)=s2(G2)), si s1(e1)≠s2(e2) alors il s'agit d'un non déterminisme interne de

la machine. Si sl(el)=s2(e2) le non déterminisme sera resolu par l'environnement et on parlera d'un non déterminisme de l'environnement.

Dans une règle, l'ensemble des communications peut être vide, et dans ce cas on parle de l'événement nul e=$\Im$. La règle (G,$\Im$,D) exprime une transition interne du processus au cours de laquelle il n'y a pas de communications.

Le comportement d'un processus est donc, décrit par l'application de règles de la forme (G,e,D), qui effectuent des changements d'états par réecriture à partir d'un état initial.

2.2.1. Exemples

Nous présentons maintenant quelques exemples de processus. Les types et opérateurs utilisés sont prédéfinis en LPG.

Exemple No.1

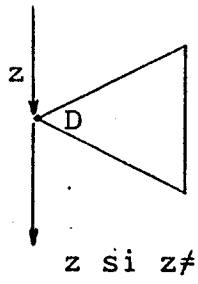
Nous allons décrire un processus FILTRE-0 dont la fonction est de "laisser passer" des valeurs en filtrant certaines. Dans ce cas particulier, nous avons une séquence de valeurs de type Naturel et nous sommes intéressés à filtrer la valeur zero. C'est un processus très simple qui a une seule porte, D, à laquelle sont attachés un connecteur d'entrée et un connecteur de sortie :

```
processus FILTRE-0
cnx   ?.D, D.? : Nat
etats R : ()
init  R
vars  z : Nat
```

```
regles
      R => ?.D(succ(z)) D.?(succ(z))  R
fin
```

Le processus FILTRE-0 a une seule règle dont l'événement est l'occurrence simultanée de communications le long des connecteurs ?.D et D.?. Les valeurs permises à travers ces connecteurs sont les naturels supérieures à zero, cela étant exprimé par le terme succ(z).

Graphiquement on a :

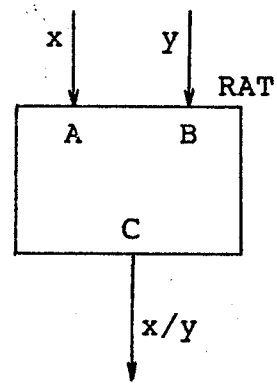


FILTRE-0

Exemple No.2

Nous voulons créer un processus qui génère les nombres Rationels, Rat. Pour ce faire, nous avons besoin de deux portes d'entrée par lesquelles on recevra des Naturels et d'une porte de sortie par laquelle on transmettra la fraction correspondante.

```
processus RAT
cnx   ?.A, B.? : Nat, C.? : Rat
etats Q : ()
init  Q
vars  x,y : Nat
regles
      Q => ?.A(x) ?.B(y) C.?(x/y)  Q
fin
```



Comme dans le cas antérieur on a une seule règle exprimant la formation des rationnels à partir de deux naturels et tout se passe simultanément.

Pour que le fonctionnement du processus RAT soit correct il faudra interdire le passage de la valeur zero par la porte B, qui correspond au denominateur. On voit déjà ainsi, un usage possible du processus FILTRE-0, qu'il faudra "brancher" sur le connecteur qui arrive à B.

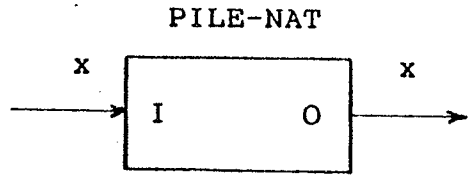
Exemple No.3

Le mecanisme proposé permet de décrire une classe très large de comportements non réguliers et même non "context free". Ceci est dû au fait qu'un procédé d'unification portant sur les termes d'états gouverne l'applicabilité des règles. Trois exemples vont permettre d'illustrer cette affirmation.

Exemple 3.1
PILE INFINIE D'OBJETS DE TYPE Nat. (NON-REGULIER)

Une pile infinie, où l'on entre les valeurs par une porte I et où on les sort par 0, n'a évidemment pas un comportement régulier : à chaque instant de son utilisation, le nombre de communications à travers 0 est inférieur ou égal au nombre de communications à travers I.

processus PILE-NAT
cnx ?.I : Nat, O.? : Nat
etats Q : Seq
init Q (Λ)
vars s : Seq, x : Nat
regles



```
        Q(s) => ?.I(x)  Q(cons(s,x))
        Q(cons(s,x)) => 0.?(x) Q(s)
fin
```

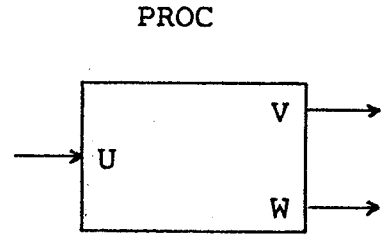
L'argument de Q est une séquence de Naturels. Λ dénote la séquence vide. La première règle décrit la réception de valeurs et construit une séquence en accumulant les valeurs reçues. La deuxième règle vérifie que cette séquence n'est pas vide (cons(s,x)), transmet la dernière valeur enregistrée et enlève cette valeur de la séquence.

Exemple 3.2

PROCESSUS AVEC COMPORTEMENT NON CONTEXT-FREE

Le processus PROC a une porte d'entrée U et deux portes de sortie V et W. Il reçoit n signaux par ?.U, puis il envoie n signaux par V, puis n signaux par W. C'est un processus qui correspond au langage non context-free $U^nV^nW^n$.

```
processus PROC
cnx   ?.U : Sig, V.?, W.? : Sig
etats A : Nat, B,C : Nat Nat
init  A(0)
vars  m,n : Nat
regles
      A(m)                 => ?.U(zz) A(succ(m))
      A(succ(m))           => V.?(zz) B(succ(m),m)
      B(succ(m),succ(n))   => V.?(zz) B(succ(m),n)
      B(succ(m),0)         => W.?(zz) C(succ(m),m)
      C(succ(m),succ(n))   => W.?(zz) C(succ(m),n)
      C(succ(m),0)         => ?.U(zz) A(succ(0))
fin
```



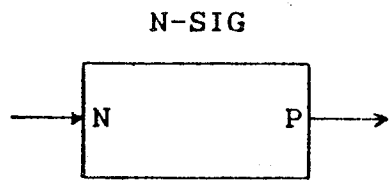
où zz est une constante de type Signal.

Exemple 3.3

COMPORTEMENT QUI DEPEND DES VALEURS TRANSMISES

Description d'un processus qui reçoit un Naturel n par sa porte N, puis envoie n signaux par sa porte P.

```
processus N-SIG
cnx    ?.N : Nat, P.? : Sig
etats  R : Nat
init   R(0)
var    n : Nat
regles
       R(0)        => ?.N(n)  R(n)
       R(succ(n)) => P.?(zz) R(n)
fin
```



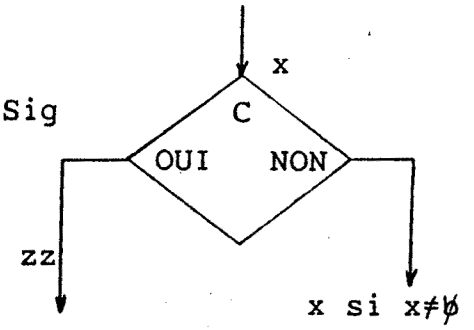
Exemple No.4

Description d'un processus qui, étant donnée une séquence de caractères, détecte le caractère ϕ en envoyant un signal ; dans le cas où le caractère n'est pas ϕ, il le "laisse passer".

La resemblance avec l'exemple No.1 est évidente. Mais on ne peut pas l'implementer avec un FILTRE-0 car il s'agit de types d'objets différents : pour les entiers, toutes les valeurs différentes de zero sont représentées par le terme succ(x), tandis que pour les caractères, nous avons besoin d'une condition pour détecter le blanc.

Le processus que l'on va décrire est donc, une condition. Il reçoit des valeurs de type caractère par sa porte C, il se demande si cette valeur est un blanc et, si oui, il envoit un signal par sa porte OUI ; dans le cas contraire il laisse passer la valeur par sa porte NON.

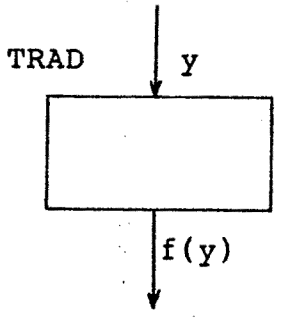
```
processus COND
cnx   ?.C, NON.? : Car, OUI.? : Sig
etats Q :(), Q1 : Car, Bool
init  Q
vars  x : Car
regles
      Q          => ?.C(x)     Q1(x,x=∅)
      Q1(x,vrai) => OUI.?(zz)  Q
      Q1(x,non)  => NON.?(x)   Q
fin
```



Exemple No.5

Description d'un processus qui applique une fonction f aux valeurs d'entrée et transmet le résultat en sortie. Supposons que f est définie sur les caractères et produit un caractère. Ce processus est donc, un "traducteur' de caractères.

```
processus TRAD
cnx   ?.A, B.? : Car
etats R : ()
init  R
vars  y : Car
regles
      R => ?.A(y) B.?(f(y)) R
fin
```



La fonction f pourrait être définie à travers de règles dans le même processus TRAD. Nous avons préféré faire appel au nom f pour simplifier. Dans le troisième chapitre, on verra dans les exemples, des fonctions calculées par le processus lui même.

Exemple No.6

Le processus suivant simule une structure de données : le buffer. Il reçoit des caractères qu'il enregistre pour former des mots : quand un signal arrive, il vide son contenu et transmet un mot.

```
processus MOT
cnx   ?.D : Car, ?.F : Signal, E.? : Chaine
etats T : Chaine
init  T(vide)
vars  m : Chaine, u : Car
regles
      T(m) => ?.D(u)          T(conc(m,ch(u))
      T(m) => ?.F(zz)E.?(m) T(vide)
fin
```

où conc est une fonction qui concatène deux chaînes et ch est la fonction qui convertit un caractère en chaîne.

Le processus MOT possède deux règles dont les membres gauches sont égaux. Néanmoins, les événements sont différents, ce qui exprime le non déterminisme de l'environnement.

2.3. Construction de réseaux de processus

Pour la construction de réseaux de processus, SPARC définit trois opérateurs :

l'opérateur d'union

‖ : processus x processus --> processus

l'opérateur de connexion

\+ : processus x connecteur interne --> processus

l'opérateur d'abstraction

\- : processus x connecteur --> processus

L'évaluation des expressions de processus se fait à partir de la définition de chaque opérateur et ce ne sont que des manipulations au niveau syntaxique : en partant de descriptions de processus on obtient la description du réseau, c'est à dire, d'un autre processus.

2.3.1. L'opérateur d'union

P=P1‖P2 signifie que l'on décrit un nouveau processus P qui sera le produit des processus P1 et P2 fonctionnant en parallèle : soit P1 seul, soit P2 seul, soit P1 et P2 simultanés.

Nous illustrerons ici la définition de l'opérateur ‖ à travers un exemple. Les commentaires seront précedés du symbol "--". Les définitions formelles seront présentées au paragraphe 2.3.4. Nous allons reprendre les exemples No.1 et No.2 de la section précedente (FILTRE-0 et RAT).

Calcul de P=P1‖P2

1. L'ensemble des connecteurs de P est l'union de l'ensemble des connecteurs de P1 et de l'ensemble de connecteurs de P2. Les noms de porte de P1 et P2 doivent être differents.
   Dans notre exemple : NON-ERR = FILTRE-0 ‖ RAT.
   processus NON-ERR
   cnx ?.D, D,?, ?.A, B.? : Nat, C.? : Rat

2. L'ensemble de symboles d'état de P est donnée par le produit des symboles d'état de P1 et des symboles d'état de P2.
   processus NON-ERR
   :
   etats T : () [T est un nom arbitraire pour représenter R() * Q()]

3. L'état initial de P est le produit des états initiaux de P1 et P2.

```
processus NON-ERR
    :
    :
etat init T
```

4. L'ensemble des variables de P sera l'union de l'ensemble des variables de P1 et de l'ensemble de variables de P2.

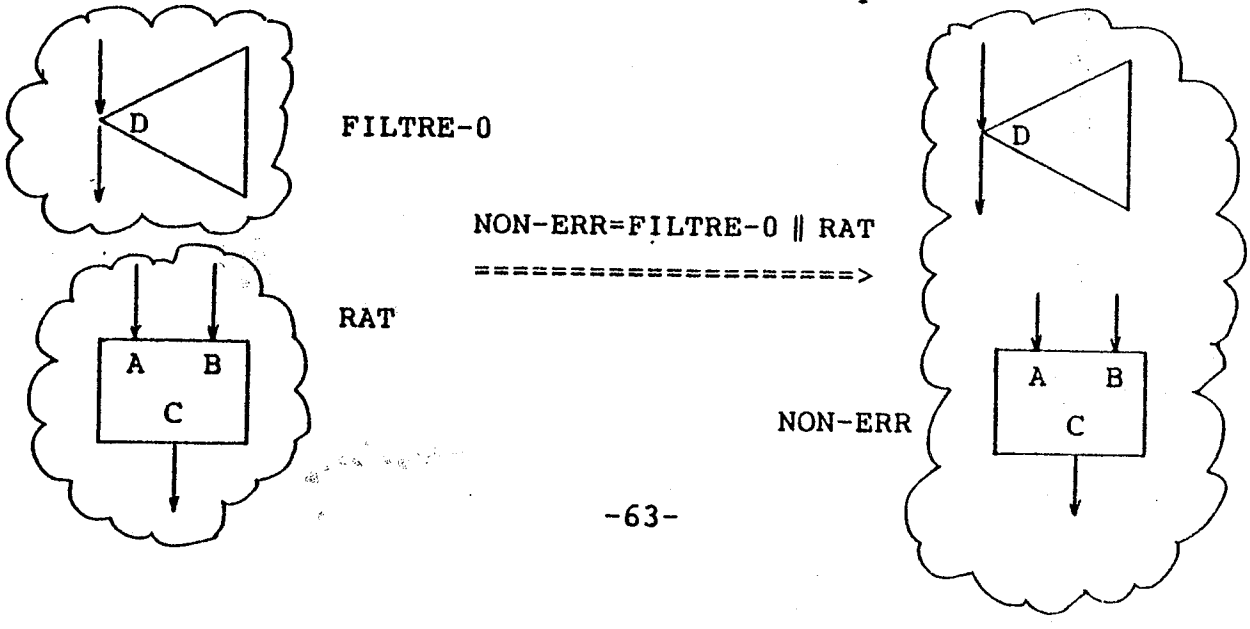
```
processus NON-ERR
    :
    :
vars z,x,y : Nat
```

5. L'ensemble des règles de P doit exprimer :
   - le comportement de P1 tout seul
   - le comportement de P2 tout seul
   - le comportement simultanné de P1 et P2.

```
processus NON-ERR

regles
     T => ?.D(succ(z))D.?(succ(z)) T  -- FILTRE-0 seul
     T => ?.A(x)?.B(y)C.?(x/y)     T  -- RAT seul
     T => ?.D(succ(z))D.?(succ(z))?.A(x)?.B(y)C.?(x/y) T
                          -- FILTRE-0 et RAT simultanés
fin
```

On peut symboliser graphiquement cette opération :



Nous allons calculer maintenant TRAD-M=COND‖TRAD‖MOT (exemples 4, 5 et 6 de la section précedente).

Calcul des symboles d'états de TRAD-M :

Q:() x R:() x T:Chaine = W:Chaine

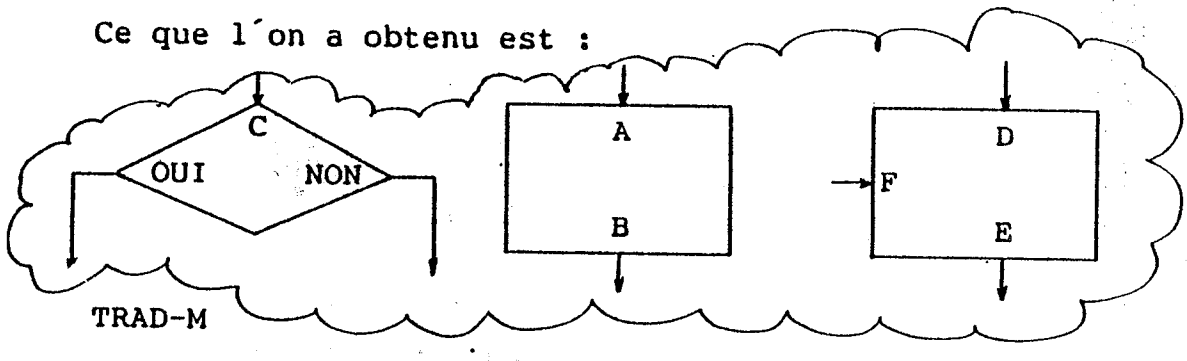
Q1:Car Bool x R:() x T:Chaine = U:Car Bool Chaine

```
processus TRAD-M
cnx  ?.C,NON.?,?.A,?.B,?.D:Car, OUI.?,?.F:Signal,
     E.?:Chaine
etats W:Chaine, U:Car Bool Chaine
init W(vide)
vars x,y,v:Car, m:Chaine
regles
     -- COND seul
     W(m)          => ?.C(x)            U(x,x=∅,m)
     U(x,vrai,m)   => OUI.?(zz)         W(m)
     U(x,faux,m)   => NON.?(x)          W(m)
     -- TRAD seul
     W(m)          => ?.A(y)B.?(f(y))   W(m)
     U(x,vrai,m)   => ?.A(y)B.?(f(y))   U(x,vrai,m)
     U(x,faux,m)   => ?.A(y)B.?(f(y))   U(x,faux,m)
     -- MOT seul
     W(m)          => ?.D(u)            W(conc(m,ch(u)))
     W(m)          => ?.F(zz)E.?(m)     W(vide)
     -- COND, TRAD et MOT simultanés
     W(m)          => ?.C(x)?.A(y)B.?(f(y))?.D(u)
                                        U(x,x=∅,conc(m,ch(u)))
     W(m)          => ?.C(x)?.A(y)B.?(f(y))?.F(zz)E.?(m)
                                        U(x,x=∅,vide)
     U(x,vrai,m)   => OUI.?(zz)?.A(y)B.?(f(y))?.D(u)
                                        W(conc(m,ch(u)))
     U(x,vrai,m)   => OUI.?(zz)?.A(y)B.?(f(y))?.F(zz)E.?(m)
                                        W(vide)
```

```
U(x,faux,m) => NON.?(x)?.A(y)B.?(f(y))?.D(u)
                                        W(conc(m,ch(u)))
U(x,faux,m) => NON.?(x)?.A(y)B.?(f(y))?.F(zz)E.?(m)
                                        W(vide)
-- COND et TRAD simultanés
W(m)        => ?.C(x)?.A(y)B.?(f(y))
                                        U(x,x=∅,m)
U(x,vrai,m) => OUI.?(zz)?.A(y)B.?(f(y)) W(m)
U(x,faux,m) => NON.?(x)?.A(y)B.?(f(y))  W(m)
-- TRAD et MOT simultanés
W(m)        => ?.A(y)B.?(f(y))?.D(u)    W(conc(m,ch(u)))
W(m)        => ?.A(y)B.?(f(y))?.F(zz)E.?(m) W(vide)
U(x,vrai,m) => ?.A(y)B.?(f(y))?.D(u)
                                        U(x,vrai,conc(m,ch(u)))
U(x,vrai,m) => ?.A(y)B.?(f(y))?.F(zz)E.?(m)
                                        U(x,vrai,vide)
U(x,faux,m) => ?.A(y)B.?(f(y))?.D(u)
                                        U(x,faux,conc(m,ch(u)))
U(x,faux,m) => ?.A(y)B.?(f(y))?.F(zz)E.?(m)
                                        U(x,faux,vide)
-- COND et MOT simultanés
W(m)        => ?.C(x)?.D(u)     U(x,x=∅,conc(m,ch(u)))
W(m)        => ?.C(x)?.F(zz)E.?(m) U(x,x=∅,vide)
U(x,vrai,m) => OUI.?(zz)?.D(u) W(conc(m,ch(u)))
U(x,vrai,m) => OUI.?(zz)?.F(zz)E.?(m) W(vide)
U(c,faux,m) => NON.?(x)?.D(u)  W(conc(m,ch(u)))
U(x,faux,m) => NON.?(x)?.F(zz)E.?(m)  W(vide)
```

fin

Ce que l'on a obtenu est :



2.3.2. L'opérateur de connexion

Etant donné un processus P1 et un connecteur interne 0.I tel que 0.? est un connecteur de sortie et ?.I un connecteur d'entrée de P1, P=P1+0.I exprime la connexion entre la porte de sortie 0.? et la porte d'entrée ?.I du processus P1. Une communication de valeurs par "rendez vous" peut desormais avoir lieu de 0 à I : dès que 0.? a quelque chose à envoyer et que ?.I peut recevoir, l'envoit et la reception se passent simultanément.

Calcul de P=P1+0.I

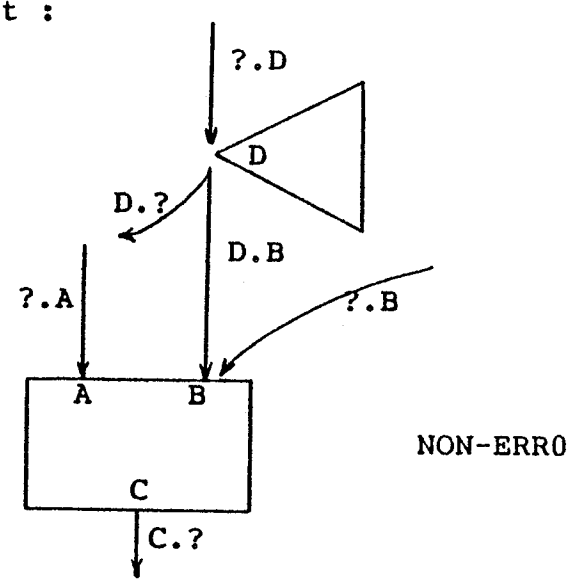
1. Vérifier qu'il existe dans l'ensemble de règles de P1, au moins une règle où les communications 0.? et ?.I font partie du même événement, c'est à dire, qu'elles peuvent avoir lieu simultanément.
2. Vérifier que le terme dénotant les valeurs transmises par 0.? soit unifiable avec celui de ?.I, c'est à dire :
   soient 0.?(t1) et ?.I(t2), il existe une substitution θ telle que $\theta(t1)=\theta(t2)$. Voir <Cis 81>.
3. Ajouter 0.I à la rubrique cnx (connecteurs).
4. Pour chaque règle r trouvée en 1. ajouter une nouvelle règle r_i' où $r_i' = r_i$ avec les modifications suivantes :
   - enlèver 0.?(t1) et ?.I(t2) de l'événement, et ajouter 0.I(θ(t1)
   - faire sur toute la règle, la substitution θ décrite en 2.

Pour notre premier exemple NON-ERR, le résultat de la connexion NON-ERR0=NON-ERR+D.B est :

```
processus NON-ERRO
cnx  ?.D,D.?,?.A,B.?,D.B : Nat, C.? : Rat

regles
     T => ?.D(succ(z)) D.?(succ(z)) T
     T => ?.A(x) ?.B(y) C.?(x/y)    T
     T => ?.D(succ(z))D.?(succ(z))?.A(x)?.B(y)C.?(x/y) T
     T => ?.D(succ(z))D.B(succ(z))?.A(x)C.?(x/succ(z)) T
fin
```

Graphiquement :



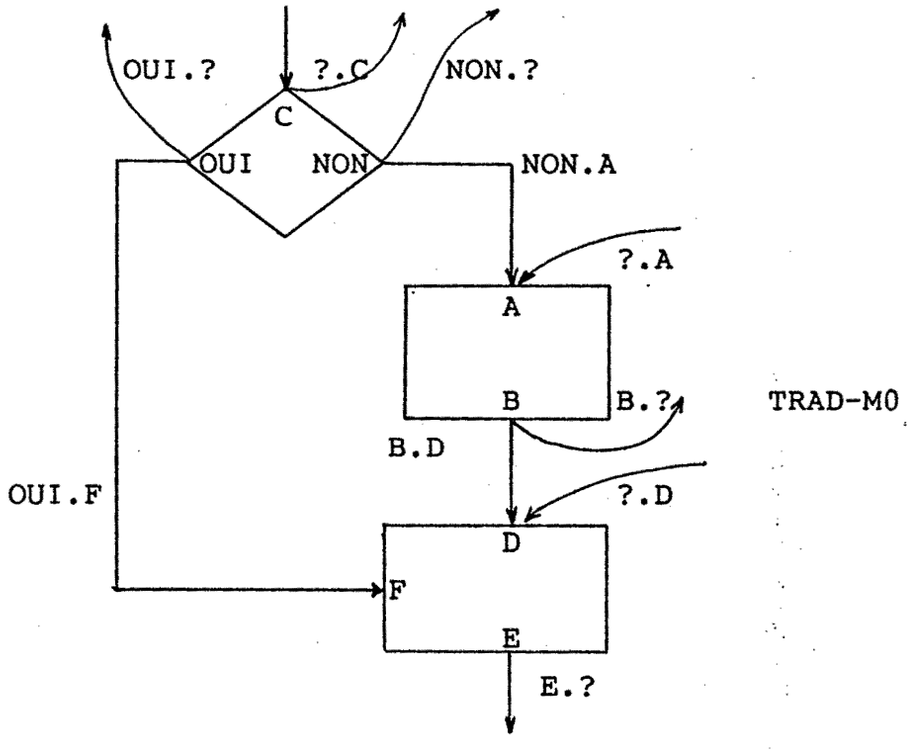
Dans le deuxième exemple nous allons faire les connexions : TRAD-MO = TRAD-M + NON.A + B.D + OUI.F

```
processus TRAD-M0
cnx ?.C,NON.?,?.A,?.B,?.D,B.D : Car,
    OUI.?,?.F,OUI.F : Signal,  E.? : Chaine
  :
regles
  :          toutes celles qui'il y avait plus
```

```
U(x,faux,m) => NON.A(x)B.?(f(x))  U(x,x=∅,m)
W(m)        => .?.A(y)B.D(f(y))   W(conc(m,ch(f(y))))
U(x,vrai,m) => ?.A(y)B.D(f(y))    U(x,vrai,conc(m,ch(f(y))))
U(x,faux,m) => ?.A(y)B.D(f(y))    U(x,faux,conc(m,ch(f(y))))
U(x,vrai,m) => OUI.F(zz) E.?(m)   U(x,x=∅,vide)
W(m)        => ?.C(x)?.A(y)B.D(f(y))
                                  U(x,x=∅,conc(m,ch(f(y))))
U(x,vrai,m) => OUI.?(zz)?.A(y)B.D(f(y)) W(conc(m,ch(f(y))))
U(x,vrai,m) => OUI.F(zz)?.A(y)B.?(f(y))E.?(m) W(vide)
U(x,faux,m) => NON.A(x) B.D(f(x)) W(conc(m,ch(f(x))))
U(x,faux,m) => NON.A(x)B.?(f(x))?.F(zl)E.?(m) W(vide)
```

fin

Graphiquement :



2.3.3. L'opérateur d'abstraction

Soit P1 un processus. Le processus P=P1-C où C peut être

?.I connecteur externe

0.? connecteur externe

0.I connecteur interne

est un processus où l'on a "caché" le connecteur C. Une fois que l'on a construit un réseau, il peut être intéressant de faire abstraction de la façon dont il a été construit et de ne laisser "voir" que certains connecteurs. Cela se fait par l'opérateur "-".

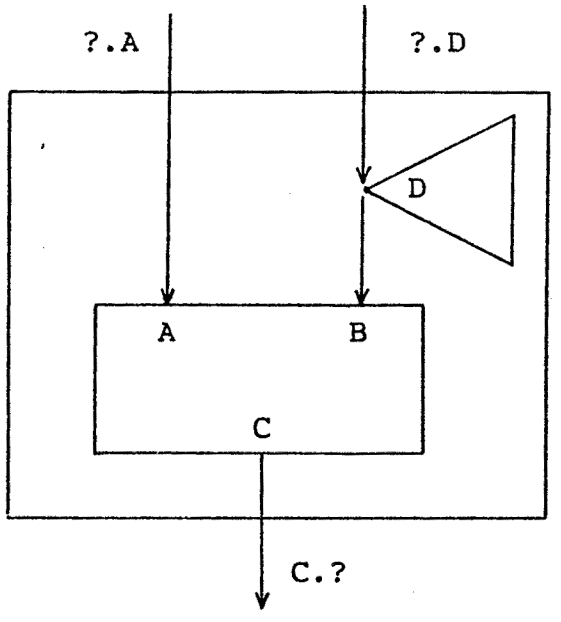
Calcul de P=P1-C

- Enlever C de la rubrique cnx (connecteurs).
- Si C est un connecteur externe, alors enlèver toutes les règles où C fait partie de l'événement.
- Si C est un connecteur interne et r_i est une règle où C apparaît dans l'événement, il faut enlèver C de l'événement et conserver r_i ainsi modifiée.

En continuant avec nos exemples, pour le premier, nous allons faire NON-ERROR=NON-ERR0 - D.? - ?.B - D.B. Ceci nous donne la description suivante :

```
processus NON-ERROR
cnx   ?.D,?.A : Nat, C.? : Rat
etats T : ()
init  T
vars  z,x : Nat
regles
      T => ?.D(succ(z)) ?.A(x) C.?(x/succ(z)) T
fin
```

Graphiquement :



NON-ERROR

Le réseau NON-ERROR est un processus qui génère des nombres rationnels à partir de deux entiers pour lesquels on a assuré qu'aucune valeur nulle ne sera dénominateur de la fraction.

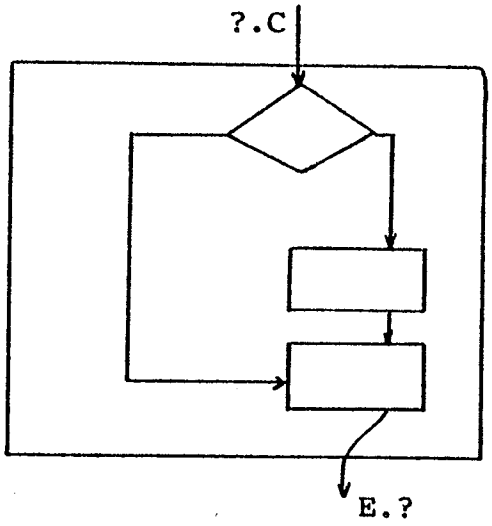
Reprenons maintenant notre deuxième exemple TRAD-MO. Nous allons faire l'abstraction :
TRAD-MOT=TRAD-MO-NON.?-?.A-NON.A-B.?-?.D-B.D-OUI.?-?.F-OUI.F

processus TRAD-MOT
cnx ?.C : Car, E.? : Chaine
etats W : Chaine, U : Car Bool Chaine
init W(vide)
vars x,y,u : Car, zz,zl : Signal, m : Chaine
regles

```
W(m)          => ?.C(x)   U(x,x=∅,m)
U(x,faux,m)   => ∫         W(conc(m,ch(f(x))))
U(x,vrai,m)   => E.?(m)    W(vide)
```

fin

Graphiquement :



Le processus TRAD-MOT reçoit des caractèrees qu'il traduit et enregistre jusqu'a recevoir le caractère blanc qui détecte fin de mot ; quand TRAD-MOT reçoit un blanc par C, il envoit par E un mot qui est déjà traduit caractère par caractère.

2.4. Description formelle.

2.4.1. Les processus comme algèbres de termes.

2.4.1.1 Définitions préliminaires

Soient : A un ensemble de "sortes"
P un ensemble de noms de portes
X un ensemble de variables
F un ensemble de noms de fonctions
(F et X sont des ensembles disjoints).

Soit α : F -> A^* l'arité d'une fonction
X -> $\mathcal{E}$ l'arité d'une variable
Soit β : F -> A la sorte d'une fonction
X -> A la sorte d'une variable

Les deux sous-ensembles suivantes sont définis :

$F[s,a] = \{f \in F \mid \alpha(f)=s,\ \beta(f)=a\}$ où $s=a_1 a_2 \ldots a_n$

et

$X[a] = \{x \in X \mid \beta(x)=a\}$

2.4.1.2 Termes Fonctionnels

U[F,X,a] est l'ensemble des termes fonctionnels sur les noms de fonctions F, les noms de variables X et de sorte a. U[F,X,a] est défini inductivement par :

1) $X[a] \subseteq U[F,X,a]$
2) $F[\mathcal{E},a] \subseteq U[F,X,a]$
3) pour tout $f \in F[a_1 \ldots,a_n,a]$,
   pour tout $u_i \in U[F,X,a_i]$, $i=1,2,\ldots,n$
   $f(u_1,\ldots,u_n) \in U[F,X,a]$

2.4.1.3 Termes d'état

Soit G un ensemble de "symboles d'états" (G,X,F sont disjoints). Soit α : G -> A^*, l'arité d'un symbole d'état. Les sous-ensembles suivants sont definis :

$G[s]=\{g \in G \mid \alpha(g)=s\}$ avec $s=a_1,\ldots,a_n$

V[F,X,G] est l'ensemble des termes d'état sur les noms de fonctions F, les noms de variables X et les symboles d'état G. V[F,X,G] est définit inductivement par :

1) $G[\mathcal{E}] \subseteq V[F,X,G]$
2) pour tout $g \in G[a_1,\ldots,a_n]$,
   pour tout $u_i \in U[F,X,a_i]$, $i=1,2,\ldots n$,
   $g(u_1,\ldots,u_n) \in V[F,X,G]$

2.4.1.4 Termes d'événement

Soit H[K] l'ensemble des noms d'événements défini par $H[K]=2^K$ où K est l'ensemble des connecteurs défini par

K = Kint U Kext
Kint = P×P (ensemble de connecteurs internes)
Kext = Kin U Kout (ensemble de connecteurs externes)
Kin = {?}×P (ensemble de connecteurs d'entrée)
Kout = P×{?} (ensemble de connecteurs de sortie)

Une application η appelée "sorte de message", est définie sur K par η : K->A. Etant $\pi[K]$: H[K]->K^* une fonction donnant une séquence arbitraire des connecteurs contenus dans un ensemble de connecteurs, nous pouvons étendre la définition de η à H[K] :
pour tout $h \in H[K]$, $\pi[K](h)=k1,k2,\ldots,kn \Rightarrow \eta(h)=\eta(k1)\ldots\eta(kn)$
Les sous ensembles suivants sont définis :
$H[K,s] = \{h \in H[K] \mid \eta(h)=s\}$ où $s=a_1,\ldots,a_n$, pour tout n.

Enfin, soit σ le nom de l'événement initial que l'on peut interpréter comme l'événement qui "donne naissance" à un processus. W[F,X,G,K] est l'ensemble des termes d'événement sur les noms de fonctions F, les noms de variables X, les noms d'états G et les noms de connecteurs K. W[F,X,G,K] est défini inductivement par :

1) $\sigma \in W[F,X,G,K]$
2) $\forall h \in H[K,a_1 \ldots a_n]$, $\forall u_i \in U[F,X,a_i]$, i=1,...,n ,
   $\forall v \in V[F,X,G]$, $h(u_1,\ldots,u_n,v) \in W[F,X,G,K]$
3) $\forall h \in H[K,a_1 \ldots a_n]$, $\forall u_i \in U[F,X,a]$, i=1,...,n ,
   $\forall w \in W[F,X,G,K]$, $h(u_1,\ldots,u_n,w) \in W[F,X,G,K]$

Un terme d'événement construit par la règle 2 est appelé terme d'événement simple. Si u est un terme fonctionnel, terme d'état ou terme d'événement, var(u) désigne l'ensemble des variables apparaissant dans le terme u.

2.4.1.5 Termes de transition

Soit $w = h(u_1,\ldots,u_n,v)$ un terme d'événement simple,
où $\pi[K](h) = k1,\ldots,kn$.
Soient X = var(v)
Xin = U var(u_i), $k_i \in$ Kin
Xout = U var(u_i), $k_i \in$ Kint U Kout
trois ensembles de noms de variables.
W est un terme de transition s'il possède la propriété Xout $\subseteq$ X U Xin ; c'est à dire, si la valeur des variables utilisées en sortie n'est fonction que des valeurs que l'on avait, ou des valeurs qui viennent d'entrer.

2.4.1.6 Syntaxe et Sémantique de Processus

Etant donné K, ensemble fini de noms de connecteurs et G, ensemble fini non vide de symboles d'état avec leurs fonctions d'arité η et α,
Σ = K U G est une signature de processus
Une signature Σ=KUG définit l'ensemble des termes syntaxiquement corrects. Etant donné un ensemble de noms de fonction F, cet ensemble de termes est :
T[Σ] = V[F,Φ,G] U W[F,Φ,G,K]

Etant donné Σ, une signature de processus et X, un ensemble de noms de variable, le couple :
<w,v> est une équation de processus
où $w \in$ W[F,X,G,K] est un terme de transition et $v \in$ V[F,var(w),G]
Une équation est "initiale" si $w=\sigma$ et $v \in$ V[F,Φ,G]. Une équation est non initiale si $w \neq \sigma$.

Syntaxe

Une description de processus est une présentation P=<Σ,I,E> où
Σ est une signature de processus
I est un ensemble non-vide d'équations initiales
E est un ensemble d'équations non-initiales.

Reprenons l'exemple No.4 de la section 2.2.1, le processus COND, qui avait été informellement décrit :

```
processus COND
cnx   ?.C,NON.? : Car, OUI.? : Sig
etats Q : (), Q1 : Car, Bool
init  Q
vars  x : Car
regles
      Q          => ?.C(x)     Q1(x,x=∅)
      Q1(x,vrai) => OUI.?(zz)  Q
      Q1(x,faux) => NON.?(x)   Q
fin
```

Formellement, la description du processus COND est la suivante :

```
COND = processus
    K = {?.C,NON.?,OUI.?}
        η(?.C)    = Car
        η(NON.?)  = Car
        η(OUI.?)  = Car
    G = {Q,Q1}                          } Signature de processus
        α(Q)  = ε
        α(Q1) = Car Bool
    X = {x}
        β(x) = Car
    I = {<σ,Q>}                         } Equat. initiales
    E = {<{?.C} (x,Q), Q1(x,x=∅)>,      } Equations non
         <{OUI.?}(zz,Q1(x,vrai)),Q>     }
```

< NON.? (x,Q1(x,faux)),Q>} initiales

fin

Néanmoins, la première description est beaucoup plus claire et c'est celle là qu'on utilisera, tout en respectant les liens avec le formalisme présenté. Par exemple :

une règle v0 => e v1

où v0,v1 sont des termes d'état et e=k1(u1)...kn(un) est un événement sur h={k1...kn}, dénote une équation non initiale <h(l,v0),v1> où la liste l est une permutation des ui correspondants à l'arité de h.

En étant X0,X1,Xin et Xout des ensembles de noms de variables définies par :

X0 = var(v0)
X1 = var(v1)
Xin = U var(ui),ki ∈ Kin
Xout = U var(ui),ki ∈ Kint U Kout

alors X1 ⊆ X0 U Xin
Xout ⊆ X0 U Xin

Sémantique

La sémantique d'un processus est donnée par l'algèbre de termes $T_\Sigma(X)$, où X est un ensemble de noms de variables, qui satisfait la présentation <Σ,I,E>. Par exemple, les termes :

NON.?(b,?.C(b,NON.?(a,?.C(a,σ)))),
?.C(C,OUI.?(zz,?.C(b,σ))),...

sont des séquences de comportement possibles du processus COND car il sont des termes qui appartiennent à l'algèbre $T_{\Sigma_{COND}}(X_{COND})$.

2.4.2. Algèbre de processus

2.4.2.1 Définitions préliminaires

1. Produit de termes d'état

Soient G1 et G2 deux ensembles de symboles d'état avec leurs fonctions d'arité $\alpha 1$ et $\alpha 2$. Soit G[i] un ensemble de symboles d'état avec une fonction d'arité $\alpha[i]$ et tel que, il existe une application injective i : G1xG2 <-> G[i].
G[i] est le produit de G1 et G2 (G1.[i]G2) ssi pour tout g∈G[i], g=i(<g1,g2>) => $\alpha[i](g)=\Psi[i](\alpha 1(g1),\alpha 2(g2))$
où Ψ[i](s1,s2) est un "merge" arbitraire de s1 et s2.

2. Produit de termes d'événement

Soient W1=W[F,X,G1,K] et W2=W[F,X,G2,K] deux ensembles de termes d'événement. Un ensemble de termes d'événement W est un produit de W1 et W2 (W1.W2), ssi W=W[F,X,G1.G2,K]. Ainsi, étant donnés deux termes d'événement simples :

w1=h1(m1,v1) ∈ W1 et w2=h2(m2,v2) ∈ W2

où h1 et h2 sont disjoints, il existe un seul élément w∈W tel que :

w=(h1 U h2) (Ψ[h1,h2,k](m1,m2),v1.v2)

où Ψ [h1,h2,k](m1,m2) est une permutation de m1 et m2 compatible avec l'arité de message de h1Uh2. Cet élément est noté w1.w2.

3. Produit d'équations

Soient Σ1=K1UG1 et Σ2=K2UG2 deux signatures de processus, telles que K1 et K2 sont des ensembles disjoints de noms de connecteurs. Une signature Σ est un produit de Σ1 et Σ2 (Σ1.Σ2), ssi Σ=(K1UK2) U (G1UG2).

Soit I1 un ensemble d'équations initiales sur Σ1 et soit I2 un ensemble d'équations initiales sur Σ2. Un ensemble I est un produit d'équations initiales I1 et I2 (I1.I2), ssi I est un ensemble d'équations initiales sur Σ1.Σ2 tel que :

$$I = \{ <\sigma, v1.v2> \; <\sigma, v1> \in I1, \; <\sigma, v2> \in I2 \}$$

Soient X1 et X2 deux ensembles disjoints de noms de variables. Soit E1 un ensemble d'équations non initiales sur Σ1 et X1, et soit E2 un ensemble d'équations non initiales sur Σ2 et X2. Un ensemble E est un produit d'équations non-initiales E1 et E2 (E1.E2) ssi E est un ensemble d'équations non initiales sur Σ1.Σ2 et X1UX2 tel que :

$$E = \{ <w1.w2, v1.v2> \mid <w1, v1> \in E1, \; <w2, v2> \in E2 \}$$

Le produit de termes d'état, de termes d'événement et d'équations est une opération commutative et associative dans les trois cas ; pour les démonstrations, voir <Jor 84a>.

4. Ensemble d'équations triviales, Z[G]

Soit G un ensemble de symboles d'état. Un terme d'état $v \in G$ est un terme d'état libre si $v \in V[\Phi, X, G]$ et il existe un nom de variable différent pour chaque position de son arité. L'ensemble de termes d'état libres appartenant à G est noté Y[G]. Une équation trivial sur G est une équation de la forme $<\Phi(v), (v)>$ où $v \in Y[G]$. L'ensemble d'équations triviales sur G est noté Z[G].

5. Substitutions et Unificateurs

Soit F un ensemble de noms de fonctions et X un ensemble de noms de variables avec les fonctions d'arité α et de sortie

β. L'application β est étendue en U[F,X]:

pour tout u = f(ul,...,un) $\in$ U[F,X], $\beta(u)=\beta(f)$

Une substitution est une application :

ρ: X -> U[F,X] telle que

$\beta(\rho(x)) = \beta(x)$

L'application d'une substitution ρ sur un terme t, donne un nouveau terme t' où toute ocurrence de x dans t est remplacé par $\rho(x)$ si x est dans le domaine de ρ.

Etant données deux substitutions ρ et ρ' dans le même domaine, ρ' est plus générale que ρ s'il existe une substitution ρ'' telle que :

$\rho = \rho'' \circ \rho'$, où $\circ$ dénote la composition de substitutions.

On dit que t1 et t2 sont unifiables s'il existe une substitution ρ telle que $\rho(t1)=\rho(t2)$. ρ est appelé l'unificateur de t1 et t2. Parmi toutes les substitutions possibles, il en existe une qui est plus générale que les autres, on la nomme le "plus général unificateur" de t1 et t2 ; soit, en abrégé, pgu(t1,t2).

6. Opérations sur les termes d'événement

Soit w=h(m,v) un terme d'événement simple où $m=u_1,\ldots,u_n$ et $\Pi[K](h)=k1\ldots kn$ défini sur un ensemble de noms de connecteurs K. Soit k un nom de connecteur, $k \in K$, et u un terme fonctionnel tel que $\beta(u)=\eta(k)$.

L'appartenance de la communication k(u) au terme w est notée $k(u) \in w$ et elle est définie :

si $k=k_i \in h$ et $u=u_i$, alors $k(u) \in w$=vrai

sinon $k(u) \in w$=faux

La soustraction de la communication k(u) du terme w est notée

w-k(u) et elle est définie :
si k(u)$\notin$w alors w-k(u)=h'(m',v)
où h'=h-$\{k\}$ et m'=ul,...,u(i-1),u(i+1),...,un
avec $\Pi[K](h')$=kl...k(i-1)k(i+1)...kn,
sinon indéfini.

L'insertion de la communication k(u) dans le terme w est notée w+k(u) et elle est définie :
si k(u)$\notin$w alors w+k(u)=h'(m',v)
où h'=h$\cup\{k\}$ et m'=ul,...,u(i-1),u,ui,...,un
avec $\Pi[K](h')$=kl...k(i-1) k ki...kn
sinon indéfini.

2.4.2.2. Union de processus

Soient P1=<Σ1,I1,E1> et P2=<Σ2,I2,E2> deux processus avec Σ1=K1$\cup$G1 et Σ2=K2$\cup$G2. K1 et K2 sont disjoints. Les ensembles de noms de variables de E1 et E2 sont aussi disjoints.

P = P1$\|$P2 = <Σ,I,E>, où

Σ = Σ1.Σ2

I = I1.I2

E = E1.Z[G2] $\cup$ E2.Z[G1] $\cup$ E1.E2

P est un nouveau processus qui se comporte : soit comme P1, soit comme P2, soit comme P1 et P2 simultanément.

2.4.2.3 Connexion de processus

Une communication entre M et N est seulement possible quand les communications à travers M.? et ?.N sont simultanément possibles.

Soient P1=(Σ1,I1,E1) une présentation de processus avec Σ1=K1$\cup$G1. Soient M et N deux noms de portes telles que M.?$\in$K1,

?.N $\in$ K1 et $\eta(M.?) = \eta(?.N)$. La connexion de M à N dans P1, notée P1+M.N est un processus P, qui peut se comporte comme le processus représenté par P1 et qui permet en plus, la communication le long M.N. Le résultat de cette opération est le suivant :

P = P1 + M.N = $<\Sigma, I, E>$, où

Σ = K U G, avec K=K1 U $\{M.N\}$ et G=G1

I = I1

E = E U E', où

E' = $\{ <\rho(w-M.?(t)-?.N(u)+M.N(t)), \rho(v)> |$
$<w,v> \in E1$, M.?(t) $\in$ w, ?.N(u) $\in$ w et ρ=pgu(t,u)$\}$.

2.4.2.4 Abstraction de processus

Soit P1=$<\Sigma 1, I1, E1>$ une présentation de processus avec Σ1=K1UG1. Soit k un nom de connecteur. "S'abstraire" dans P1 de k, P1-k, revient à obtenir un nouveau processus qui se comporte comme le processus P1 sauf que :

si k $\in$ Kext, alors les communications au long de k sont, maintenant impossibles.

si k $\in$ Kint, alors les communications au long de k sont, maintenant invisibles

Le résultat de cette opération est :

P = P1-k = (Σ, I, E), où

Σ = K U G avec K=K1-$\{k\}$ et G=G1

I = I1

E = (E1-E') U E'', où

E' = $\{<w,v> | <w,v> \in E1$ et il existe u, k(u) $\in$ w$\}$

E''= $\{<w-k(u),v> | <w,v> \in E1$ et k$\in$Kint et k(u)$\in$w$\}$

2.4.3. Généricité.

Les présentations de processus peuvent être génériques :

des types peuvent être spécifiés comme paramètres formels et les arités des connecteurs et des symboles d'état peuvent être définis en termes de ces types formels. Mais, les processus peuvent être génériques dans d'autres dimensions : noms de portes comme paramètres et aussi des paramètres valeur.

Regardons un exemple :
Une pile P de capacité n (paramètre-valeur), qui stocke des objets du type T (type formel) et qui a comme noms de portes formels : I (porte d'entrée) et O (porte de sortie) :

```
P = (T : ftype).     (paramètres de type)
    (I : porte, O : porte)
    (n : Nat)
    cnx   ?.I,O.? : Nat
    etats S : Seq Nat
    init  S(Λ,n)
    vars  s : Seq, m : Nat
    regles
          S(s,succ(m))    => ?.I(v) S(cons(v,s),m)
          S(cons(v,s),m) => O.?(v) S(s,succ(m))
    fin
```

2.5. SPARC par rapport à d'autres projets. Conclusion.

Le thème du parallélisme et des processus communicants bénéficie actuellement d'une exceptionnelle attention de la part des chercheurs. De nombreaux travaux ont été faits à ce sujet comme on a vu dans le chapitre 1, dont deux ont suscité beaucoup d'intérêt: CSP et CCS.

Le projet SPARC, par ses objectifs, se place au même niveau, en répondant aux problèmes posés au moyen d'un modèle pour la communication qui a des fondements algébriques. Nous exposerons ici certaines de ses caractéristiques les plus

importantes en soulignant les différences par rapport aux autres systèmes.

- Les processus en SPARC sont des entités communicantes capables aussi de réaliser des calculs de fonctions. Au moyen d'un processus SPARC on peut exprimer toutes les fonctions calculables, comme le montre <Per 84>.

- L'algèbre de SPARC possède trois opérateurs de base :
  ‖ : le parallélisme
  + : la connexion
  - : l'abstraction
qui sont "les outils" pour la manipulation de processus, c'est à dire, pour la construction de réseaux. Le formalisme à travers lequel ils ont été définis nous permet d'obtenir la plupart des opérateurs proposés par les autres systèmes du même style, en particulier de CCS, voir <Jor 84b>.

- En SPARC, l'opérateur de mise en parallèle (P=P1 ‖ P2) est interprété de la façon suivante :
P est un processus qui exprime toutes les possibilités de comportement en ce qui concerne P1 et P2 évoluant en parallèle, c'est à dire : P1 seul, P2 seul et P1 et P2 simultanés. Le processus P permet l'observation des événements simultanés à la différence de CSP ou CCS.

- Le mécanisme de synchronisation de communications en SPARC est le "rendez vous" comme en CSP et CCS. Mais une communication aura lieu quand elle a été rendue possible par l'opérateur de connexion +. Quand dans deux processus travaillant en parallèle (P), une des possibilités est l'ocurrence d'une communication de sortie O.? et une communication d'entrée ?.I simultanées, on peut décider que la communication devient possible en construisant P+O.I. Cela constitue une différence par rapport à CSP ou CCS où la connexion est établie en même temps que la mise

en parallèle.

- La définition de nos trois opérateurs permet de voir toujours quel est le processus résultant d'une expression. Cela est dû au fait que les calculs en SPARC sont des manipulations syntaxiques à partir de descriptions de processus et le résultat est aussi une description de processus, celle qui correspond à la spécification du réseau construit. Ainsi, SPARC offre une visibilité des résultats du calcul, ce qui n'est pas le cas dans la plupart des autres systèmes.

- Le fait d'avoir des événements simultanés (à différence de CCS) et de ne pas avoir de restrictions sur les variables dans les entrées et les sorties (elles peuvent être les mêmes, à la différence de CSP qui est impératif) nous permet de décrire un processus en une seule règle :

Q => ?.I(x)O.?(x-1)?.I1(f(x-1))O2.?(x*f(x-1)) Q

où ?.I, ?.I1 sont les portes d'entrée du processus

et O.?, O1.? sont les portes de sortie du processus.

Si l'on imagine un réseau de n+1 processus de cette sorte, connectés ayant tous cette même règle, on obtient le calcul instantané de factorielle de x.

- Pour finir, SPARC est intégré dans un langage applicatif, FP2, au moyen duquel on peut exprimer toutes les notions du système. FP2 est à SPARC ce qu'OCCAM est à CSP. Il n'est encore que partiellement implémenté.

3. LPG : LANGAGE DE PROGRAMMATION GENERIQUE

3.1. Présentation Générale du Langage

Le langage LPG est un maillon d'un système de spécification et de programation.

Maillon d'un système de programation, car on peut programmer des algorithmes exécutables sur un ordinateur. A ce propos, il existe une version de LPG <Ber 83> qui a été implantée au LIFIA.

Maillon d'un système de spécification, puisqu'on peut décrire des problèmes dont les solutions pourront être ensuite programées en LPG. Le formalisme de spécification et de programmation étant le même, cela facilite l'apprentissage du langage et simplifie la définition des conditions de preuve.

C'est un langage applicatif, c'est à dire qu'il ne possède que les notions d'expressions, d'abstraction (opérateurs) et d'application (appel).

LPG est un langage fortement typé et modulaire. Les déclarations de types se font d'une manière abstraite, par un ensemble d'opérateurs et non par une représentation.

On a aussii la possibilité de définir des fonctions sur les types, la sémantique de tout opérateur étant algébrique et décrite par des axiomes équationnels.

L'une de ses caractéristiques la plus importante est la généricité. Ainsi, les unités de définition de types et de fonctions peuvent être paramétrées de façon telle qu'avec une seule définition, on peut exprimer des familles d'objets ayant la même structure.

LPG est un langage conversationnel. Il permet de définir et de conserver des environnements de programation composés de types et de fonctions.

LPG dispose d'un mécanisme de contrôle pour rompre l'évaluation

normale des expressions. Il s'agit des exceptions qui peuvent être déclenchées et qui activent un traitement (ou récupérateur) de l'exception.

3.1.1. Définition d'un type abstrait

Un type abstrait est un type défini par l'ensemble des opérations qui s'appliquent aux objets de ce type. L'intérêt de cete "abstraction" est que l'on n'a pas à connaître une représentation particulière pour pouvoir déclarer et utiliser ce type. "Un objet est caractérisé par ses fonctions et non par sa représentation".

Dans LPG, un type abstrait est défini suivant une méthode "algébrique". Pour spécifier un type on donnera donc, le nom du type, l'ensemble d'opérateurs et un ensemble d'axiomes ou équations.

Opérateurs

Chaque opérateur sera décrit par son nom, le type de ses paramètres (domaine) et le type de son résultat (codomaine). Les constantes sont des opérateurs qui n'ont pas de domaine.

Parmi les opérateurs, il y en aura certains que l'on appelle "constructeurs", et qui devront être précédés de l'attribut cons. Les opérateurs constructeurs sont l'ensemble minimal d'opérateurs avec lesquels on peut obtenir tous les éléments du domaine. Il y a un autre attribut : le mot privé qui, mis devant un opérateur, indique que celui ci ne peut pas être vu de l'extérieur.
Un opérateur peut être une exception. Il est alors déclaré avec le codomaine "exc".

Equations

Les équations permettent d'exprimer la sémantique des opérateurs. Une équation est une égalité de deux termes construits à l'aide des opérateurs et de variables typées. Les variables sont des identificateurs qui doivent être déclarées dans une rubrique "vars". Elles sont implicitement quantifiées universellement pour chaque équation.

Les équations, syntaxiquement sont différenciées en : "générales" et "exécutables". La distinction permet simplement d'exécuter des programes LPG, car les équations exécutables définissent les opérateurs de manière opérationnelle. Ce sont la plupart des équations dans les types et les fonctions.
Exemple d'équation générale :

```
\\ no.  x+y==y+x
```

Exemple d'équation exécutable :

```
\ no. fact(x)==si x=0 alors 1
                    sinon x*fact(x-1)
              fsi
```

Dans les exemples qui suivent, nous omettrons cette distinction.

Voyons un exemple de type abstrait défini en LPG. Le type Boolean, avec ses deux valeurs constantes : vrai et faux, et deux opérateurs : ¬(la negation) et ∧(le "et").

```
Type Bool
    opns
        cons vrai : -> Bool
        cons faux : -> Bool
             ¬ : Bool -> Bool
             ∧ : (Bool,Bool) -> Bool
    vars b : Bool
```

```
    eqns
        ¬(vrai) == faux
        ¬(faux) == vrai
        b ∧ vrai == b
        b ∧ faux == faux
fin
```

3.1.2. Définition de fonction (enrichissement)

Lorsqu'on a défini des types abstraits avec leurs opérations primitives, il est souvent nécessaire, pour une application particulière, de déclarer des nouveaux opérateurs sur les types existants, sans pour cela déclarer des nouveaux types. La déclaration de ces nouveaux opérateurs se fait à l'intérieur d'une définition appelée "enrichissement".

Un enrichissement ("enrich") comprend le nom de l'enrichissement, un ensemble d'opérateurs et un ensemble d'axiomes ou équations.

Exemple d'enrichissement :

```
enrich or-bool
    opns ∨ : (Bool, Bool) -> Bool
    vars b1, b2 : Bool
    eqns
        b1 ∨ b2 == ¬((¬b1) ∧ (¬b2))
fin
```

3.1.3. Définition de Propriété

La notion de propriété est à la base de la généricité en LPG. Comme on l'a déjà dit, la généricité consiste à paramétrer les définitions. Ainsi, un type peut être générique, c'est à dire, paramétré par d'autres types appelés types formels. Les

opérateurs sont génériques de la même manière que les types. Les opérateurs associés aux types formels sont des opérateurs formels qui peuvent être utilisés dans les équations des opérateurs génériques.

Un type formel est caractérisé par un ensemble minimun d'opérateurs que devra posseder le type effectif. .Ces ensembles minimun peuvent être identifiés sous le nom de propriétés. Ainsi, une propriété est ce que l'on exige d'un type effectif pour pouvoir être paramètre d'une certaine classe d'objets.

La structure d'une définition de propriété est : nom de la propriété éventuellement suivi de types formels, un ensemble d'opérateurs, et les équations qui décrivent la sémantique.

Il est souvent nécessaire de caractériser les propriétés les unes par rapport aux autres ; par exemple un monoide "contient" la propriété d'associativité, ou d'une manière équivalente, la propriété d'associativité est "incluse" dans la propriété monoide. Ce sont ces relations que l'on appelle inclusion de propriétés. L'ensemble des propriétés d'une spécification forme ce que l'on appelle un "graphe" de propriétés.

En LPG il y a deux clauses qui permettent de relier les propriétés entre elles :

1) La clause "satisfait"
La clause satisfait P1 écrite après les équations d'une propriété P2 signifie qu'il existe un sous-ensemble de types formels et d'opérateurs de P2 qui vérifient (satisfont) l'axiomatisation de P1.
Regardons certains exemples de propriétés :
prop tyfo sur t
fin

tyfo est la propriété plus générale. Elle est satisfaite par tous les types.

```
prop égalité sur t
  opns eq : (t,t) -> Bool
  vars x,y,z : t
  eqns
      eq(x,x) == vrai                                    (reflexivité)
      eq(x,y) == eq(y,x)                                 (simétrie)
      eq(x,y) ∧ eq(y,z)∧ ¬(eq(x,z)) == faux              (transitivité)
  satisfait tyfo[t]
fin

prop ordre-partiel sur tl
  opns inf-eg, égal : (tl,tl) -> Bool
  vars u,v,w : tl
  eqns
      inf-eg(u,u) == vrai                                (reflexivité)
      égal(u,v)   == inf-eg(u,v) ∧ inf-eq(v,u)           (antisymetrie)
      inf-eg(u,v) ∧ inf-eg(v,w)∧ ¬ inf-eg(u,w) == faux
                                                         (transitivité)
  satisfait égalite [tl opns égal]
fin
```

La propriété ordre-partiel satisfait l'égalité : le type tl avec l'opérateur égal, vérifient les trois règles définies dans l'égalité. Ainsi :

tyfo ⊆ égalité ⊆ ordre-partiel

2) La clause "hérite".

La clause herite P1 écrite après les équations d'une propriété P2, indique que P2 possède, outre ses propres équations, toutes les équations de P1, via la substitution d'opérateurs et de types indiquée dans la clause. Par exemple, si l'on a :

prop associativité sur tl

```
  opns + : tl,tl -> tl
  vars x,y,z : tl
  eqns
      x+(y+z) == (x+y)+z
  satisfait tyfo [tl]
fin
```

Alors la propriété monoide se définit comme :

```
prop monoide sur t
  opns 0 : -> t
       + : (t,t) -> t
  vars x : t
  eqns
       0+x == x
       x+0 == x
  herite associativité [t opns +]
fin
```

3.1.4. Unités Génériques

Une unité générique est une définition de propriété, de type, ou d'enrichissement, paramétrée par une propriété déjà définie. La propriété paramètre s'appelle la propriété exige ; les types et opérateurs de cette propriété exigée sont les types et opérateurs formels de l'unité générique.

Exemple : le type générique ensemble.

```
type ens exige égalité [elem opns =]
  opns cons e-vide : -> ens
       cons ins    : (elem, ens [elem]) -> ens [elem]
  vars x1,x2 : elem, y : ens [elem]
  eqns ins(x1,ins(x2,y)) == ins(x2,ins(x1,y))
       ins(x1,ins(x2,y)) == si x1=x2 alors ins(x2,y) fsi
fin
```

Supposons que l'on veut ajouter maintenant la possibilité de :

1) Tester la cardinalité d'un ensemble
2) Tester l'appartenance d'un élément à un ensemble
3) Savoir si un ensemble est vide, et
4) Avoir l'opération "enlever".

Nous écrivons l'enrichissement "op-sur-ens" :

```
enrich op-sur-ens exige égalite [elem opns eq]
opns      #  : ens[elem] -> Entier
          ∈  : (elem,ens[elem]) -> Bool
      vide? : ens[elem] -> Bool
          -  : (ens[elem],elem) -> ens[elem]
vars a,b : elem, s : ens[elem]
eqns #(e-vide : ens[elem])   == 0
     #( ins(a,s) )           == 1 +#(-(s,a) )
     a ∈ e-vide : ens[elem]  == faux
     a ∈ ( ins(b,s) )        == si eq(a,b) alors vrai
                                sinon a ∈ s
                                fsi

     - ( ins(b,s),a )        == si eq(a,b) alors s-a
                                sinon ins(b,-(s,a))
                                fsi

     - (e-vide : ens[elem],a)== e-vide : ens[elem]
     vide?(e-vide:ens[elem]) == vrai
     vide?(ins(a,s))         == faux
fin
```

Remarque. Il existe en LPG la qualification explicite des opérateurs instanciés. Cela veut dire que si le type des paramètres d'un opérateur ne suffit pas à déterminer les types effectifs d'un modèle de propriété exigée (c'est le cas particulier de toutes les constantes génériques), on doit qualifier l'instance de cet opérateur par le type du résultat.

C'est pour cela que l'on a écrit dans l'enrichissement op-sur-ens

e-vide : ens [elem]

3.1.5. Conclusion

Il est intéressant de voir la différence parmi ces trois définitions : type, enrich et propriété. Si l'on regarde la syntaxe, ces trois unités de définition sont toutes des ensembles d'opérateurs regroupés sous un nom et dont la sémantique est exprimée à travers des équations. Cependant, ces modules d'opérateurs décrivent des choses très différentes les unes des autres.

Les types : l'ensemble d'opérateurs définit une classe d'objets se comportant de la même façon.

Les propriétés : l'ensemble d'opérateurs de la propriété indique qu'un ou plusieurs types (les types formels) peuvent se manipuler d'une certaine façon. Les propriétés caractérisent des familles de types.

Les Fonctions ou enrichissements : l'ensemble d'opérateurs dans ce cas, sert à étendre la puissance (si l'on veut entendre par puissance, la quantité des fonctions applicables à un objet) d'un ou plusieurs types.

On peut considérer les objets d'un type comme des entités physiques (en négligeant leur représentation) se comportant de la même façon. L'enrichissement "enrichit" la puissance de différentes classes d'objets. Les propriétés décrivent le potentiel minimal exigé d'un certain type.

3.2. Description algébrique des types et propriétés : syntaxe et sémantique.

LPG est un langage dont les définitions sont décrites à travers un formalisme algébrique.

La description algébrique d'un objet consiste à spécifier, d'une part la syntaxe, en donnant les noms des opérateurs avec leurs domaines et codomaines, et d'autre part la sémantique de ces opérateurs à travers un ensemble d'équations. Cette unité opérateurs-équations, sous un nom spécifique, constitue la définition d'un objet.

Dans la première partie de ce chapitre, nous avons fait un exposé de certains concepts dont nous nous servons maintenant pour faire la correspondace entre les unités de LPG et ces définitions algébriques.

Syntaxe

- Une définition de type abstrait est une présentation, c'est à dire, une paire $<\Sigma,E>$ où Σ est la signature du type (l'ensemble d'opérateurs avec sa fonction arité) et E est un ensemble d'équations.

- Une définition d'enrichissement est une présentation, donc, une paire $<\Sigma,E>$ où Σ est la signature de l'enrichissement et E est l'ensemble d'équations qui décrivent la sémantique des opérateurs de Σ.

- Une définition de propriété est aussi une présentation $<\Sigma,E>$ où Σ est la signature de la propriété et E est l'ensemble d'équations.

Syntaxiquement, il n'y a pas de différence entre type, enrichissement et propriété : tous les trois sont des présentations. On avait fait la remarque en 3.1.5. C'est du point de vue de leur sémantique que ces trois concepts deviennent différents.

Sémantique

- Un type abstrait T=(Σ,E) est l'algèbre de termes W_Σ (initiale et libre) qui satisfait la présentation <Σ,E> ou toute autre algèbre isomorphe à W_Σ. A partir de cet ensemble d'algèbres isomorphes, on obtient la fermeture de E, $\overline{E}$. Alors la paire <Σ,$\overline{E}$> ainsi obtenue, constitue la théorie équationnelle de T.

- Une propriété P=<Σ,E> définit une catégorie d'algèbres Alg dont les objets sont les Σ-algèbres qui satisfont E, et dont les morphismes sont des homomorphismes d'algèbres. La fermeture de E,$\overline{E}$, peut s'obtenir automatiquement en appliquant un système logique aux équations de E, ce qui nous donne la théorie équationnelle <Σ,$\overline{E}$> de P.
  Nous avons déjà vu que les propriétés peuvent être reliées entre elles par la relation d'inclusion. C'est à travers des clauses "satisfait" et "herite" que cette relation est établie.
  Soient P1=<S1,Ω1,E1> et P2=<S2,Ω2,E2> des présentations de théories de propriétés. Si P2 satisfait (herite) P1, alors il existe un morphisme de théorie β:T_{P1}->T_{P2} composé de :

  1. Un morphisme de signature σ=<f,g> où
     f : S1->S2 est une fonction qui détermine la correspondance entre les sortes de P1 et celles de P2.
     g : Ω1->Ω2 est une fonction qui assigne aux opérateurs de P1 les opérateurs correspondants de P2.

2. Une condition sémantique

$\sigma(E1) \subseteq \overline{E2}$ où $\forall e \in E1$, $E2 \vdash \sigma(e)$

$\sigma(e)$ étant défini $\forall x \in S1 \wedge \forall w \in \Omega 1$

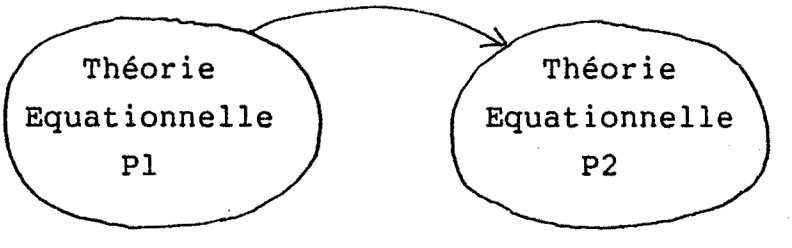
$\sigma(e)$ = e, où on a substitué toute s par f(s)
et tout w par g(w)

Exemple :

```
prop P1 sur t1,t2
  op   f1,f2
  vars x1:t1, x2:t2
  eqns f1(x1,x2) == f2(x2,x1)   }
                    :           } E1
fin

prop P2 sur u1,u2,u3
  op   g1,g2,g3,g4
  vars .....
  eqns    :                     }
          :                     } E2
  satisfait p1[u2,u3 opns g1,g4]
fin
```

β : Morphisme de Théorie



$\beta : \sigma$ = <f,g>

f = t1 -> u2
t2 -> u3

g = f1 -> g1
f2 -> g4

$\sigma(E1) \subseteq \overline{E2}$: $g1(x1,x2)=g4(x2,x1) \subseteq E2$ où $x1 \in u2$

$x2 \in u3$

- Une unité générique est une présentation $U=<\Sigma,E>$. Les unités génériques sont paramétrés par des types pour lesquels on exige certains propriétés, et comme les propriétés, elles définissent une catégorie, la catégorie des modèles de la théorie $U=<\Sigma,\overline{E}>$.

CHAPITRE III

FP2: UN LANGAGE POUR LES PROCESSUS COMMUNICANTS

CHAPITRE III

FP2: UN LANGAGE POUR LES PROCESSUS COMMUNICANTS

Il existe beaucoup de systèmes qui peuvent être modélisés en termes d'unités interconnectées qui communiquent des informations les unes aux autres : le matériel et logiciel d'un ordinateur, les réseaux de télécommunications, les réseaux de bureautique, etc. Avant d'envoyer une information (ou avant d'exécuter une action sur l'environnement), chaque unité doit la calculer commme fonction d'une information reçue au préalable. Les différentes unités peuvent traiter des informations indépendamment les unes des autres et elles peuvent se comporter en parallèle.

Cependant très peu de langages ont été conçus en regroupant les notions de communication, fonction et parallélisme comme concepts de base pour la construction de systèmes. L'architecture de Von Neuman continue à influencer la conception de langages malgré les propositions d'autres principes comme la programmation fonctionnelle de Backus <Bac 78> et le calcul de systèmes communicants de Milner <Mil 80>.

Nous présentons le langage FP2 (Functional Parallel Programming), langage dans le quel les notions de communication, fonctions et parallélisme sont coherentes : sa structure est simple et sa sémantique formelle est rigoureusement définie. Cela est dû à l'utilisation systématique de termes et substitutions de termes pour représenter les notions essentielles du langage.

FP2 étend l'élégance sémantique des langages applicatifs <Tur 81> à la communication, au non déterminisme et au parallélisme. Pour cette raison, ce langage fournit une base

cohérente pour la programmation, la spécification, la conception, la vérification, l'analyse et la synthèse de presque tous les systèmes discrets.

1. INTEGRATION DE LPG A FP2

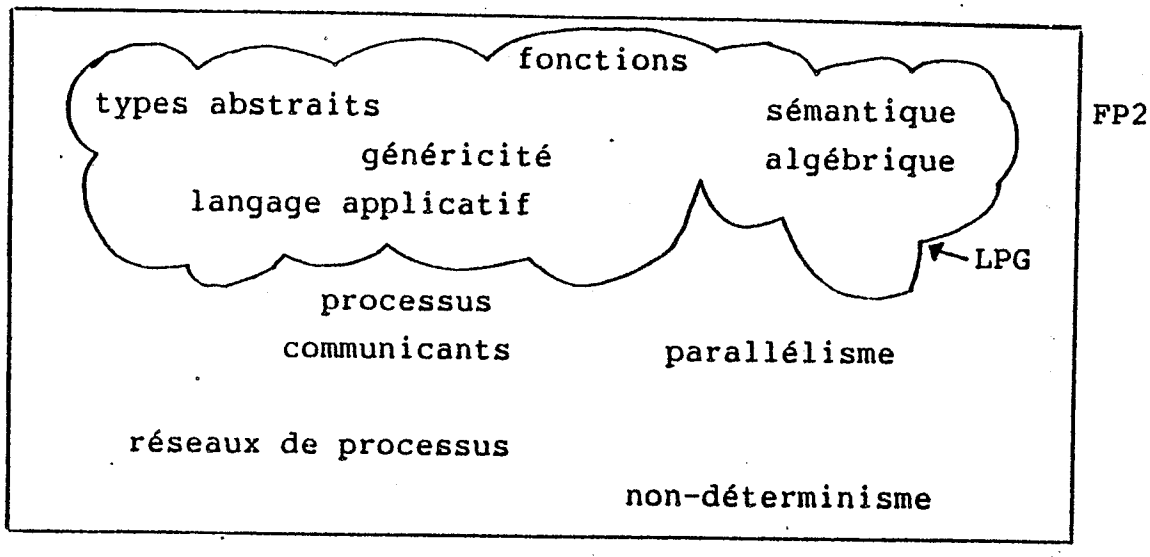
FP2 est le langage proposé pour traduire toutes les idées présentées par le projet SPARC. Ainsi, FP2 se fixe parmi d'autres objectifs, celui d'offrir la possibilité de définition de types abstraits, de fonctions et de processus communicants. On souhaite aussi avoir la généricité, c'est à dire, pouvoir paramètrer ces unités de définition par des types.

Le travail concernant les types abstraits et les fonctions, ainsi que la réalisation de la généricité à travers des propriétés, est dû à LPG, Langage de Programmation Générique <Ber 83>. Des études préalables à LPG sur les types abstraits et la généricité (Propriétés) ont été présentés dans <Jac 78> et <Ber 79>.

Nous trouvons en LPG la mise en oeuvre d'un sous ensemble de nos buts, le tout étant présenté d'une façon tout à fait compatible avec notre propos :

- langage applicatif
- définition de types et fonctions
- généricité
- sémantique algébrique.

C'est pour cela que l'on a pu, d'une façon totalement naturelle, intégrer LPG à notre langage FP2.



En ce qui concerne la sémantique formelle, FP2 et LPG sont parfaitement compatibles. Les définitions des trois entités qui nous intéressent : types, fonctions et processus, sont des présentations (voir II.1) et leur sémantique est décrite par l'algèbre de termes satisfaisant la présentation.

Les propriétés étant aussi des présentations, leur sémantique est décrite par la catégorie des algèbres qui satisfont la présentation. FP2 se sert de cette notion de propriété, de la même façon que LPG, pour obtenir la généricité des processus communicants. Ainsi, un processus générique a la même interprétation algébrique que les unités génériques de LPG (types, fonctions) : ce sont des présentations qui définissent des catégories d'algèbres.

Dans le paragraphe 2.3 de ce chapitre, nous présenterons la carte syntaxique de FP2, où il apparaît l'intégration de LPG : on laisse intactes les définitions de types, fonctions et propriétés et, pour la définition des processus, on fait plusieurs appels aux productions de la grammaire de LPG. Voir (a), (b) dans la Carte Syntaxique.

2. PRESENTATION DU LANGAGE FP2

2.1. Généralités

Dans le premier chapitre on a parlé des avantages des langages applicatifs. On a vu aussi l'importance actuelle de langages pour la communication. Dans le deuxième chapitre nous avons exposé l'interêt d'avoir des spécifications formelles avant la progammation et en particulier, on a présenté la technique de spécification algébrique. Ensuite, les systèmes SPARC et LPG nous montrent une concrétisation des notions des réseaux communicants, parallélisme, etc. (SPARC) et des types abstraits, fonctions, généricité (LPG).

Le langage FP2 (Functional Parallel Programming) est proposé ici commme le langage du système SPARC qui englobe toutes ces caractéristiques qui répondent à des questions actuellement importantes en informatique.

FP2 est un langage qui permet de composer et de structurer des spécifications :

- de processus communicants, fonctions réalisées et comportement ;
- de réseaux de processus, à travers des expressions construites avec les opérateurs de processus : Union (Parallélisme), Connexion et Abstraction plus un opérateur conditionnel si, alors, sinon, au sens habituel ;
- de types abstraits ;
- de fonctions.

Les processus, types et fonctions sont définis d'une manière abstraite et algébrique : par un ensemble d'opérateurs (portes et symboles d'état pour les processus) et non par une représentation. La sémantique des opérateurs est décrite par des axiomes équationnels.

FP2 est un langage qui autorise un grand usage de la généricité, c'est à dire de la paramétrisation. Les processus, types et fonctions peuvent être paramétrés par des types appelés types formels. Un type formel est caractérisé par un ensemble minimum d'opérateurs que devra posséder tout type effectif. Ces ensembles minimum peuvent être identifiés sous le nom de propriétés. Une propriété est donc définie par l'ensemble des opérateurs se rapportant à des types formels.

FP2 est un langage applicatif qui permet de décrire la communication, ce qui le rend élégant, actuel et puissant.

2.2. Définition du Langage

FP2 permet la spécification de processus, types, fonctions (enrichissements) et propriétés. Les trois dernières unités sont décrites en LPG et c'est pour cette raison que nous nous limiterons dans ce paragraphe à la définition des processus. (1)

Quand nous avons parlé de SPARC, nous avons vu que la construction d'un réseau de processus se fait en appliquant les opérateurs de processus : ‖(union), +(connexion) et -(abstraction). On obtient ainsi des expressions dont le résultat, lors de manipulations syntaxiques, est aussi une description de processus semblable à celle d'un processus simple.

Une définition de processus ((1)DEF-DE-PROCESSUS) est,

---

(1) A partir de maintenant, les explications seront accompagnées de la référence à la production de la grammaire correspondante au paragraphe 2.3.

soit une déclaration de processus simple, soit une expression de processus. Toute définition de processus doit être précédée du mot processus suivi du nom du processus à définir, et peut être paramétrée par diverses sortes d'objets comme on le montre ci dessous (2.2.1).

processus nom-de-processus {paramètres}
{déclarations}

ou bien,

processus nom-de-processus = {paramètres} {expression}

2.2.1. Généricité de processus

Les processus, ainsi que les types et les fonctions de LPG, peuvent être génériques, c'est à dire, paramétrés. Nous distinguerons quatre sortes de paramètres de processus ((2)PAR-FORM).

Paramètres-Types ((3)TYPE-FORM).
On peut paramétrer les processus par des types formels. Cela se fait à travers des propriétés, de la même façon que LPG :

processus P exige prop [tf1,...,tfn opns opf1,...,opfn]
(tf1,...,tfn : types formels ; opf1,...,opfn : opérateurs formels)

Le processus P recevra comme paramètres des types effectifs tel,...ten correspondants aux types formels et que vérifient la propriété "prop" avec les opérateurs opel,...,open correspondants aux opérateurs formels.
Par exemple, si l'on a comme paramètre formel

processus P exige monoide [T1 opns +,0]

le processus P pourra travailler avec
[Nat opns +,0], [Nat opns .,1], [Bool opns or,faux].

Paramètres-fonction ((4)FONC-FORM).
On permet de passer des noms de fonctions comme paramètre. La

fonction formelle sera accompagnée de son domaine et son range et précédé du mot "lambda-f".

processus P lambda-f nom-fonct : (d1,...,dn) -> dn1
fonction formelle — domaine — codomaine

Par exemple dans la déclaration :

processus P lambda-f h : Nat -> Nat

toutes les fi telles que fi : Nat -> Nat pourront être fonctions effectives en P.

Paramètres-valeur ((5)VAL-FORM).

Le passage des valeurs comme paramètres d'un processus est permis. Comme pour les fonctions, au moment de la déclaration du paramètre il faut spécifier son type et le préceder du mot "lambda-v".

processus P lambda-v x : T

Exemple :

processus P lambda-v x,y : Nat, b : Bool

Paramètres-porte ((6)PORT-FORM).

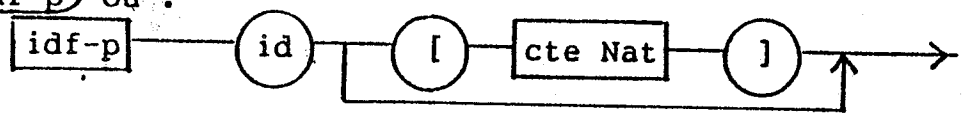
Les noms de porte peuvent être passés commme paramètre aussi. Il suffit de faire précéder les noms de porte formels du mot "portes".

processus P portes A1,...,An

Une syntaxe permise pour les noms de portes est :

identificateur [cte. Naturelle]

C'est une façon d'indexer les noms des portes ; cela nous sera utile au moment de faire des constructions récursives, pour que chaque appel de processus ait des noms de portes différents. Dans la carte syntaxique, les noms de porte sont définis come (idf-p) où :



Chaque sorte de paramètre formel est donc, précédé d'un

mot clé qui l'identifie : exige, lambda-f, lambda-v et portes ; et doit finir par un point.

La possibilité de rendre un processus si fortement générique, confère à notre système une puissance d'expression pour la construction de réseaux. En effet, en jouant avec les paramètres portes et valeurs et nos quatre opérateurs de processus, on peut réaliser des constructions récursives.

Voyons par exemple la construction du BUFFER de capacité n à partir du BUFFER de capacité 1.

Soit BUFF la spécification d'un buffer simple qui peut contenir une valeur d'un type quelconque. La propriété exigée de ce type formel est la plus petite propriété : tyfo, elle est vérifiée par tous les types et elle est sous ensemble de toutes les autres propriétés.

```
processus BUFF exige tyfo [tl]. portes I,O.
  cnx    ?.I,O.? : tl
  etats  Q : (), Q1 : tl
  init   Q
  vars   x : t
  regles
         Q     => ?.I(x) Q1(x)
         Q1(x) => O.?(x) Q
fin
```

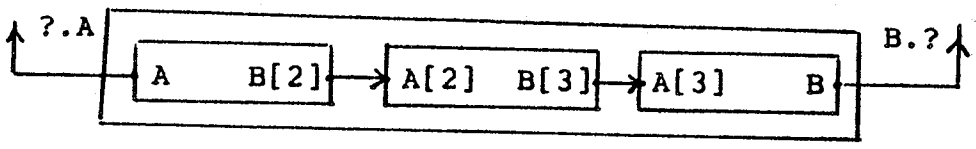
Nous allons construire l'expression qui donne pour résultat un buffer de n cases, BUFF-N :

```
processus BUFF-N = exige tyfo[tl].
                   lambda-v n. portes I,O.
        si n=1 alors
                   BUFF[tl](I,O)
               sinon
                   BUFF-N[tl](n-1)(I,O[n]) ||
```

BUFF[tl](I[n],O) ⊕ O[n].I[n]

fsi

Par exemple, si on instancie BUFF-N[Nat](3)(A,B), ce qu'on obtient est :



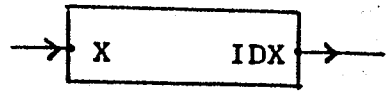
Buffer de Naturels, 3 cases, portes A,B

Les paramètres fonctions sont aussi d'une grande utilité. Supposons par exemple, un processus qui fait le calcul d'une fonction quelconque, définie pour un objet d'un type quelconque, et qui rend une valeur dans ce même type. Ce processus est FONCT-GEN :

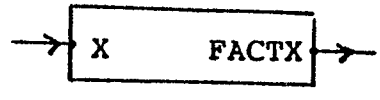
```
processus FONCT-GEN exige tyfo[tl]. lambda-f h:tl->tl.
                    portes X,HX.
  cnx    ?.X:tl, HX.?:tl
  etats P:()
  init  P
  vars  y:tl
  regles
        P => ?.X(y) HX.?(h(y)) P
fin
```

Des instanciations possibles sont :

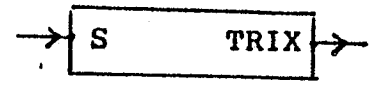
FONCT-GEN[Nat](id)(X,IDX)



FONCT-GEN[Nat](fact)(X,FACTX)



FONCT-GEN[Seq[Nat]](tri)(S,TRIX)



Dans le paragraphe 3, on trouvera d'autres exemples qui illustrent la puissance et l'intérêt pratique de FP2.

2.2.2. Spécification d'un processus

La spécification d'un processus commence donc, par le mot "processus", suivi du nom de processus et éventuellement de la liste de paramètres formels. Ensuite, on commence la description du processus, c'est à dire, le corps de déclarations ((7)DECLARATIONS) composé de : connecteurs, états, états initiaux, variables et règles. La spécification d'un processus doit finir par le mot "fin".

Toute spécification de processus doit avoir au moins un état initial et, en consequence, au moins un symbole d'état. Par contre, les ensembles de connecteurs, de variables et de règles peuvent être vides. Les mots identificateurs des différentes rubriques doivent toujours apparaître.

Un processus qui n'a ni des connecteurs, ni des variables, ni des règles, est le processus NEUTRE pour l'opérateur union :

processus NEUTRE
  cnx
  etats $\{Qi : (\{Tj\})\}$, non vide, $Tj \in$ Types
  init $\{Qi : \{(\{pj \mid pj \in Tj \wedge pj \text{ sans variables }\})\}$, non vide
  vars
  regles
fin

Pour tout $P \in$ Processus, $P \| NEUTRE = NEUTRE \| P = P$
avec renomage des termes d'état.

Déclaration de connecteurs ((8)CCNX)
La déclaration de connecteurs commence avec le mot cnx suivi de

la liste de connecteurs du processus, chacun étant accompagné de son type, séparés par des virgules. On peut regrouper les connecteurs ayant le même type. Exemple :

cnx ?.A,B.? : T1, ?.I,?.O : T2, C.? : T3 où T1,T2,T3 $\in$ Types

Il peut exister des processus sans portes ; dans ce cas, le mot cnx ne sera pas suivi de connecteurs. Un processus sans connecteurs est une entité qui ne communique pas avec l'environnement : les évènements de ses règles, s'il y en a, sont tous l'évènement nul, $\mathcal{Y}$.

Etats ((13)CETATS).

Ici, on énumère les symboles d'état qui apparaîtront dans les règles, accompagnés par les types de leurs arguments. On peut regrouper les symboles d'état ayant la même liste de types. L'identification de la rubrique est le mot "etats". Exemple :

etats Q,R : (), P : Nat Bool, T,S : Seq[Nat]...

Etat initial ((15)CINIT).

Ici, on décrit le, ou les états à partir desquels commencera le comportement du processus. Dans les arguments de l'état initial, il ne peut pas apparaître des variables. La règle dont la partie gauche unifie avec l'état initial, sera la première à être appliquée. S'il y en a plusieurs, le démarrage sera non déterministe. Ces états initiaux, sont précédés du mot "init". Exemple :

init P(0) Q(λ,vrai)

Variables ((18)CVARS).

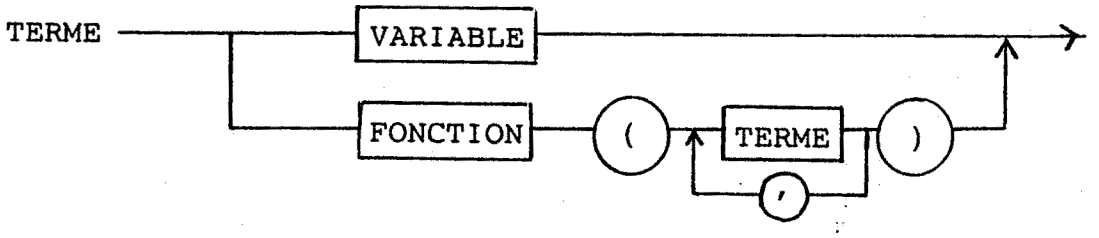
La déclaration de variables est semblable à celle de LPG.

Règles ((19)CREGLES).

La description de l'ensemble de règles est précédée du mot "regles", suivi de la spécification de chacune des règles qui consiste en : une partie gauche (16)ETAT, une flêche =>, l'évènement, et une partie droite (16)ETAT.

L'événement peut être nul, pour les transitions internes, ou bien, un ensemble de communications, c'est à dire, de connecteurs accompagnés de l'expression ((17)TERME) à recevoir ou envoyer.

Terme ((17)TERME). Un terme, dans l'état actuel de FP2, a la forme :



Cependant, dans la définition de terme de notre carte syntaxique, on fait appel à la production EXPR de LPG. Ceci laisse ouverte la possibilité d'utiliser dans le futur, les autres formes permises en LPG. Exemple de termes :

```
    regles
        Q(s,b) => ?.A(succ(x)) B.?(x+s) Q(cons(s,x),faux)
termes  ↑↑       ↑             ↑         ↑
```

Instanciation ((22)INSTANCIATION).
L'instanciation d'un processus, se fait en appelant le processus avec les paramètres effectifs correspondants aux paramètres formels de la spécification ((23)TYPE-EFF, (24)FONC-EFF, (25)VAL-EFF, (26)PORT-EFF).

2.2.3. Réseaux de processus : Expressions ((21)EXPRESSION).

Une expression de processus a pour résultat un processus dont les connecteurs, états, règles, etc. sont calculés pour chaque opérateur. (Voir paragraphes II.2.3 et II.2.4.2).

Une expression est, soit l'instanciation d'un processus déjà défini, soit une phrase dont les opérandes sont des

instanciations ou des connecteurs, et dont les opérateurs sont : ‖(union), +(connexion), -(abstraction), ⊕(connexion-abstraction) et si alors sinon. Les trois premiers ont été déjà définis.

L'opérateur ⊕ a été implémenté pour simplifier l'écriture. Il effectue d'abord la connexion interne, puis, il fait l'abstraction du connecteur interne et des deux connecteurs externes concernés.

⊕ : processus x connecteur interne -> processus

Soit P un processus et A.B un connecteur interne :

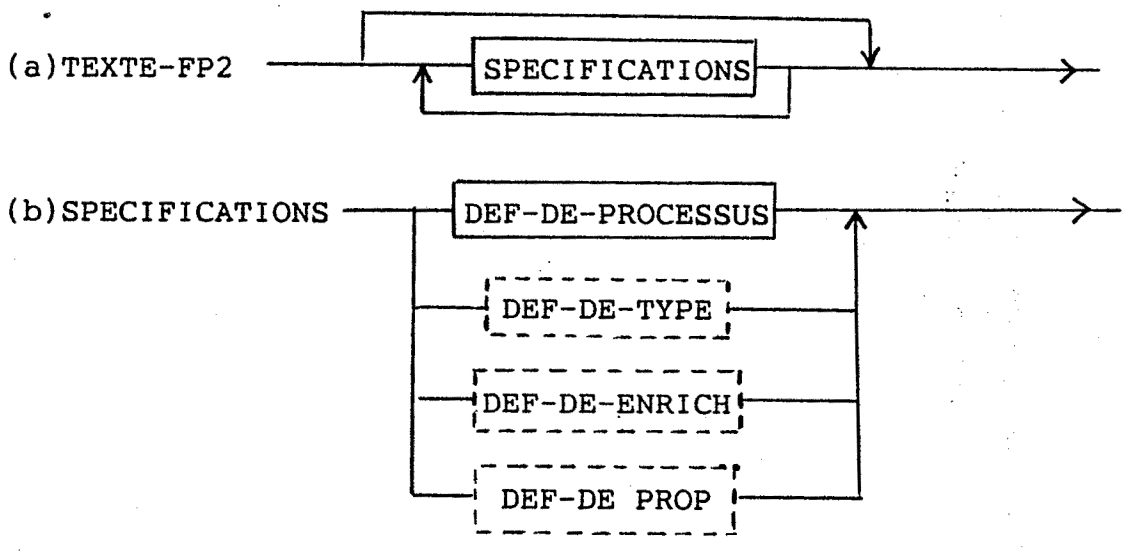
P ⊕ A.B = P+A.B-A.B-A.?-?.B

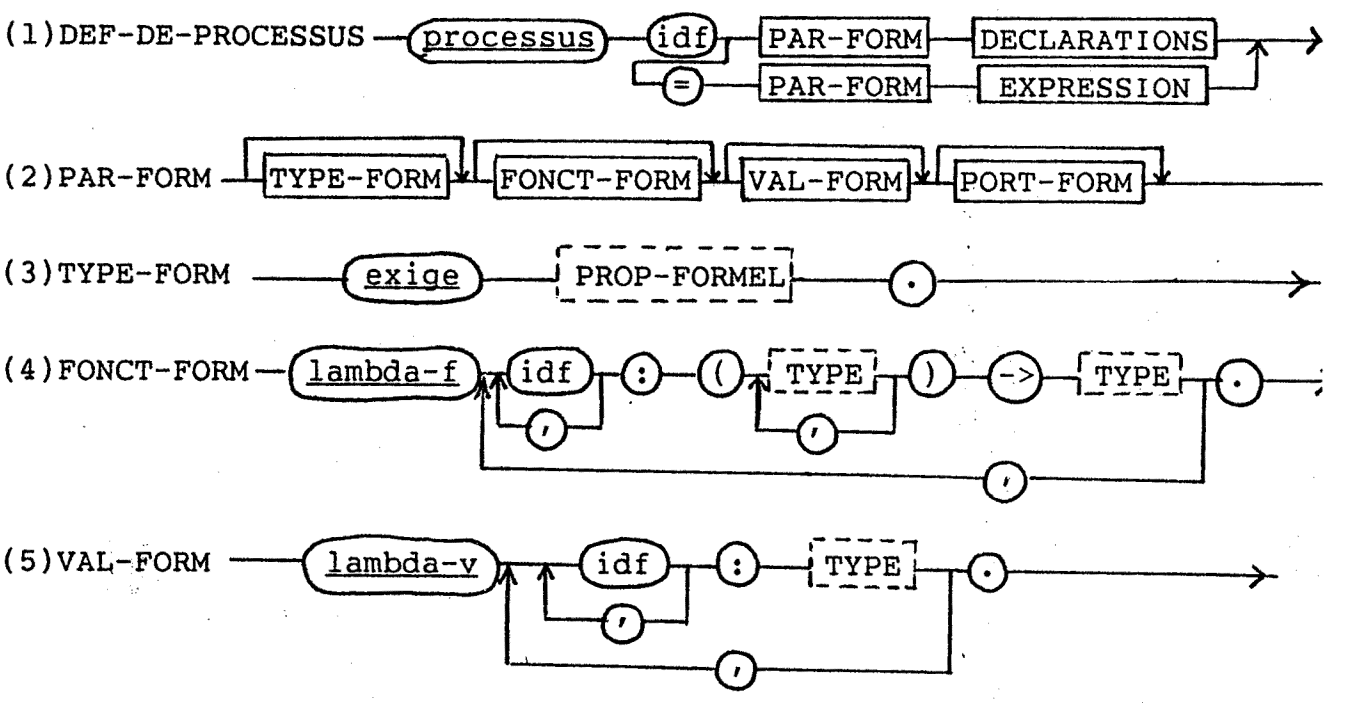
En effet, dans la construction des réseaux, la démarche qui consiste à faire une connexion, puis à "cacher" après tous les connecteurs externes et internes concernés par la connexion, est une démarche très courante.

Le conditionnel si alors sinon a sa signification habituelle. Cet opérateur nous permet de faire des constructions récursives comme celle de BUFF-N à partir de BUFF (parag. 2.2.1).

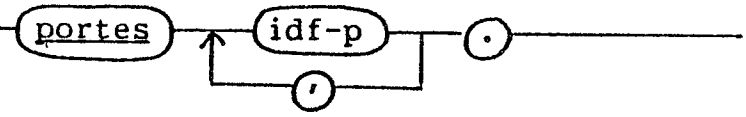
2.3. Carte Syntaxique de FP2.

Dans la carte syntaxique présentée ici, les non terminaux « » encadrés par des lignes pointillées appartiennent à la grammaire de LPG (voir annexe 1). Comme on a déjà dit au paragraphe 1, FP2 fait appel aux définitions de LPG.



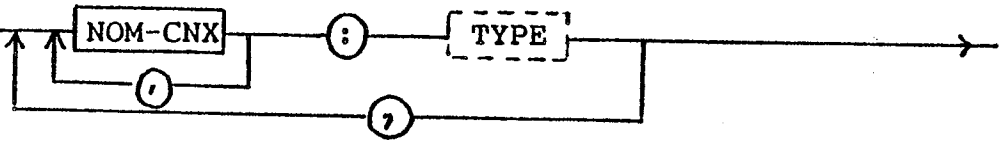


(6)PORT-FORM

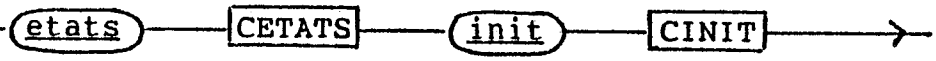


(7)DECLARATIONS

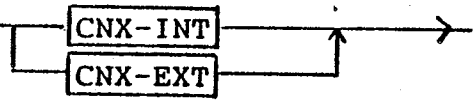
(8)CCNX



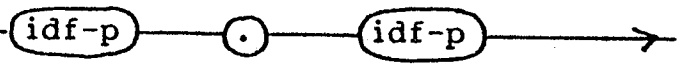
(9)ETINI



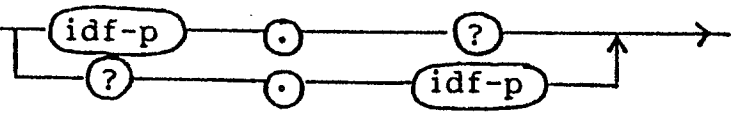
(10)NOM-CNX



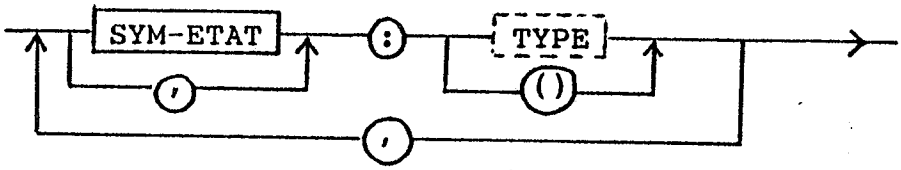
(11)CNX-INT



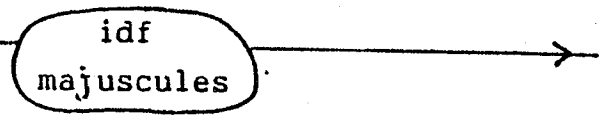
(12)CNX-EXT



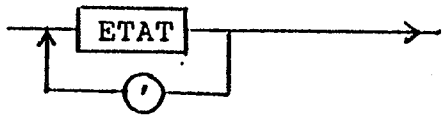
(13)CETATS



(14)SYM-ETAT

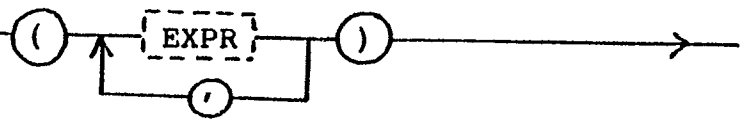


(15)CINIT

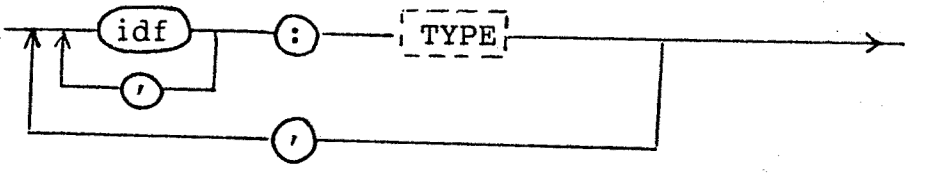


(16)ETAT

(17)TERME

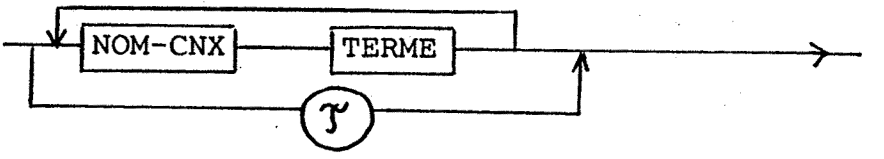


(18)CVARS

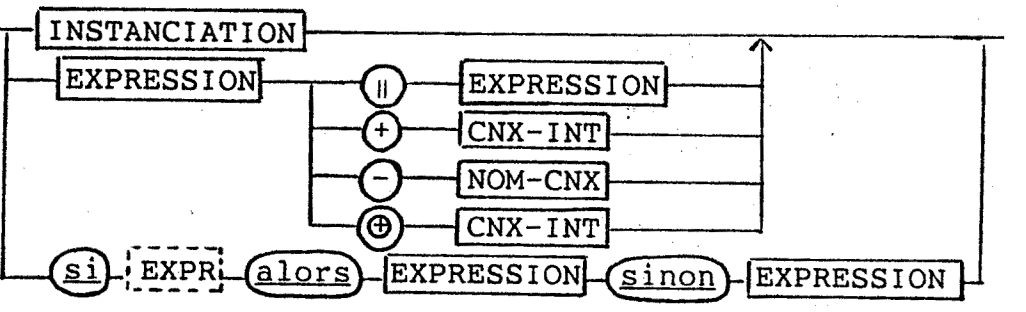


(19)CREGLES

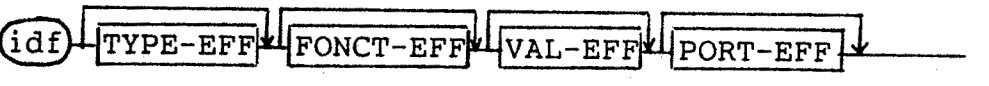
(20)EVENEMENT



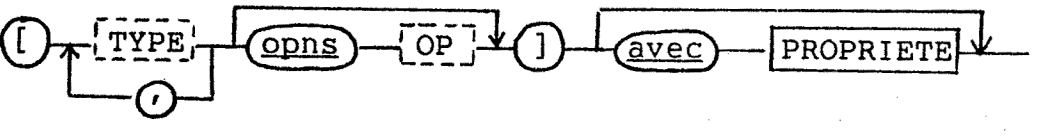
(21)EXPRESSION



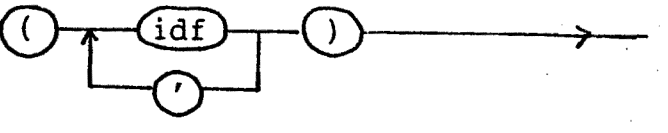
(22)INSTANCIATION



(23)TYPE-EFF

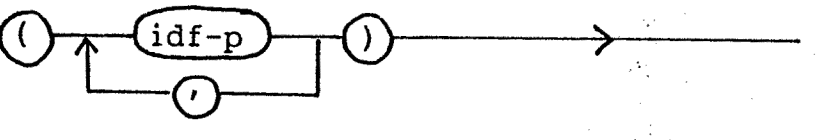


(24)FONCT-EFF



(25)VAL-EFF

(26)PORT-EFF



3. EXEMPLES

Nous avons choisi quelques exemples pour montrer la programmation en FP2. Les types et fonctions utilisées sont définis en LPG ; voir <Ber 83>, page 41. Le premier exemple montre la possibilité, au moment de décrire un processus, de calculer des fonctions par des règles de transitions internes. Le deuxième, fait un grand usage de la généricité en composant différentes instanciations d'un même processus. Le troisième est un exemple classique : le partage d'une ressource ; et le dernier exemple appartient à des travaux faits dans le domaine matériel et en algorithmique parallèle : réseaux systoliques.

Ces deux derniers exemples, sont des cas pratiques où interviennent les concepts de communication et parallélisme. Pour les programmer nous nous servirons de toutes les ressources offertes par FP2 : types abstraits, fonctions, généricité, construction récursive, non déterminisme, transitions internes, etc. Nous traiterons des problèmes comme la concurrence et la synchronisation.

3.1. Exemple No.1: FONCTION MULTIPLICATION

Les processus peuvent appeler des fonctions qui sont définies sur les types des objets manipulés, mais ils peuvent aussi les calculer eux mêmes, en utilisant uniquement les constructeurs du type en question.

Nous traiterons la fonction multiplication définie sur les Naturels : mult : (Nat, Nat)->Nat. Une possibilité est de définir multip comme un enrichissement des Naturels et l'appeler ensuite :

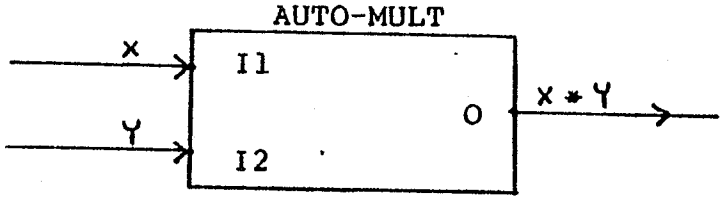
```
enrich multip exige monoide [t opns mult,1]
  opns mult : (Nat, Nat) -> Nat
```

```
  vars  x,y : Nat
  eqns mult(x,y) == si or(x=0,y=0) alors 0
                                   sinon mult(add(x,y),y-1)
fin
processus MULTIPLICATION
  cnx  ?.I1, ?.I2, O.? : Nat
  etats Q : ()
  init Q
  vars x,y : Nat
  regles
       Q => I1(x)?.I2(y)O.?(mult(x,y)) Q
fin
```

Le processus MULTIPLICATION fait appel à l'opérateur mult que l'on avait défini dans l'enrichissement multip.

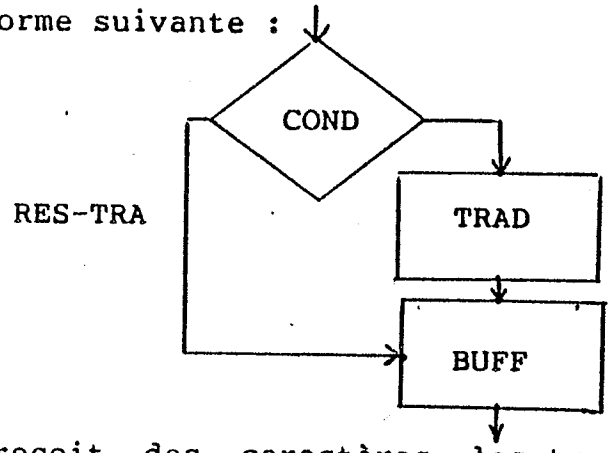
Mais regardons maintenant le processus AUTO-MULT qui calcule lui même la fonction multiplication sans faire appel à aucun opérateur des naturels, exceptés les constructeurs. Si l'on n'avait pas eu LPG, on aurait programmé le processus AUTO-MULT de la façon suivante :

```
processus AUTO-MULT
cnx   ?.I1,?.I2, O.? : Nat
etats Q:(), Q1,Q2:Nat Nat Nat Nat
init  Q
vars  x,y,u,v: Nat
regles
  Q                        => ?.I1(x)?.I2(y) Q1(x,x,x,y)
  Q1(x,x,x,0)              => O.?(0)         Q
  Q1(x,x,x,succ(y))        => 𝒯              Q2(x,x,x,y)
  Q2(x,succ(u),v,succ(y))=> 𝒯                Q2(succ(x),u,v,succ(y))
  Q2(x,0,v,succ(y))        => 𝒯              Q2(x,v,v,y)
  Q2(x,u,v,0)              => O.?(x)         Q
fin
```



3.2. Exemple No.2: TRADUCTEUR GENERIQUE

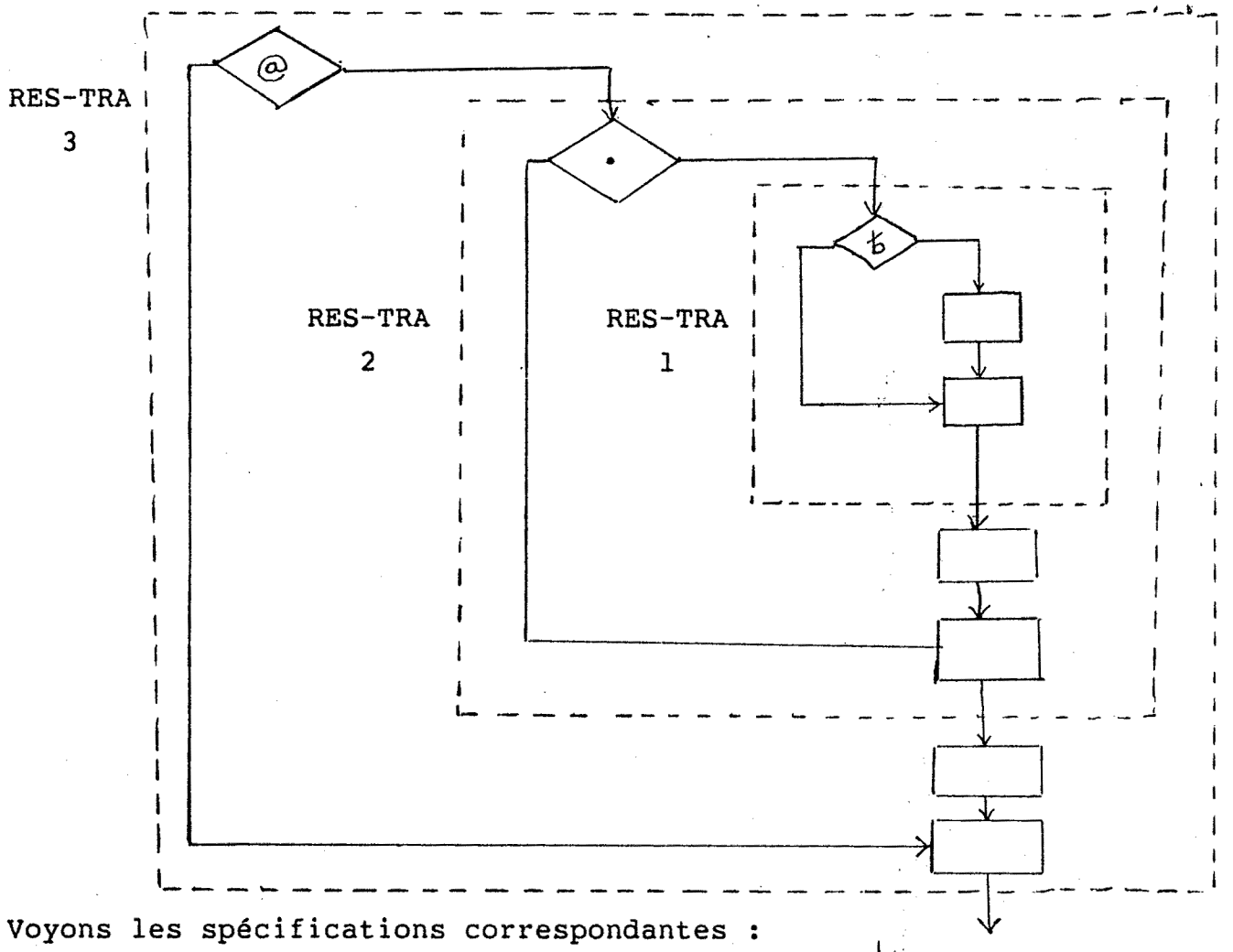
Avec la possibilité de généricité, nous décrivons trois unités de base pour un réseau traducteur. La première est COND, un processus qui détecte la fin de texte et avertit l'unité où est rangé le résultat qu'il faut sortir le texte traduit. La deuxième, TRAD, applique une fonction sur l'entrée et rend le résultat comme sortie : c'est le processus qui traduit. La troisième, BUFF, enregistre des unités traduites jusqu'à recevoir le signal de fin de texte envoyé par COND. Ce réseau a la forme suivante :



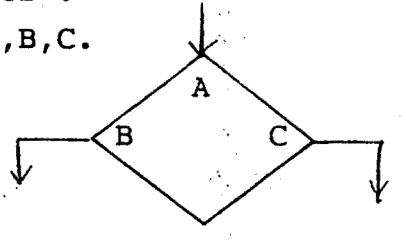
Il reçoit des caractères, les traduit et les accumule en BUFF jusqu'à former le mot (les mots sont séparés par des blancs). Il traduit les mots caractère par caractère.

Mais on peut imaginer, avec la même entrée des caractères une traduction mot par mot dont la sortie est une phrase (les phrases sont séparées par un point). Et aussi une traduction phrase par phrase dont la sortie est un paragraphe (les paragraphes sont séparés par @). Toutes ces possibilités sont

realisables à partir du même réseau RES-TRA, en composant différentes instanciations de ce processus.



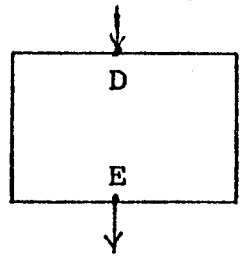
Voyons les spécifications correspondantes :




```
processus COND lambda-v c:Car. portes A,B,C.
  cnx    ?.A,C.? : Car
         B.? : Signal
  etats  Q:(), Q1:Car Bool
  init   Q
  vars   x : Car
  regles
         Q           => ?.A(x)  Q1(x,x=c)
         Q1(x,vrai)  => C.?(x)  Q
         Q1(x,faux)  => B.?(zz) Q
fin
```

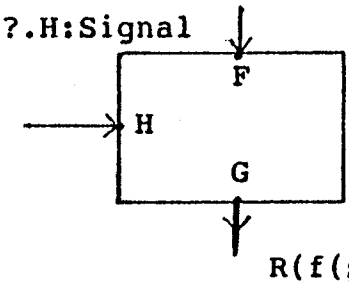
Le processus COND a un paramètre C, qui sera le caractère à détecter, ce qui conduit COND à envoyer un signal zz.

```
processus TRAD exige tyfo[t]. lambda-f h:t->t. portes D,E.
  cxn   ?.D,E.? : t
  etats P:()
  init  P
  vars  x : t
  regles
        P => ?.D(x) E.?(h(x)) P
fin
```



TRAD reçoit un type formel t (soit Caractère, soit Chaine), utilise une fonction formelle h, la fonction de traduction ou l'identité selon le paramètre effectif qui convient.

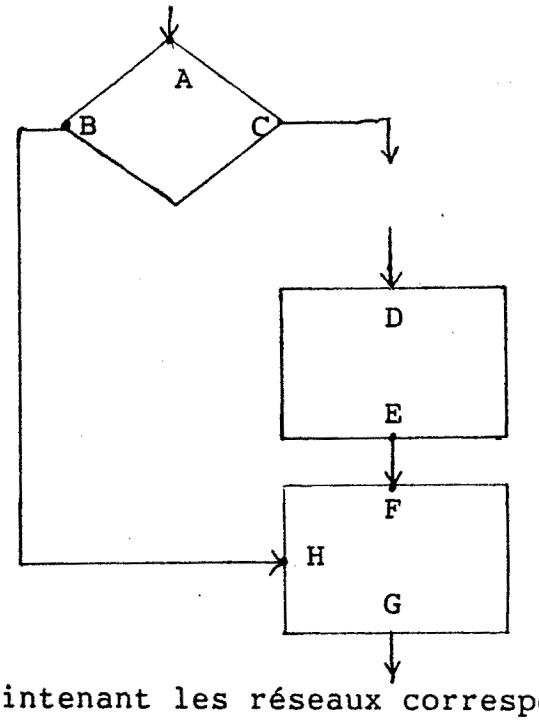
```
processus BUFF exige tyfo[t]. lambda-f f:(Chaine,t)->Chaine.
                lambda-v b:Bool. portes F,G,H.
  cnx   ?.F:t, G.?:Chaine, ?.H:Signal
  etats R:Chaine Bool
  init  R(vide,b)
  vars  s:Chaine, x:t
  regles
        R(s,b)    => ?.F(x)                  R(f(s,x),b)
        R(s,vrai) => ?.H(zz)G.?(+>(s,b))     R(vide,vrai)
        R(s,faux) => ?.H(zz)G.?(s)           R(vide,faux)
fin
```



Le processus BUFF reçoit des valeurs du type t (soit caractère, soit chaine), utilise une fonction formelle f qui concatène soit chaine-caractère (+>), soit chaine-chaine (+). En plus, il a un paramètre-valeur b pour distinguer quand il s'agit de caractères pour former des mots et ajouter alors le caractère blanc à la fin, ou bien quand il s'agit de chaines (phrases ou paragraphes), cas où il ne faut rien ajouter.

On construit alors, le réseau de base en laissant ouverte la connexion de COND avec TRAD pour pouvoir "introduire" à cette place un autre appel du même réseau dans le cas où cela est nécessaire.

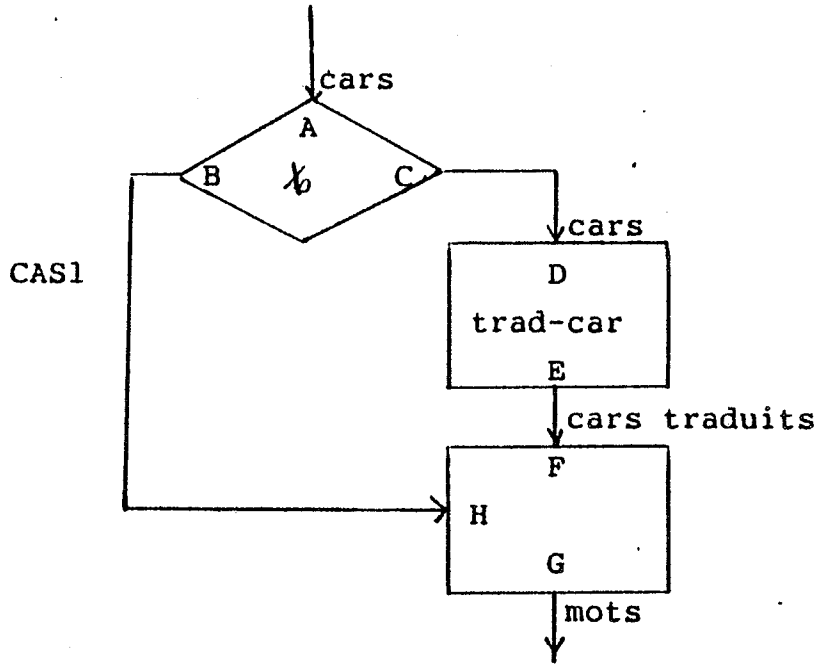
processus RES-BAS = exige tyfo[t].
lambda-f h:t->t,f:(Chaine,t)->Chaine.
lambda-v c:Car, b:Bool.
portes A,B,C,D,E,F,G,H.
COND(c)(A,B,C) ‖ TRAD[t](h)(D,E)‖
BUFF[t](f)(b)(F,G,H) ⊕ E.F ⊕ B.H



Considérons maintenant les réseaux correspondant aux trois versions de notre exemple :

CAS1 : Machine qui reçoit une séquence de caractères, et qui sort des mots "traduits", dont la traduction a consisté à traduire caractère par caractère. Les mots sont séparés par des blancs.

processus CAS1 = RES-BAS[Car](trad-car,+>)(b̸,vrai)⊕C.D



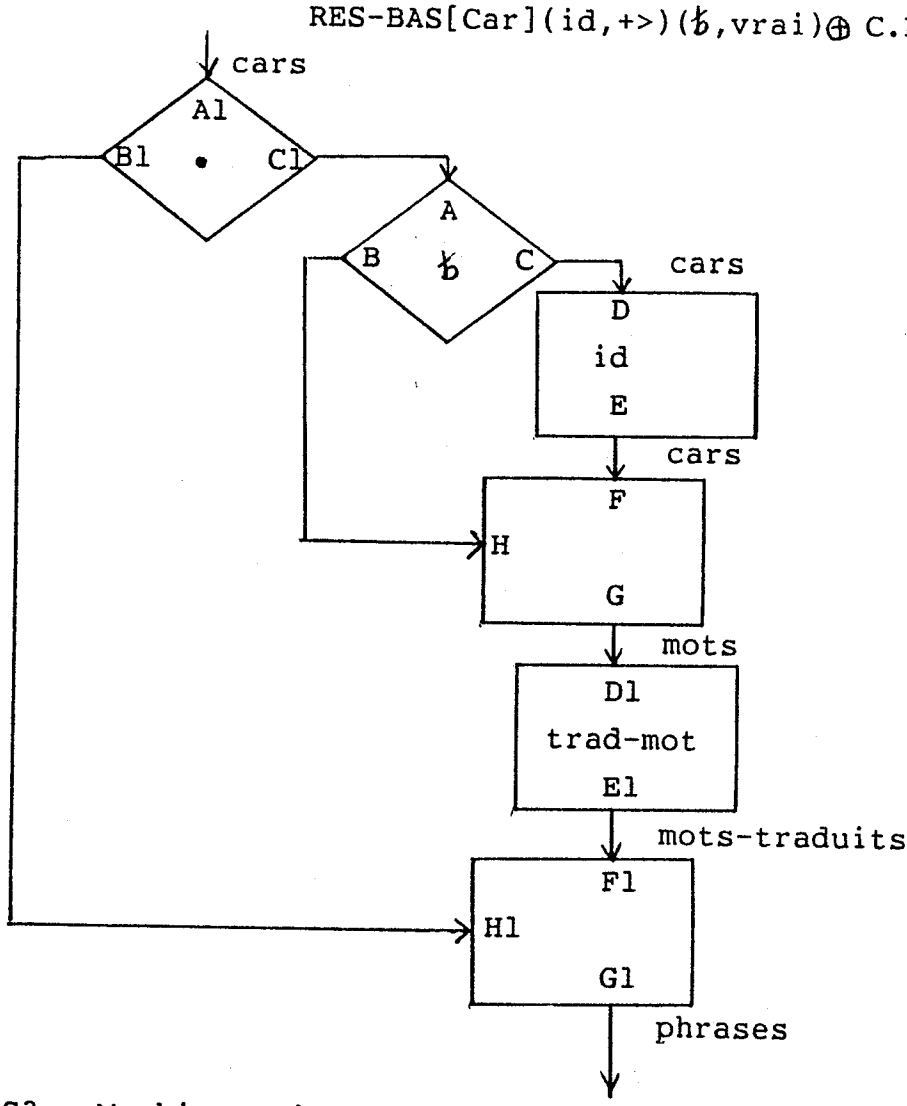
Le réseau CAS1 après le calcul, a la forme suivante :

```
processus CAS1
  cnx    ?.A:Car, G.?:Chaine
  etats  QPR:Chaine Bool, Q1PR:Car Bool Chaine Bool
  init   QPR(vide,vrai)
  vars   x:Car,s:Chaine,b:Bool
  regles
    QPR(s,vrai)          => ?.A(x)          Q1PR(x,x=ϵ,s,vrai)
    Q1PR(x,vrai,s,vrai)=> G.?(+>(s,ϵ))  QPR(vide,vrai)
    Q1PR(x,vrai,s,faux)=> G.?(s)         QPR(vide,faux)
    Q1PR(x,faux,s,faux)=> ϒ              QPR(<+(s,trad-car(x)),vrai)
fin
```

Ce résultat ser à montrer que même si le calcul de l'opérateur union produit un grand nombre de règles, le nombre de règles du réseau calculé est proportionnel à la solution du problème. En plus, dans l'implémentation présentée en <Ark 84>, on montre que dans l'évaluation d'une expression de processus ce n'est pas nécessaire d'écrire les règles susceptibles à disparaître à cause de l'opérateur d'abstraction. Cette évaluation faite en <Ark 84> ressemble à "l'évaluation paresseuse", dans le sens que l'on ne calcule que les choses nécessaires.

CAS2 : Machine qui reçoit une séquence de caractères, qui forme des mots, qui traduit ces mots et qui fait sortir des phrases traduites. Les phrases sont séparés par ".".

processus CAS2 = RES-BAS[Chaine](trad-mot,+)(.,faux)
(A1,B1,C1,D1,E1,F1,G1,H1) ||
RES-BAS[Car](id,+>)(ƀ,vrai)⊕ C.D⊕C1.A⊕ G.D1

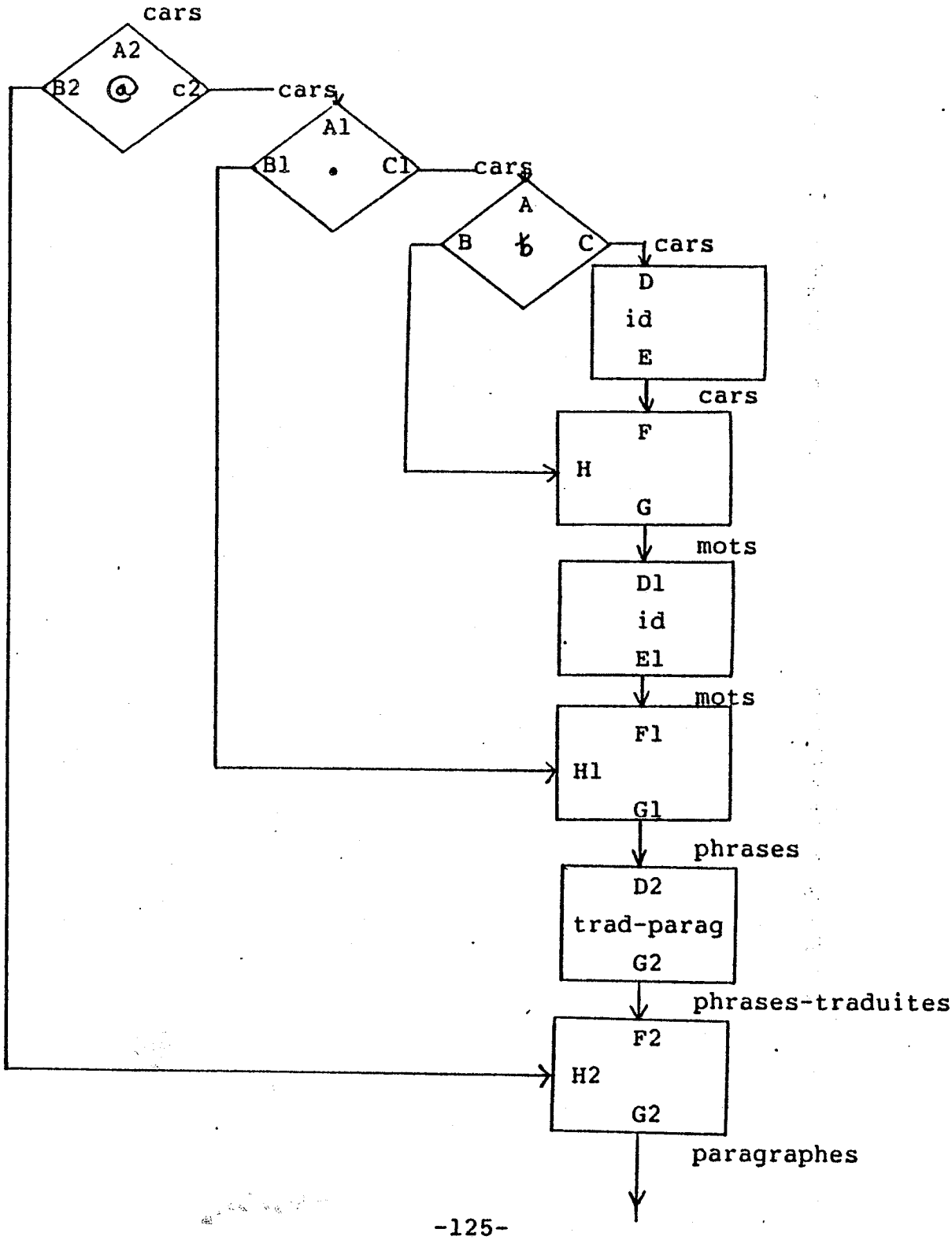


CAS3 : Machine qui reçoit une séquence de caractères, forme des mots (séparés par ƀ), forme des phrases (séparés par .), traduit chaque phrase et sort des paragraphes (séparés par @).

processus CAS3 = RES-BAS[Chaine](id,+)(.,faux)
(A1,B1,C1,D1,E1,F1,G1,H1) ||
RES-BAS[Car](id,+>)(ƀ,vrai) ||

RES-BAS[Chaine](trad-phrase,+)(@,faux)
(A2,B2,C2,D2,E2,F2,G2,H2)
⊕ C.D ⊕ C1.A ⊕ G.D1 ⊕ C2.A1 ⊕ G1.D2

Graphiquement on a obtenu :



3.3. Exemple No.3: PARTAGE D'UNE RESSOURCE

Il s'agit d'un ensemble de processus qui sont concurrents sur une même ressource : ils sont tous connectés à une même voie, dans notre cas "porte", qui ne permet qu'une réception à la fois. La nécessité d'établir une politique d'utilisation de la voie à partager est évident.

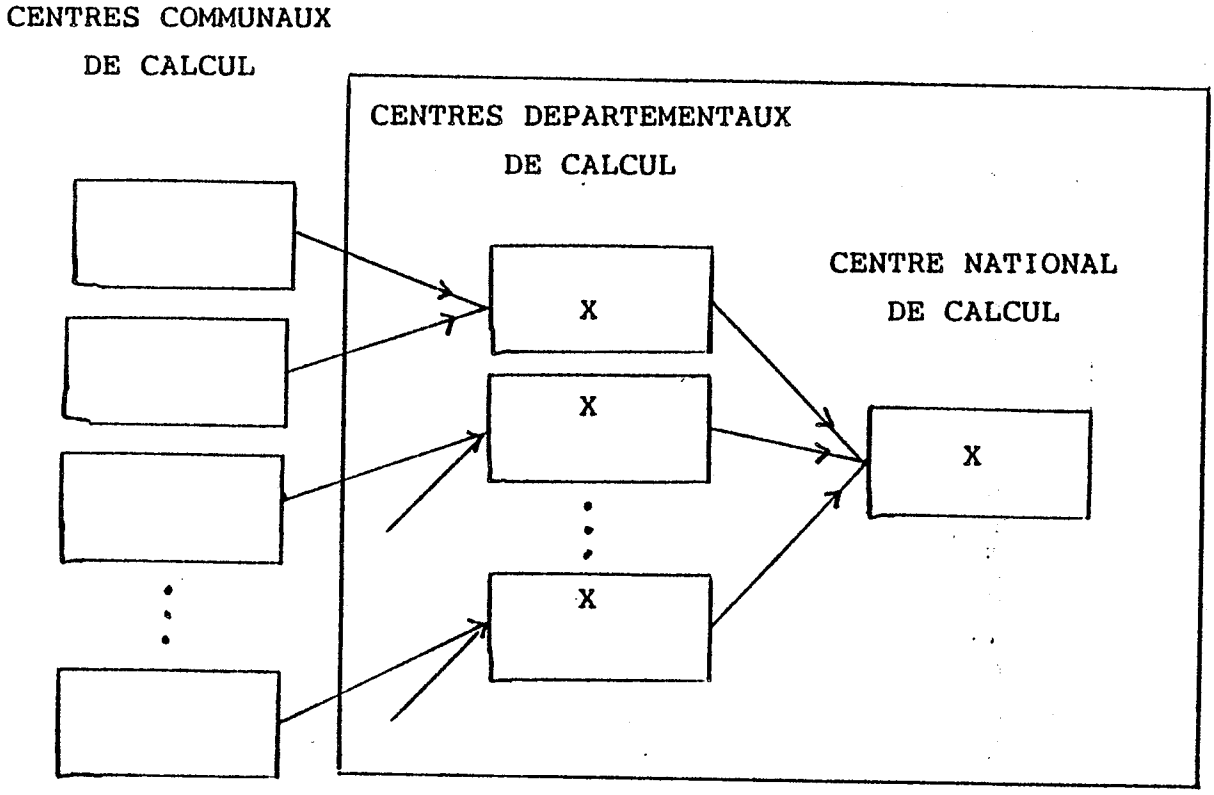
On a choisi la méthode de "partage par élection" <COR 81> et plus concrètement la "consultation" ou "polling". Cette politique consiste à interroger tous les émetteurs dans un ordre fixe. Lors de la rencontre d'un émetteur prêt, celui ci est élu et la consultation s'arrête. Une fois le message transmis la consultation recommence.

Cette méthode sera implantée de façon centralisée, c'est à dire, qu'il y aura un "arbitre" qui procédera à l'élection. La consultation que notre arbitre va faire sera un parcours circulaire de la liste des concurrents. La méthode sera ainsi équitable : aucun émetteur ne sera privé indéfiniment de la voie.

L'arbitre fera un octroi séquentiel et circulaire de la permission de transmission, en commençant la première fois par le premier processus de la liste et les autres fois, par celui qui suit le dernier processus qui a transmis.

Nous proposons un cas pratique simple pour illustrer ce problème celui d'un réseau hiérarchisé de centres de calcul.

Nous nous intéresserons seulement au premier niveau :



Les X représentent l'accumulation de votes, et c'est le centre de calcul national qui totalise pour produire le résultat global. Les centres de calcul departementaux seront donc, les processus concurrents CDC et le centre national CNC sera la ressource.

La spécification de chacun d'eux dans notre langage est la suivante :

```
processus CNC portes ECR, LIB, LIR
  cnx   ?.ECR, LIR.? : Nat,  LIB.? : Signal
  etats R : Nat
  init  R(0)
  vars  y,x : Nat
  regles
        R(y) => LIB.?(zz) ?.ECR(x) R(y+x)
        R(y) => LIR.?(y)           R(y)
fin
```

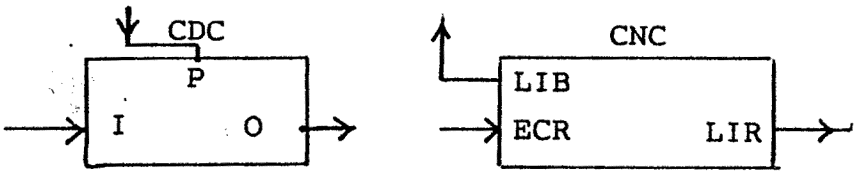
La fonction du processus CNC est d'actualiser une donnée (y) en faisant l'addition de tout ce qu'il reçoit par ECR. Il a aussi une porte de sortie LIR par laquelle l'extérieur peut consulter le scrutin à chaque instant. Quand il envoie le signal indiquant qu'il est libre, il reçoit simultanément une information par ?.ECR. Au moment où il est consulté par LIR.?, cette consultation est la seule communication possible.

Voyons maintenant la spécification des processus concurrents :

```
processus CDC portes I,O,P.
  cnx    ?.I, O.? : Nat
         ?.P : Signal
  etats  Q : Nat
  init   Q(0)
  vars   x,u : Nat
  regles
         Q(x) => ?.I(u)          Q(x+u)
         Q(x) => ?.P(zz)O.?(x)   Q(0)
fin
```

Les processus CDC réalisent la même fonction que RES : actualiser une donnée. Ils ont trois portes : I pour la réception de l'information, O pour la transmission qui sera possible au cas où un signal est reçu par la porte P. CDC se comporte de la façon suivante :

- tant qu'il reçoit des messages, il va les lire et les accumuler en faisant l'addition.
- quand le signal de permission ?.P lui arrive, alors il peut transmettre.

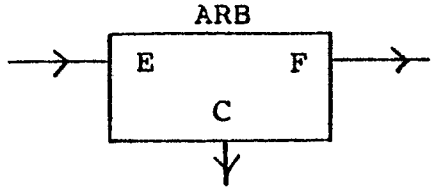


Comme on a déjà dit, la présence d'un arbitre pour contrôler la concurrence est nécessaire. Nous construisons cet arbitre en créant autant de processus ARB qu'il y a de concurrents. Chacun des ARB aura comme mission de signaler à chaque CDC la possibilité d'envoyer vers CNC et, en même temps, ils communiquent entre eux pour constituer le contrôleur général.

Chaque composant ARB de l'arbitre peut se trouver dans l'un des deux états suivantes :

1) Il est élu.
   - Quand il recevra le signal de que la ressource est libre, il transmettra au CDC qui lui correspond et celui là pourra envoyer vers CNC.
   - En même temps, il communique au prochain ARB qui est celui qui est élu.
   - Il passe dans l'état d'attente.
2) Il est dans l'état d'attente.
   - Il ne peut que transmettre le signal de rester en attente.
   - Lorsqu'il reçoit le signal qu'il sera élu, il change d'état.

Les processus ARB ont la forme :



où ?.E et F.? sont des connecxions internes à l'arbitre pour faire circuler les signaux entre eux ; et C.? représente l'envoi de la permission. Les états "élu" et "attente" sont implémentés au moyen d'un paramètre booléan "vrai", "faux". Alors, ARB est décrit de la façon suivante :

processus ARB lambda-y n:Nat. portes F,E,C.
cnx ?.E,F.?: Bool, C.?: Signal

```
etats Q : Bool
init  Q(faux)
vars  b : Bool
regles
      Q(vrai) => ?.E(b)C.?(zz)F.?(vrai) Q(faux)
      Q(faux) => ?.E(vrai)F.?(faux)     Q(vrai)
      Q(faux) => ?.E(faux)F.?(faux)     Q(faux)
fin
```

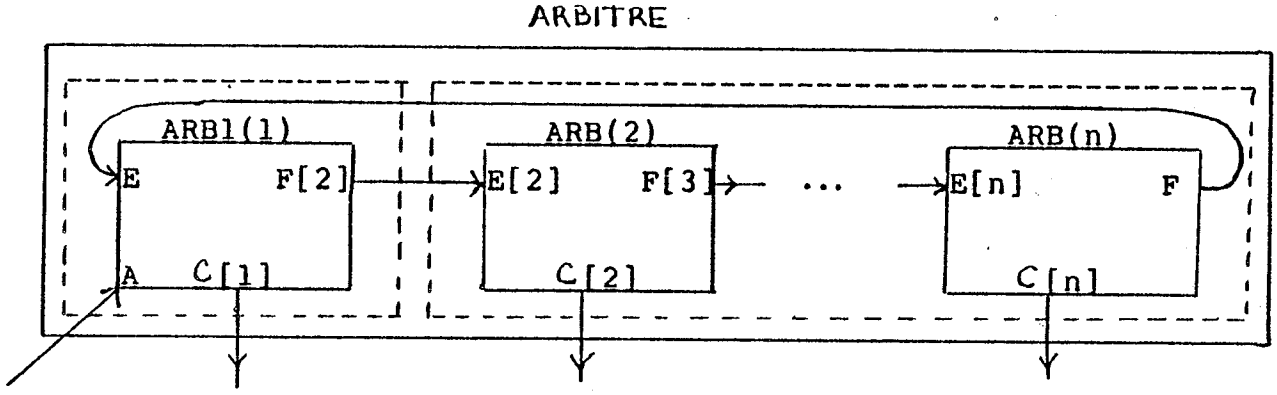
L'arbitre est composé d'un premier ARB1 qui est celui qui reçoit le signal de ressource libre (la tête) et n-1 processus ARB.

```
processus ARB1 lambda-v n:Nat. portes A,F,E,C.
  cnx   ?.A,C.?:Signal, ?.E,F.?:Bool
  etats Q:Bool
  init  Q(vrai)
  vars
  regles
    Q(vrai) => ?.A(zz)C.?(zz)F.?(vrai)?.E(vrai) Q(vrai)
    Q(vrai) => ?.A(zz)C.?(zz)F.?(vrai)?.E(faux) Q(faux)
    Q(faux) => ?.A(zz)?.E(vrai)F.?(faux)        Q(vrai)
    P(faux) => ?.A(zz)?.E(faux)F.?(faux)        Q(faux)
fin
```

Le processus ARBITRE sera un réseau de composants ARB1, ARB(n-1) organisé en anneau :

```
processus ARBITRE lambda-v n:Nat. portes F,E,C.
si n=1
alors ARB1(n)(A,F,E,C[n]) ⊕ F.E
sinon ARBITRE(n-1)(F[n].E,C) ||
      ARB(n)(F,E[n],C[n]) ⊕ F[n].E[n]
fsi
```

Graphiquement :

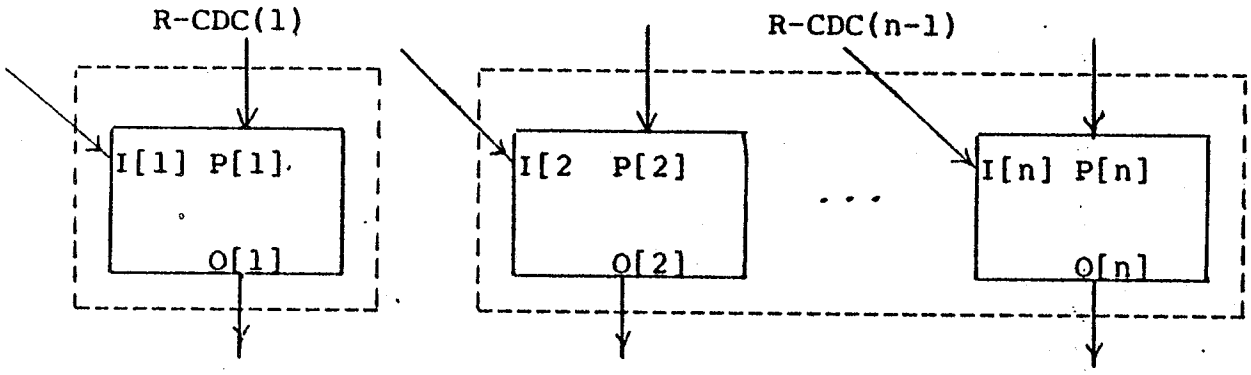


Les lignes pointillées reflètent la façon de construire le réseau.

On construit l'ensemble de centres departementaux de la même façon que l'ARBITRE, c'est à dire :

processus R-CDC = lambda-v n:Nat. portes I,O,P.
si n=1
alors CDC(I[n],O[n],P[n])
sinon R-CDC(n-1)(I[n-1],O[n-1],P[n-1]) || CDC(I[n],O[n],P[n])
fsi

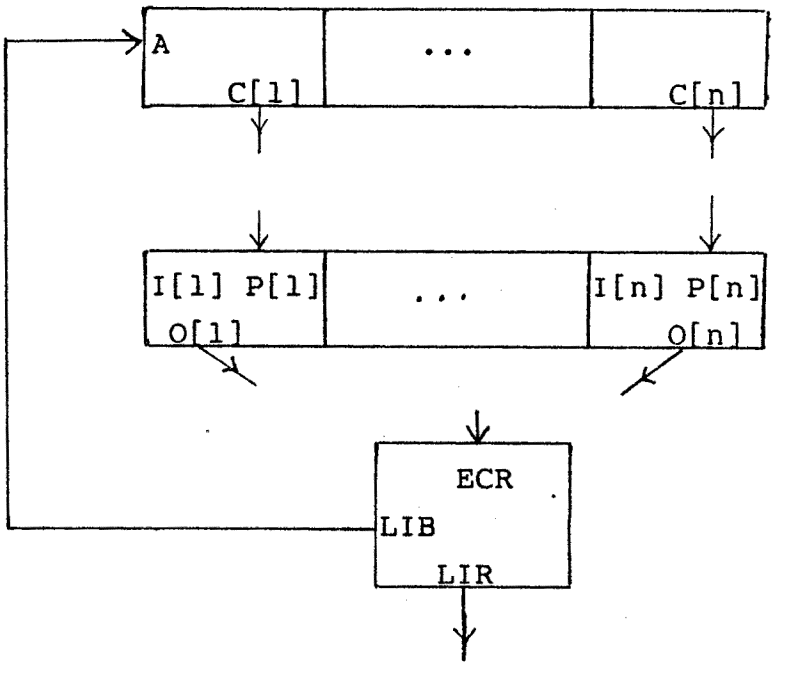
Graphiquement :



Maintenant on mettra le tout ensemble D-RESEAU, c'est à dire, les centres departementaux, le centre national de calcul et le contrôleur de la communication : l'arbitre.

processus D-RESEAU = lambda-v n:Nat.
ARBITRE(n) || R-CDC(n) || CNC⊕LIB.A

On a obtenu :

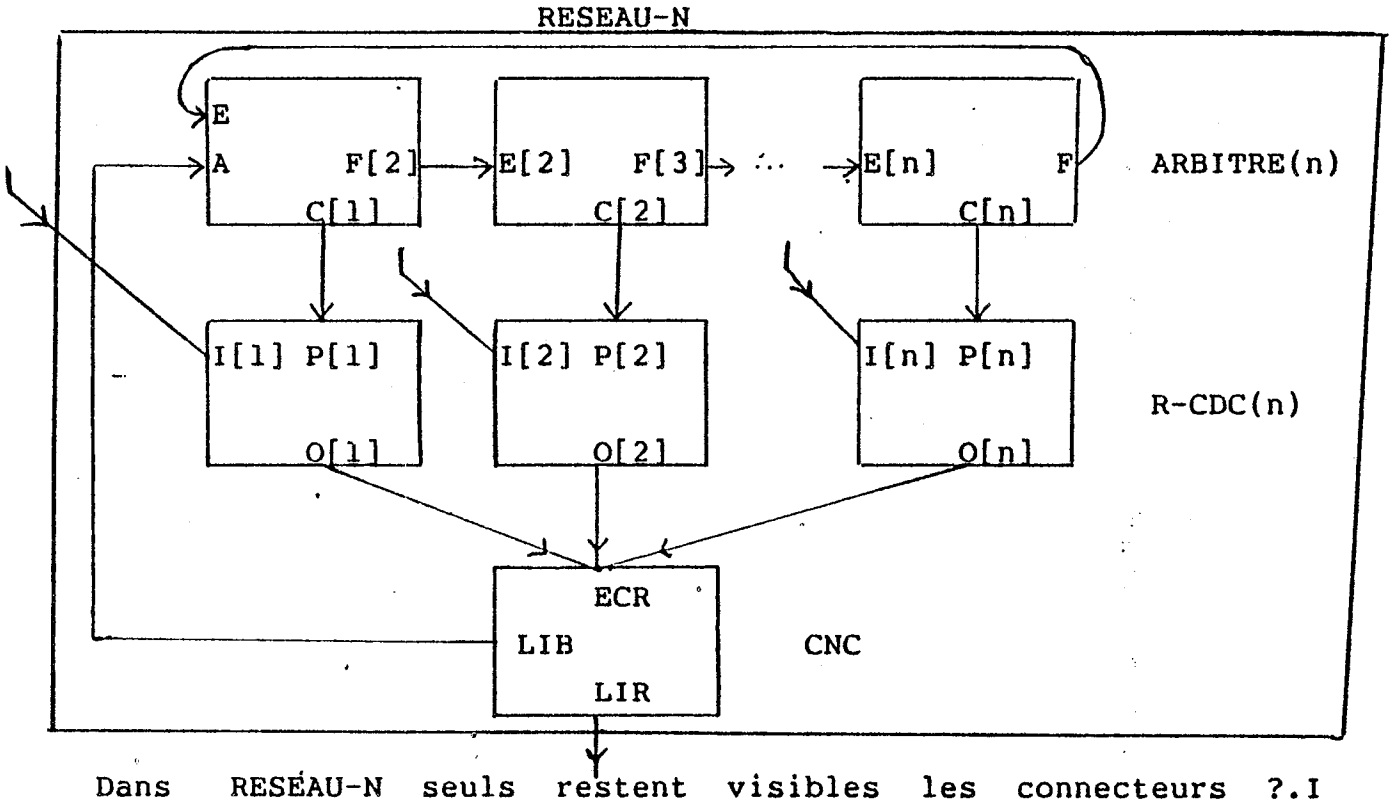


Il ne nous reste qu'à établir les connexions et cela on l'obtient au moyen du processus RESEAU.

```
processus RESEAU = lambda-v n,m:Nat. portes C,P,O.
si m=1
alors D-RESEAU(n) ⊕ C[1].P[1] ⊕ O[1].ECR
sinon RESEAU(n,m-1)(C[m-1],P[m-1],O[m-1]) ⊕
      C[m].P[m] + O[m].ECR - O[m]
fin
processus RESEAU-N = RESEAU(n,n)
```

Le processus RESEAU-N est le résultat du contrôle de n concurrents qui se partagent une ressource .

Graphiquement :



Dans RESEAU-N seuls restent visibles les connecteurs ?.I associés aux CDC et le connecteur LIR de CNC.
Le processus RESEAU-N aura 2n+1 règles : n règles dont l'événement est ?.I[n], 1 règle pour LIR.? et n règles Υ qui décrivent à qui est le tour d'utiliser la ressource.

Remarque. Le fait d'avoir la récursivité comme la seule forme de construction de réseaux possible (à part les opérateurs inhérents aux processus), alourdit dans certains cas la demarche. Par exemple, après le processus D-RESEAU qui met en parallèle les trois composantes : ARBITRE, R-CDC et CNC on a du faire deux autres "réseaux" (RESEAU et RESEAU-N) pour réaliser les connexions C[m].P[m] et O[m].ECR, m=1,..,n. Dans des langages applicatifs modernes, ce genre d'opérations se font au moyen de l'abstraction d'ensembles. Si l'on ajoutait "l'abstraction d'ensembles" à FP2, RESEAU-N pourrait être :

RESEAU-N = lambda-v n:Nat.
D-RESEAU(n) $\oplus$ $\{C[i].P[i],O[i].ECR \mid i<-[1,..,n]\}$

3.4. Exemple No.4: RESEAUX SYSTOLIQUES.
Calcul en parallèle d'une suite vectorielle.

Il s'agit d'un exemple résolu en programmant en FP2 un réseau systolique. Le concept de système systolique a été introduit par H.T. KUNG <Kun 80>, <Kun 81>, et on a tiré cet exemple d'un travail de Y. ROBERT et M. TCHUENTE <Rob-Tch 81> fondé sur les présentations de Kung.

Un système systolique est un réseau de processeurs ou cellules :
- qui ont tous la même structure,
- qui sont connectés entre eux,
- qui, au même rythme, effectuent des opérations élémentaires sur les valeurs en provenance de cellules voisines en entrée, puis envoient les résultats de celles ci vers des cellules voisines en sortie.

Ainsi, chaque cellule a-t-elle un mode de fonctionnement semblable à celui du coeur humain, d'où la terminologie. Le problème spécifique est de faire un calcul en "temps réel" d'une suite vectorielle.

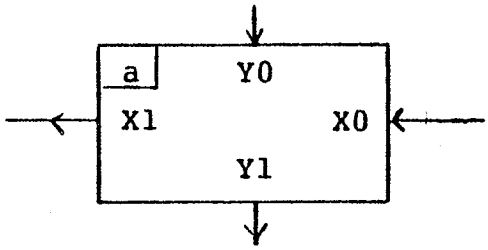
Enoncé

Soient A et B, deux matrices carrées d'ordre n. Etant donnée une suite Xi,i>=0 de vecteurs à n composantes, nous voulons calculer la suite Yi,i>=0, où Yi=AXi+BXi-1, à l'aide d'un réseau composé de $2n^2$ cellules (puisque celles ci n'effectuent que des opérations scalaires).
La description d'une cellule en FP2 est la suivante :
CELLULE a un paramètre qui prendra pour valeur, soit un élément de A, soit un élément de B, selon l'emplacement de la cellule dans le réseau.

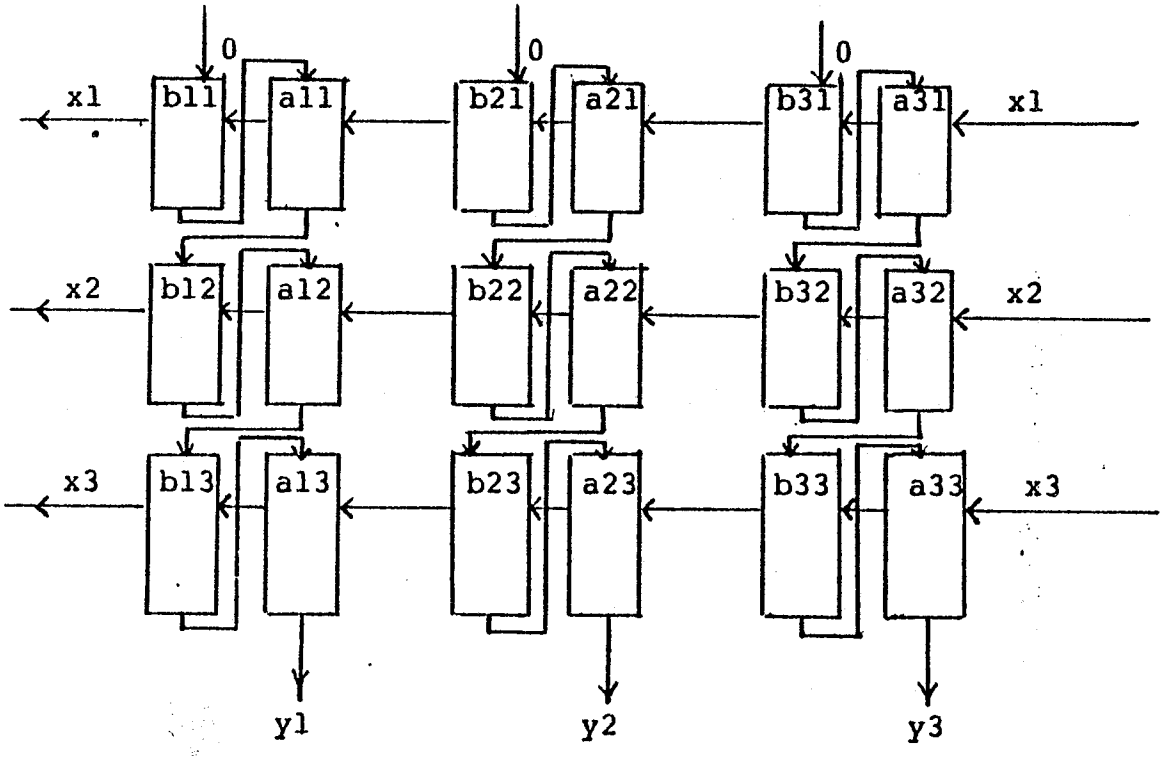
processus CELLULE lambda-v a:Nat. portes X0,X1,Y0,Y1.

```
cnx    ?.X0,X1.?,?.Y0,Y1.?:Nat
etats  Q:Nat
init   Q(0)
vars   x0,x1,y1:Nat
regles
       Q(x0) => ?.X0(x1)?.Y0(y1)X1.?(x0)Y1.?(y1+x0.a) Q(x1)
fin
```

Une cellule peut être représenté par :



Pour n=3, le réseau qui résout le problème, a la forme suivante :



A partir du graphique, on peut voir comment sont interconnectés les $2n^2$ processus CELLULE. Un rangement de n couples de

CELLULES donne une construction en forme de matrice carrée qui constitue le système systolique qui calcule la suite vectorielle. A droite, on fait rentrer les vecteurs X qu'on va récupérer par les sorties à gauche. Les Y calculés seront transmis verticalement, les vecteurs Y résultants se trouvent en bas. Les valeurs entrées par le haut sont des "0".

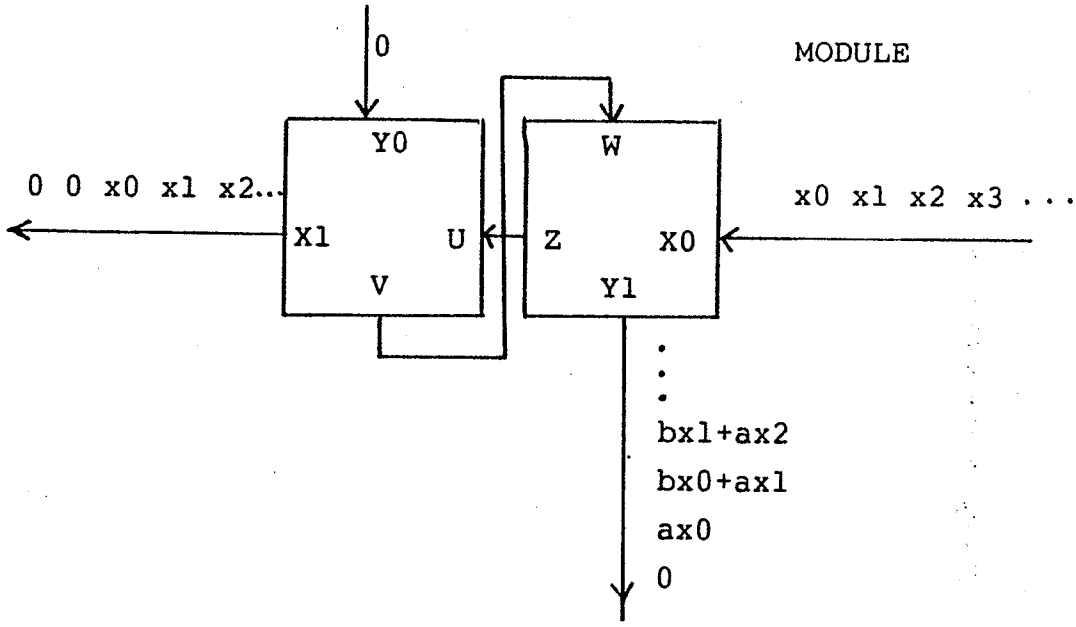
Nous allons décrire un de ces couples de CELLULES, en l'appelant MODULE. a et b sont des valeurs constantes qui représentent les éléments des matrices A et B.

processus MODULE = lambda-v a,b:Nat. portes X0,Y0,X1,Y1.
CELLULE(b)(U,Y0,X1,V) || CELLULE(a)(X0,W,Z,Y1)⊕V.W⊕Z.U

Le résultat de cette expression, est un processus MODULE qui a comme état initial Q(0,0) et comme seule règle :

Q(u0,x0) => ?.X0(x1)X1.?(u0)?.Y0(y1)Y1.?(y1+bu0+ax0) Q(x0,x1)

Le processus MODULE, graphiquement peut être représenté par :



Un MODULE fonctionne de la façon suivante : pour une suite de valeurs x0,x1,x2,x3,... entrée en X0, on obtient 0,ax0,bx0+ax1,bx1+ax2,... par la sortie Y1. On retrouve l'entrée 0,0,x0,x1,... à la sortie X1. MODULE produit donc, un zéro dans le calcul vertical des Y, et deux zéros dans

l'horizontal. Il faudrait supprimer ces sorties inutiles pour obtenir les résultats souhaités. Mais, pour l'instant, voyons comment on construit le réseau matrice. Soient :
n : la dimension du vecteur X
i : l'indice pour les colonnes,
j : l'indice pour les lignes, et k : une constante qui nous servira pour recupérer la valeur de n dans le processus MATRICE.

```
processus RESEAU = lambda-v n:Nat. MATRICE(n,n,n)
processus MATRICE = lambda-v i,j,k. portes X0,Y0,X1,Y1.
si j=1
alors si i=1
      alors
       MODULE(a[1,1],b[1,1])(X0[1,1],Y0[1,1],X1[1,1],Y1[1,1])
      sinon
       MATRICE(i-1,k,k)||MODULE(a[i,1],b[i,1])
       (X0[i,1],Y0[i,1],X1[i,1],Y1[i,1])⊕X1[i,1].X0[i-1,1]
      fsi
sinon si i=1
      alors
       MATRICE(1,j-1,k)||MODULE(a[i,j],b[i,j])
       (X0[1,j],Y0[1,j],X1[1,j],Y1[1,j])⊕Y1[1,j-1].Y0[1,j]
      sinon
       MATRICE(i,j-1,k)||MODULE(a[1,j],b[1,j])
       (X0[i,j],Y0[i,j],X1[i,j],Y1[i,j])
       ⊕Y1[i,j-1].Y0[i,j]⊕X0[i-1,j].X1[i,j-1]
      fsi
fsi
```

Le processus MATRICE est le résultat d'une expression conditionnelle qui construit récursivement de bas en haut et de droite à gauche. Il nous faut encore ajouter certains éléments au réseau MATRICE pour qu'il produise les résultats attendus :
1. Il est nécessaire d'envoyer des zéros aux entrées Y0 de la première ligne,

2. Pour mettre au point les résultats, il faudra filtrer les sorties pour enlever les valeurs sans intèrêt, c'est à dire, les zéros produits par le système quand il fait les communications horizontales et verticales.
3. Après avoir obtenu les résultats "propres", nous devons les synchroniser. On veut avoir, par exemple, la première composante de tous les vecteurs Y en même temps ; de même pour les X.

Nous allons donc, construire les "machines" qui réalisent les tâches énoncées ci dessus :

L'émetteur permanent de zéros sera un processus très simple, avec une seule porte et une seule règle : il envoie un zéro à travers sa porte de sortie.

```
processus ZERO portes S.
  cnx   S.?:Nat
  etats P:()
  init  P
  vars
  regles
        P => S.?(0) P
fin
```

Quand au filtre et au synchronisateur, nous allons tout effectuer dans un même processus SYNCHR. Mais avant d'expliquer son fonctionnement, nous allons calculer "combien" de valeurs il faut ignorer. Rappelons que n est la dimension des vecteurs X et, par conséquent, celle de Y.

Filtrage des X de sortie (X1)

Etant donné que chaque MODULE produit deux zéros avant de reproduire l'entrée, la quantité de sorties à ignorer au debut de chaque résultat d'une ligne est 2n.

Filtrage des Y de sortie (Y1)

Etant donné que chaque MODULE produit un zéro dans son résultat vertical, et qu'il va accumuler aussi un zéro par CELLULES des colonnes précedentes, le nombre total de sorties à ignorer est, pour chaque colonne i, 2(n-i)+1.

On doit aussi calculer pour chaque colonne, le nombre de fois qu'elle doit attendre avant de laisser passer les sorties de façon à synchroniser tous les Y. La synchronisation n'est pas nécessaire pour les lignes car elles produisent les sorties en même temps. Chaque colonne i doit attendre le nombre de zéros de la première (celle qui produit le plus) : 2n-2+1, moins le nombre de zéros qui lui sont propres : 2n-2i+1; c'est à dire : 2i-2.

Les valeurs correspondant aux 2i-2 sorties qu'il faut retarder, seront gardées par les processus SYNCHR dans un buffer qu'on commencera à vider au "2.i-2+1 ème" instant :
Le processus SYNCHR aura trois paramètres de valeur m,n et b. m reçoit la valeur de i (indice de colonne), n reçoit la valeur k (constante représentant la dimension des vecteurs X) et b est une valeur booléenne dont la fonction est de distinguer si SYNCHR est appelé pour travailler sur les lignes (b=vrai) ou sur les colonnes (b=faux).

La première des règles correspond au calcul pour les colonnes. R1 a comme premier paramètre le nombre de zéros à enlever et, comme deuxième, le nombre de fois à attendre pour synchroniser cette colonne avec les autres. La deuxième règle fait la même chose pour les lignes. La spécification de SYNCHR est :

processus SYNCHR lambda-v n,m:Nat, b:Bool. portes C,Cl.
cnx ?.C,Cl.?:Nat
etats R:Bool Nat Nat, R1:Nat Nat
init R(b,n,m)

```
vars  x,y,u:Nat, s:Seq
regles
      R(faux,n,m)    => T              R1(2n-2m+1,2m-2)
      R(vrai,n,m)    => T              R1(2n,0)
      R1(succ(x),u)  => ?.C(0)         R1(x,u)
      R1(0,succ(u))  => ?.C(y)         R2(cons(y,nil),u)
      R1(0,0)        => ?.C(y)C1.?(y)  R1(0,0)
      R2(s,succ(u))  => ?.C(y)         R2(cons(y,s),u)
      R2(s,0)        => C1.?(tête(s))  R2(reste(s),0)
      R2(nil,0)      => T              R1(0,0)
fin
```

Maintenant que nous avons tous les processus réalisant les tâches nécessaires pour que le réseau MATRICE produise les résultats propres et synchronisés, nous allons présenter sa nouvelle spécification.

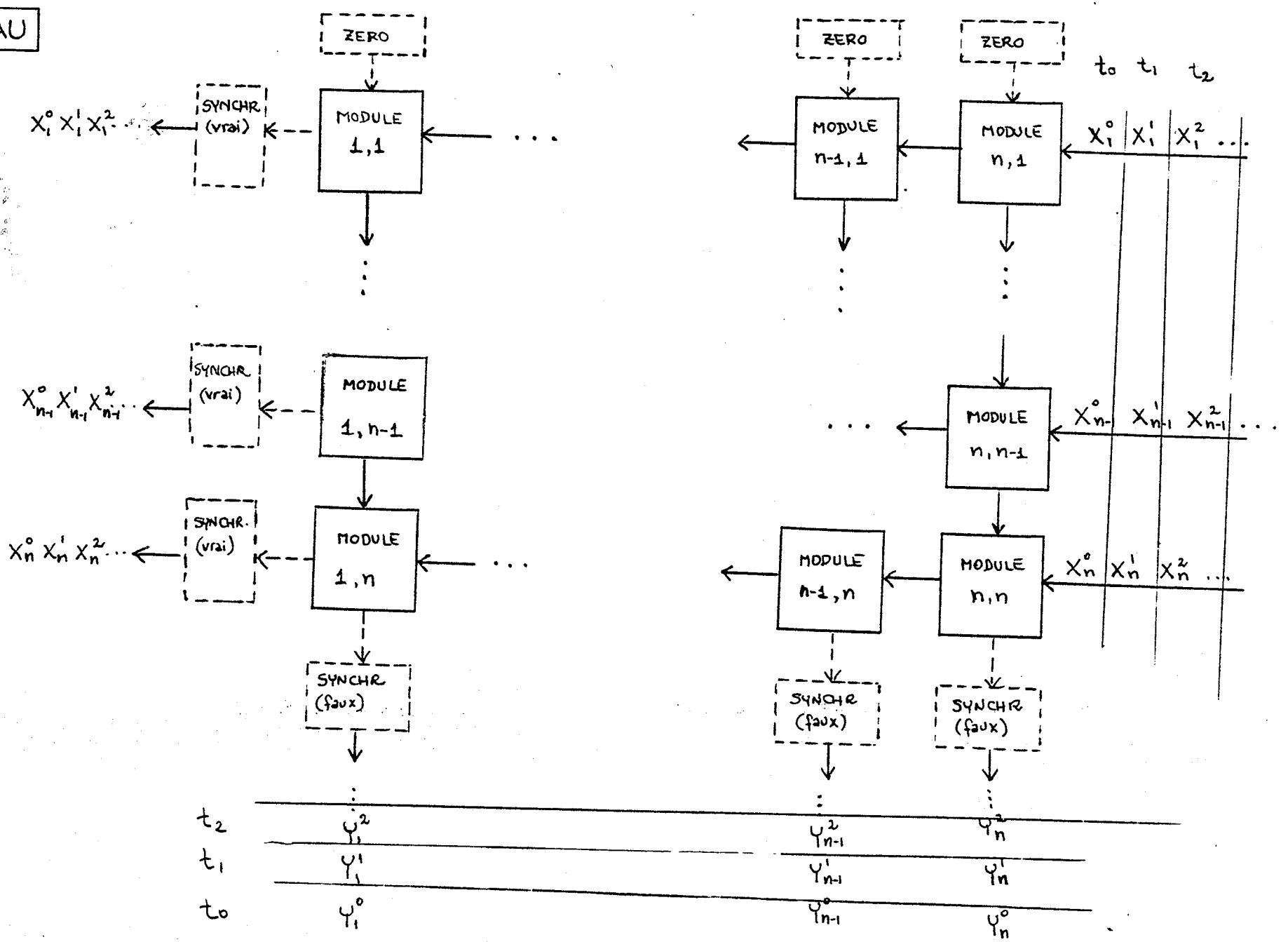
```
processus RESEAU = lambda-v n:Nat, MATRICE(n,n,n)

processus MATRICE = lambda i,j,k:Nat
si j=1
alors si i=1
      alors
       MODULE(a[1,1],b[1,1])(X0[1,1],Y0[1,1],Z1[1,1],Y1[1,1]) ||
       ZERO(S[1])||SYNCHR(k,1,vrai)(C[1,1],X1[1,1])⊕S[1].Y0[1,1]
       ⊕Z[1,1].C[1,1]
      sinon
       MATRICE(i-1,k,k)||MODULE(a[i,1],b[i,1])
       (X0[i,1],Y0[i,1],X1[i,1],Y1[i,1]||ZERO(S[i])
        ⊕ X1[i,1].X0[i-1,1] ⊕ S[i].Y0[i,1]
      fsi
sinon si i=1
      alors si j=k
            alors
             MATRICE(1,k-1,k)||MODULE(a[1,k],b[1,k])
```

```
            (X0[1,k],Y0[1,k],Z[1,k],W[1,k])||
            SYNCHR(k,1,vrai)(C[1,k],X1[1,k])||
            SYNCHR(k,1,faux)(D[1,k],Y1[1,k])||
            ⊕Y1[1,k-1].Y0[1,k]⊕Z[1,k].C[1,k]⊕W[1,k].D[1,k]
          sinon
            MATRICE(1,j-1,k)||MODULE(a[1,j],b[1,j])
            (X0[1,j],Y0[1,j],Z[1,j],Y1[1,j])||
            SYNCHR(k,1,vrai)(C[1,j],X1[1,j])||
            ⊕ Y1[1,j-1].Y0[1,j] ⊕ Z[1,j].C[1,j]
          fsi
      sinon si j=k
          alors
            MATRICE(i,k-1,k)||MODULE(a[i,k],b[i,k])
            (X0[i,k],Y0[i,k],X1[i,k],W[i,k]) ||
            SYNCHR(k,i,faux)(C[i,k],Y1[i,k]) ||
            ⊕Y1[i,k-1].Y0[i,k]⊕X1[i,k].X0[i-1,k]⊕W[i,k].C[i,k]
          sinon
            MATRICE(i,j-1,k)||MODULE(a[i,j],b[i,j])
            (X0[i,j],Y0[i,j],X1[i,j],Y1[i,j])
            ⊕Y1[i,j-1].Y0[i,j]⊕X1[i,j].X0[i-1,j]
          fsi
      fsi
fsi
```

Voyons le résultat du processus RESEAU graphiquement :

RESEAU


ZERO
SYNCHR (vrai)
MODULE 1,1
$X_1^0 X_1^1 X_1^2 \ldots$
SYNCHR (vrai)
MODULE 1, n-1
$X_{n-1}^0 X_{n-1}^1 X_{n-1}^2 \ldots$
SYNCHR (vrai)
MODULE 1, n
$X_n^0 X_n^1 X_n^2 \ldots$
SYNCHR (faux)
ZERO
ZERO
MODULE n-1, 1
MODULE n, 1
$X_1^0 X_1^1 X_1^2 \ldots$
MODULE n-1, n-1
MODULE n, n-1
$X_{n-1}^0 X_{n-1}^1 X_{n-1}^2 \ldots$
MODULE n-1, n
MODULE n, n
$X_n^0 X_n^1 X_n^2 \ldots$
SYNCHR (faux)
SYNCHR (faux)
t_0 t_1 t_2
t_2 t_1 t_0
Y_1^2 Y_1^1 Y_1^0
Y_{n-1}^2 Y_{n-1}^1 Y_{n-1}^0
Y_n^2 Y_n^1 Y_n^0

4. EXTENSIONS A FP2

4.1. Formes Fonctionnelles

Sur la base de l'existence de fonctions et définitions de types, on propose d'ajouter au langage la possibilité de combiner les fonctions en formes fonctionnelles au moyen d'opérateurs fonctionnels. Ces opérateurs sont des versions typées de ceux de FP <Bac 78>, et leur sémantique formelle est simple : les formes fonctionnelles sont évaluées et les résultats sont des définitions équationnelles de fonctions.

Opérateurs Fonctionnels et Formes Fonctionnelles

Dans <Jor 84> six opérateurs fonctionnels sont proposés pour combiner les fonctions définies en formes fonctionnelles. En plus des types et propriétés prédéfinis en LPG (Seq(t), monoide), on a besoin d'un opérateur générique COND.

```
enrich E exige tyfo[t]
opns cond : (bool,t,t) -> t
vars x,y : t
eqns cond(vrai,x,y) == x
     cond(faux,x,y) == y
end
```

Soient r,s,t,t1,t2,...,tn des types. Les six opérateurs fonctionnels sont :

1. Composition

Si f et g sont des fonctions, f:t->r et g:r->s, alors g.f est une forme fonctionnelle qui dénote une opération en t->s. Si x est un terme de type t, la réduction de (g.f)(x) est la réduction de g(f(x)).

2. Condition

Si p, f et g sont des fonctions avec p:t->bool, f:t->r et g:t->r, alors (p->f;g) est une forme fonctionnelle qui dénote

une opération en t->r. Si x est un terme de type t, la réduction de (p->f;g)(x) est la réduction de cond(p(x),f(x),g(x)).

3. Tuple

Si f1,f2,...,fn sont des fonctions telles que fi:t->ti, alors [f1,f2,...,fn] est une forme fonctionnelle qui denote une opération en t->(t1,...,tn). Si x est un terme de type t, alors la réduction de [f1,...fn](x) est la réduction de [f1(x),...,fn(x)].

4. Constante

Si x est un terme de type t, alors x est une forme fonctionnelle qui dénote une opération en r->t pour un type r quelconque. Si y est un terme de n'importe quel type r, alors la réduction de x (y) est la réduction de x.

5. Application

Si f est une fonction, f:t->r, alors *f est une forme fonctionnelle qui dénote une opération en Seq[t]->Seq[r]. Si x est de type Seq[t], alors il y a deux cas :

i) si x réduit à nil, alors *f(x) réduit à nil,

ii) si x réduit à cons(e,y), alors la réduction de *f(x) est la réduction de cons(f(e),*f(y)).

6. Insertion

Si f est une fonction, f:(t,t)->t, alors /f est une forme fonctionnelle qui dénote une opération en Seq[t]->t, étant donné que le type t satisfait la propriété monoide avec les opérations f et 0. Si x est de type seq[t], il existe deux cas :

i) si x réduit à nil, alors /f(x) réduit à 0,

ii) si x réduit à cons(e,y), alors la réduction de /f(x) est la réduction de f(e,/f(y)).

La sémantique des opérateurs fonctionnels est définie en considérant que les formes fonctionnelles sont des expressions

de second ordre qui peuvent être évaluées. Cela est possible en FP2, car les définitions de fonctions (noms d'opérateurs, domaine, codomaine, variables et équations) constituent une forme élémentaire pour décrire les opérations et, évaluer des formes fonctionnelles, signifie produire ces définitions élémentaires d'opérations.

Soit F(x) un terme où F est une forme fonctionnelle. L'évaluation de F est guidée par sa syntaxe : de noms d'opérations f0,f1,... sont générés ; un pour chaque sous forme syntaxique en F où f0 est l'opération pour F. Ainsi, une définition élémentaire d'opération pour f0,f1,... avec ses noms, domaines, codomaines, variables et équations, peut être générée automatiquement avec toutes les clauses "exige" nécessaires. A la fin, F(x) est remplacé par f0(x) qui a sa réduction définie par la définition élémentaire d'opération générée.
Par exemple, en supposant que monoide [Nat opns add,0] est un modèle, l'évaluation de /add produit :

```
enrich e0
opns f0 : Seq[Nat]->Nat
vars v0 : Nat
     v1 : Seq[Nat]
eqns f0(nil) == 0
     f0(cons(v0,v1)) == add(v0,f0(v1))
fin
```

Un terme comme /add(x) où x doit être de type Seq[Nat] est alors remplacé par f0(x) et réduit à l'addition des éléments de x.

Les formes fonctionnelles peuvent aussi être appelées explicitement. Supposons monoide[Nat opns mul,1] un autre modèle de monoide. Il sera possible de définir :

```
enrich sigma = /add
enrich pi    = (null-> 0 ;/mul)
enrich sigpi = sigma.*pi
```

Cela produit :

- une définition comme celle de e0, mais à la place de f0 on met sigma,
- deux définitions

```
enrich e1 exige tyfo[to]
  opns pi : Seq[Nat]->Nat
       f1 : t0->Nat
       f2 : Seq[Nat]->Nat
  vars v0 : Nat, v1 : Seq[Nat], v2 : t0
  eqns pi(v1) == cond(null(v1),f1(v1),f2(v1))
       f1(v2) == 0
       f2(nil) == 1
       f2(cons(v0,v1)) == mul(v0,f2(v1))
fin
enrich e2
  opns sigpi : Seq[Seq[Nat]]->Nat
       f3    : Seq[Seq[Nat]]->Seq[Nat]
  vars v0    : Seq[Nat]
       v1    : Seq[Seq[Nat]]
  eqns sigpi(v1) == sigma(f3(v1))
       f3(nil)   == nil
       f3(cons(v0,v1)) == cons(pi(v0),f3(v1))
fin
```

Des définitions récursives d'opérateurs avec formes fonctionnelles peuvent aussi se faire d'une façon simple. Regardons par exemple, le calcul du "pgcd" d'après l'algorithme d'Euclide. On supposse que les définitions suivantes sont déjà faites :

```
enrich projection exige tyfo[t]
  opns .1 : (t,t) -> t
       .2 : (t,t) -> t
  vars x,y : t
  eqns .1(x,y) == x
       .2(x,y) == y
fin
```

```
enrich op-nat
  opns moins : (Nat,Nat) -> Nat
       eq    : (Nat,Nat) -> Bool
       gr    : (Nat,Nat) -> Bool
  vars m,n : Nat
  eqns moins(m,n) == m-n
       eq(m,n)    == m=n
       gr(m,n)    == m>n
fin
```

```
enrich !1 = .1[Nat]      enrich eqxy = eq.pr
enrich !2 = .2[Nat]      enrich grxy = gr.pr
enrich pr = [!1,!2]      enrich a    = moins.pr
enrich rp = [!2,!1]      enrich b    = moins.rp
```

Le pgcd de deux nombres est alors calculé par :

```
enrich pgcd = eqxy->!1; grxy->pgcd.[a,!2]; pgcd[!1,b]
```

La définition du pgcd produit la définition élémentaire d'opération suivante :

```
enrich e3
  opns pgcd: (Nat,Nat)->Nat
       f4 : (Nat,Nat)->Nat
       f5 : (Nat,Nat)->Nat
       f6 : (Nat,Nat)->Nat
       f7 : (Nat,Nat)->(Nat Nat)
       f8 : (Nat,Nat)->(Nat Nat)
  vars v0 : (Nat,Nat)
  eqns pgcd(v0)== cond(eqxy(v0),!1(v0),f4(v0))
       f4(v0) == cond(grxy(v0),f5(v0),f6(v0))
       f5(v0) == pgcd(f7(v0))
       f6(v0) == pgcd(f8(v0))
       f7(v0) == (a,!2)
       f8(v0) == (!1,b)
fin
```

Comme les équations définissent des classes de congruence de termes, les lois de l'algèbre de programmes fonctionnels de Backus <Bac 78> peuvent être prouvés en FP2.

4.2. Les processus comme paramètres: Processus Formels.

En FP2 on peut avoir différentes sortes de paramètres : types, valeurs, fonctions, portes,... pourquoi pas des processus? En effet, FP2 étant un langage générique, il paraît souhaitable de pouvoir paramètrer les processus avec d'autres processus ; ce faisant on augmente la puissance des expressions.

On a vu que le regroupement de processus, types et fonctions se réalise d'une façon harmonieuse et, c'est pour cela, que l'idée d'implémenter les processus-paramètres est fondamentalement la même que pour les types, à travers des propriétés. Mais, malgré la ressemblance entre types et processus, il existe une grande différence qui est responsable de l'impossibilité de définir un processus comme un type abstrait : la notion de communication, laquelle implique toute une description de comportement.

Dans la programmation générique, l'entité "propriété" a été créée pour jouer un rôle de description minimale des "caractéristiques" essentielles d'un objet. Ainsi, les types formels exigent du type effectif la vérification d'une certaine propriété, c'est à dire, la garantie que le type effectif réalise un certain ensemble de fonctions. Maintenant, pour les processus communicants, quelles sont les caractéristiques à exiger d'un processus effectif? Ou en posant la question d'une autre façon, que doit contenir une propriété de processus? Un propriété de processus "prop-proc" va établir des contraintes sur :

- les portes,
- les types d'objets manipulés,
- le comportement, qui concerne à la fois les séquences de communication et les fonctions effectuées sur les entrées.

En fait, une propriété de processus n'est qu'une spécificaiton de processus avec ses portes et son ensemble de règles. Ainsi, prouver qu'un processus Q satisfait une propriété P, revient à prouver que Q est une implémentation correcte de P. Par exemple,

processus EXP = ... exige P [Q portes ...]

où P est une propriété de processus et Q est un processus formel. L'instanciation, processus EXP1 = EXP [Q(...)] sera correcte ssi P<=Q, où <= est une relation "d'implémentation correcte" qui resterait à caractériser.

Nous aborderons ici le problème d'une façon beaucoup plus simple, en traitant seulement ce qui concerne les portes. Ainsi, une propriété de processus, prop-proc, consiste à donner un ensemble de connecteurs avec les types correspondants.

```
prop-proc nom-de-propriété
  cnx
    .
    .
    .
    .
fin
```

Les propriétés de processus peuvent être liées entre elles par l'inclusion de la même façon que les propriétés de types à travers la clause "satisfait". Par exemple,

```
prop-proc p1
  cnx ?.A : Nat
      B.? : Nat
fin
```

```
proc-prop p2
  cnx ?.I : Nat
      ?.L : Nat
      O.? : Nat
  satisfait p1[L,O]
fin
```

- p2 avec ses portes ?.L et O.? vérifie la propriété p1.
- p1 p2

Supposons maintenant qu'on a la spécification d'un certain processus Q :

```
processus Q ... portes R,S,T,W,U. ....
  cnx ?.R : Nat
      ?.S : Nat
      ?.T : Nat
      W.? : Nat
      U.? : Nat
          .
  satisfait p2[?.R,?.S,U.?]
fin
```

Le processus Q satisfait la propriété p2 avec ses connecteurs ?.R,?.S et U.?.

Si l'on a une expression,

```
  processus EXP1 = ... exige p2[q portes ?.M,?.N,C.?] ...
```

une instanciation correcte est par exemple :

```
  processus A = EXP1[Q(R1,S1,T1,W1,U1)] ...
```

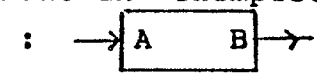
et dans l'expression les connecteurs qui seront utilisés sont ?.R1 comme ?.M, ?.S1 comme ?.N et U1.? comme C.?. Cette association se fait automatiquement.

Voyons une deuxième expression :

```
  processus EXP2 = ... exige p1[q portes ?.M,?.N] ...
```

Comme p2 satisfait p1, tous les processus satisfaisant p2, vérifient aussi p1. On peut alors faire l'instanciation,

processus B = EXP2 [Q(R2,S2,T2,W2,U2)] ...
dont les portes effectives correspondantes à ?.M et N.? sont ?.S2 et U2.?.

Voyons maintenant un exemple. La construction linéaire d'un même processus : 

CHAINE

Le processus CHAINE est une construction systolique à une dimension dont l'unité de base est un processus quelconque disposant d'une porte d'entrée A et d'une porte de sortie B, et dont les objets sont d'un type quelconque qui vérifie la propriété tyfo.

```
processus CHAINE = exige p3 [q portes A,B]. lambda-v n:Nat.
  si n=1
  alors q(A,B)
  sinon CHAINE[q(A,B[n])](n-1) || q(A[n],B)⊕B[n].A[n]
  fsi
```

La propriété p3 est définie par :

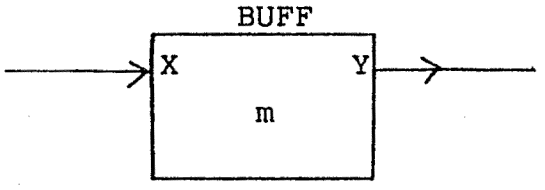
```
prop-proc p3 exige tyfo[t]
  cnx ?.I : t
      O.? : t
fin
```

Soient les processus suivants :

```
processus BUFF exige tyfo[t].
  cnx   ?.X : t, Y.? : t
  etats Q : (), Q1 : t
  init  Q
  vars  m:t
  regles
       Q     => ?.X(m) Q1(m)
       Q1(m) => Y.?(m) Q
  satisfait p3[X,Y]
```

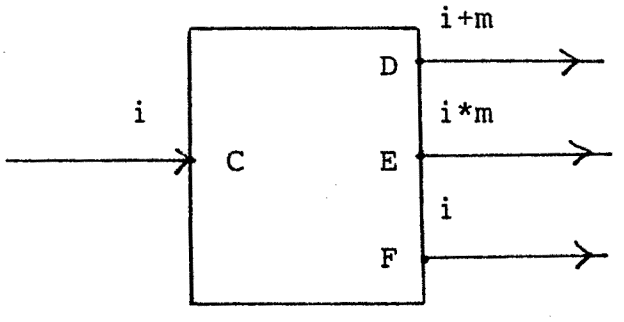
fin

Le processus BUFF est un registre à une case,




```
processus ADD-MUL exige monoide[t opns +,0,*,1]. lambda-v m:Nat.
  cnx   ?.C : t
        D.? : t
        E.? : t
        F.? : t
  etats P:()
  init  P
  vars  i:t
  regles
        P => ?.C(i)D.?(i+m)E.?(i*m)F.?(i) P
fin
```

Le processus ADD_MUL reçoit une valeur i et simultanément va envoyer l'addition +m, la multiplication *m et la valeur reçue.



Alors, si l'on fait

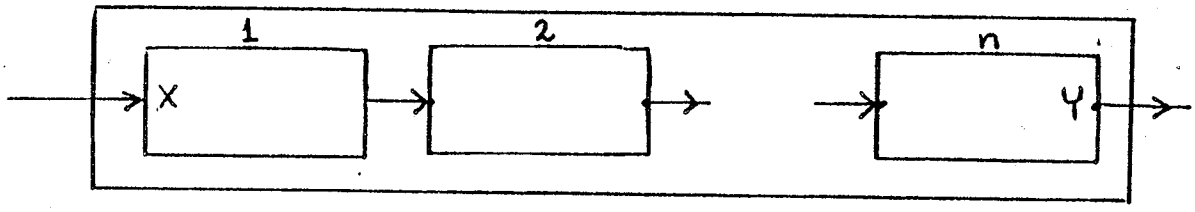
```
  processus BUFFER-N-NAT = CHAINE [BUFF[Nat]](n)
```

ce qu'on obtient c'est un buffer à n cases dont les éléments à enregistrer sont des Naturels. Mais, on aurait pu faire aussi :

```
  processus BUFFER-N-CAR = CHAINE [BUFF[Car]](n)
```

c'est à dire, en passant le processus BUFF comme paramètre à l'expression CHAINE, on construit des buffers d'objets de type

't'.

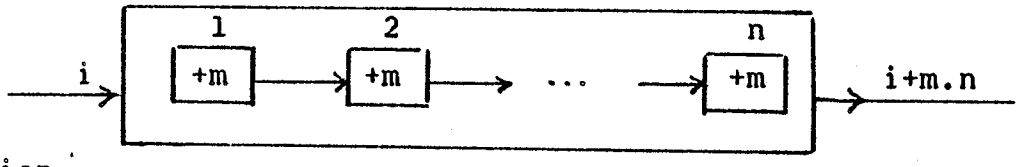


BUFFER-N-t

Le processus ADD-MUL peut être aussi paramètre effectif de CHAINE. En plus, ADD-MUL vérifie la propriété p3 de trois façons, soit avec les portes ?.C,D.?, soit avec ?.C,E.?, soit avec ?.C,F.?. Selon notre intérêt, nous pouvons faire les instanciations suivantes :

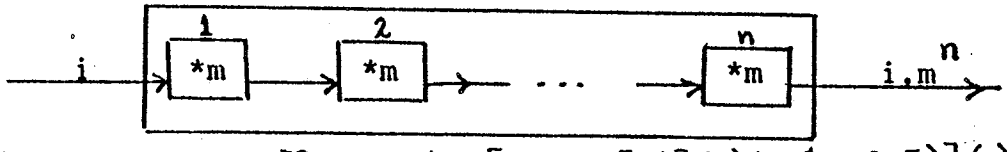
processus I+MN = CHAINE [ADD-MUL[Nat](m)(portes C,D)](n)

On obtient une chaine de n processus où chacun ajoute m à l'entrée :

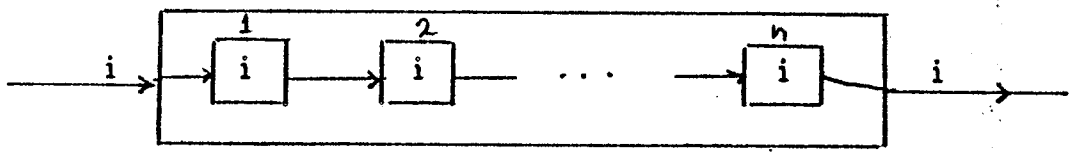


ou bien,

processus IM = CHAINE[ADD-MUL[Nat](m)(portes C,E)](n)



ou bien, processus ID = CHAINE [ADD-MUL [Nat] (m)(portes C,F)] (n)



Comme on peut voir dans cet exemple, il existe une autre façon de spécifier les portes à utiliser du processus effectif : c'est de les nommer précédées du mot portes tout de suite après l'instanciation. Dans les autres exemples, la spécification du processus contenait la clause satisfait p suivie des portes

d'intérêt ; de cette manière il n'était pas nécessaire de spécifier les portes au moment de l'instanciation, car cela se passe automatiquement en faisant la liaison.

Mais notre processus ADD-MUL satisfait la propriété p3 avec trois paires de portes. Si nous mettons la clause "satisfait" dans la spécification, nous restreignons la vérification de la propriété à une seule paire. Dans des cas comme celui là, il vaut mieux faire la déclaration explicitement pour permettre le choix.

CONCLUSION

Nous avons donc commencé par exposer deux des styles de programmation sur lesquels de nombreux travaux de logiciel sont aujourd'hui concentrés: la programmation applicative et la programmation parallèle.

Dans cette thèse on propose un langage: FP2 (Functional Parallel Programming). Ce langage a pour cadre le projet SPARC (Systèmes PARallèles Communicants). Le langage FP2 réunit les caractéristiques suivantes:

1. C'est un langage applicatif pour la description de la communication et du parallélisme. Il appartient à un style de programmation mixte: programmation applicative et programmation parallèle.

2. Le langage permet la définition de processus ainsi que de types abstraits et de fonctions. Un processus est constitué d'un ensemble de déclarations, de portes et d'états, et d'un ensemble de règles qui définissent son comportement communicant et fonctionnel.

3. En ce qui concerne les fonctions, un processus est capable d'exprimer toutes les fonctions calculables, voir <Per 84>.

4. En ce qui concerne la communication, un processus peut exprimer n'importe quel comportement individuel, ainsi que le comportement des réseaux construits à partir de ses trois opérateurs de base: l'union (||), la connexion (+) et l'abstraction (-).

5. Le calcul des opérateurs de processus donne comme résultat un autre processus avec ses déclarations et ses règles. Cela est dû

au fait que l'application des opérateurs n'entraîne que des manipulations syntaxiques de termes, ce qui permet une visibilité du calcul à tout moment.

6. FP2 introduit deux autres opérateurs de processus en plus des trois proposés en SPARC: le "⊕" (connexion suivie des abstractions sur les connecteurs concernés par la connexion) et le conditionnel si alors sinon.

7. FP2 est un langage fortement typé et générique.

8. En FP2 l'observation de communications simultanées (l'événement) est possible.

9. Les caractéristiques 6, 7 et 8 rendent la programmation FP2 très puissante en ce qui concerne la richesse d'expression et la construction des structures. La description des solutions à des nombreux problèmes est réalisée d'une façon simple, élégante et puissante comme le montrent les exemples présentés au chapitre III.

10. La complexité du résultat du calcul d'un réseau est en relation directe avec la complexité du problème à résoudre. Voir chapitre III, exemple No.2, cas 1.

11. Deux extensions à FP2 sont proposées :
- l'introduction des formes fonctionnelles semblable à celles de FP <Bac 78>.
- la possibilité du passage de processus comme paramètres.

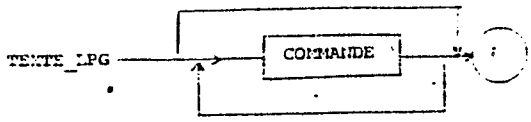
12. La sémantique de FP2 se ramène complètement à la sémantique de SPARC: les processus sont des algèbres de termes, ainsi que les types et les fonctions. Cependant d'autres interprétations sémantiques ont été faites; voir <Per 84>.

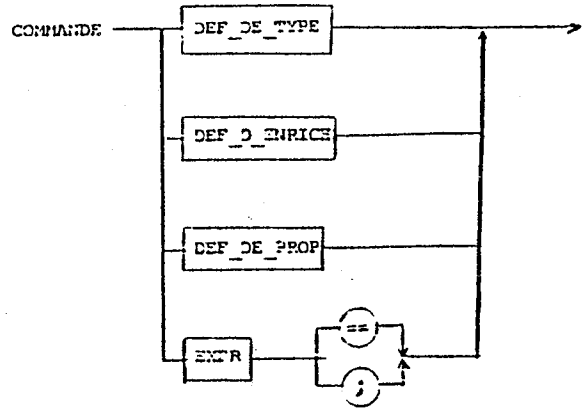
13. FP2 dans son état actuel, bien que langage puissant, est trop rigide en ce qui concerne les constructions de réseaux: on est toujours obligé d'avoir un paramètre "valeur" pour contrôler les niveaux de la récursion. Nous étudions l'introduction d'un mécanisme permettant la construction dynamique.

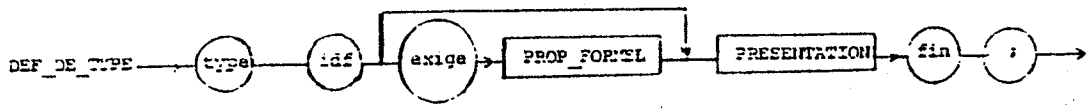
14. Un autre inconvénient de n'avoir que la récursion comme outil de construction est que cela nous oblige à l'utiliser dans des cas où un parcours itératif aurait été suffisant. A ce propos nous envisageons de nous servir dans le futur de constructions standards comme l'abstraction d'ensembles proposée en HOPE et KRC pour éviter la récursion explicite (voir la remarque à la fin de l'exemple No.3 du chapitre III).

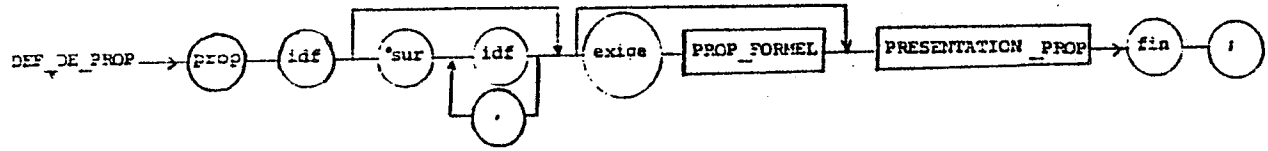
15. Pour finir, nous voulons remarquer que FP2 a été conçu avec une idée centrale: faciliter l'analyse des systèmes communicants, sans négliger pour cela les caractéristiques propres à un langage de programmation. Ainsi, nous exprimons le comportement au moyen d'un ensemble de règles de réécriture dont le mécanisme est purement applicatif. Un travail qui permet de décider de la divergence d'une règle est présenté en <Per 84>. D'autres sur la relation "d'implémentation correcte" et l'équivalence comportementale sont en cours.

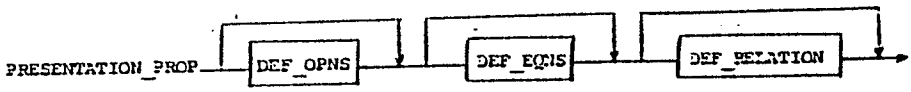
CARTE SYNTAXIQUE DE LPG

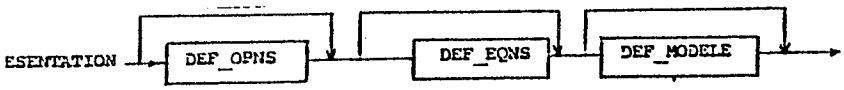


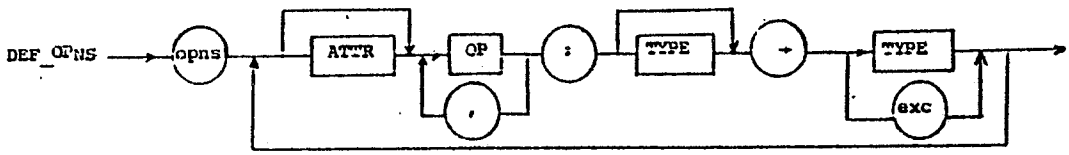


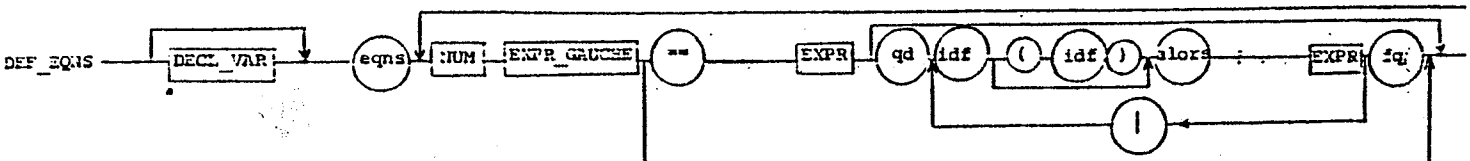


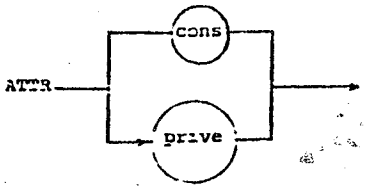


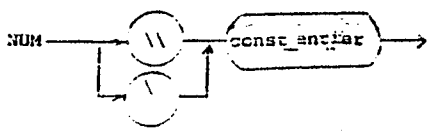


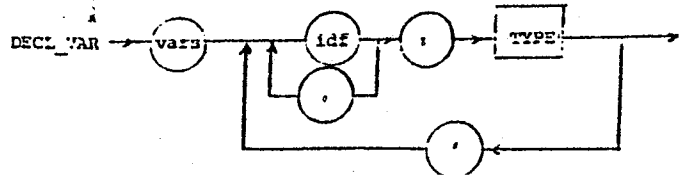


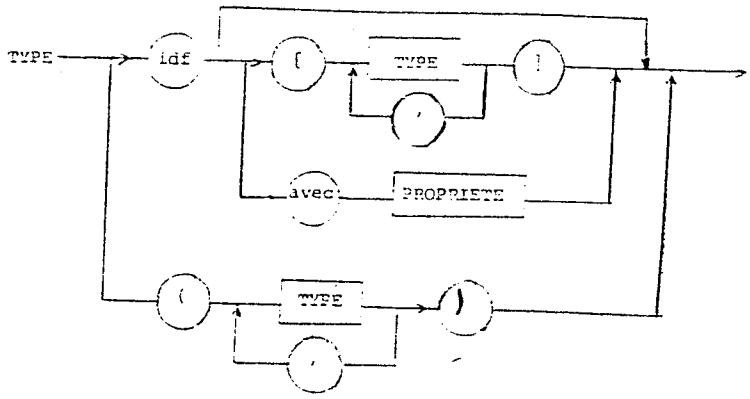


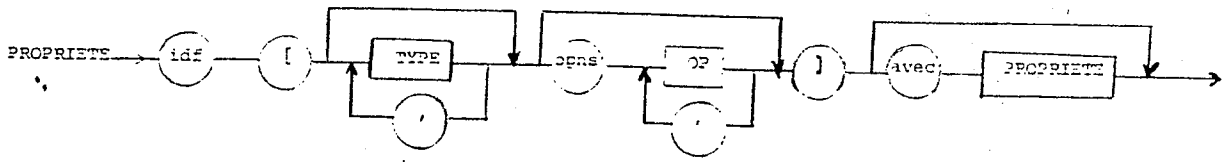


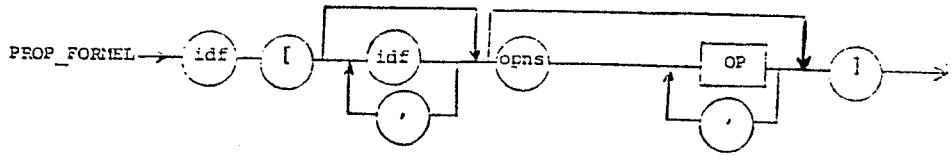


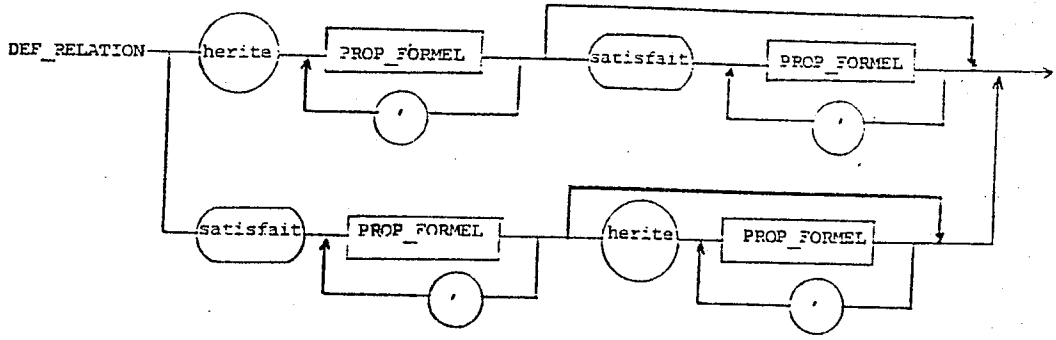


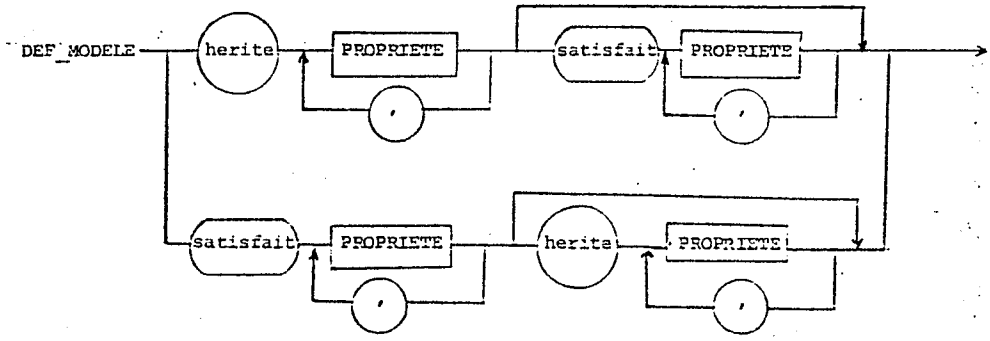


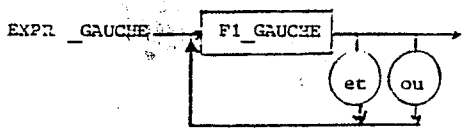

TYPE
idf
(
TYPE
)
,
avec
PROPRIETE
(
TYPE
)
,

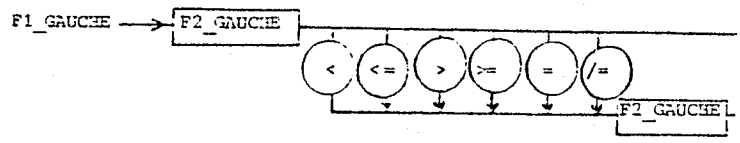

PROPRIETE
idf
[
TYPE
,
opns
OP
,
]
avec
PROPRIETE

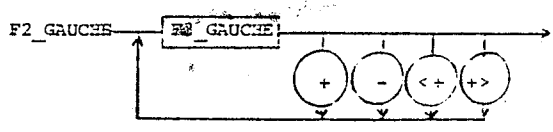

PROP_FORMEL
idf
[
idf
,
opns
OP
,
]

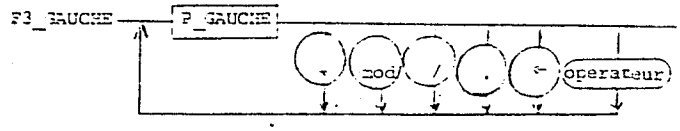

DEF_RELATION
herite
PROP_FORMEL
,
satisfait
PROP_FORMEL
,
satisfait
PROP_FORMEL
,
herite
PROP_FORMEL
,

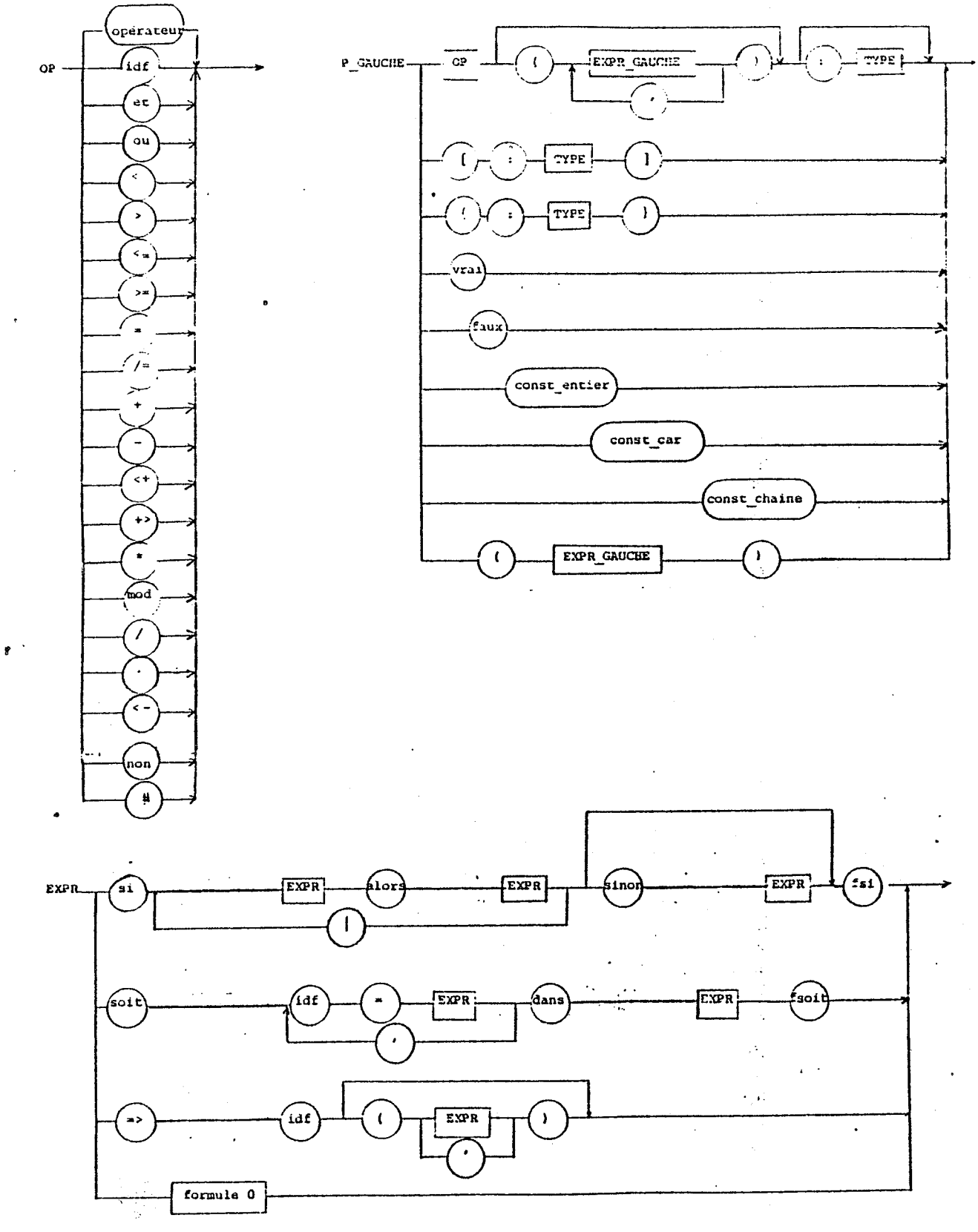

DEF_MODELE
herite
PROPRIETE
,
satisfait
PROPRIETE
,
satisfait
PROPRIETE
,
herite
PROPRIETE
,

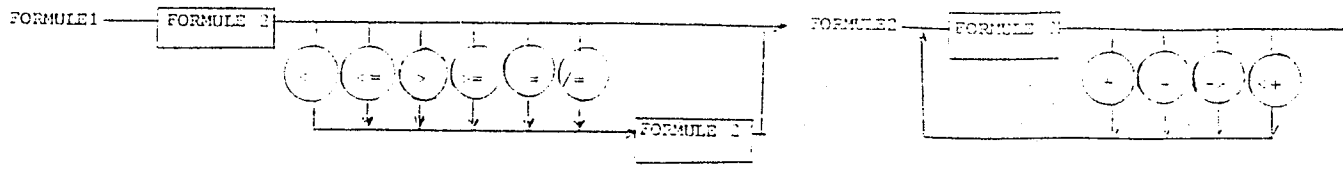

EXPR_GAUCHE
F1_GAUCHE
et
ou

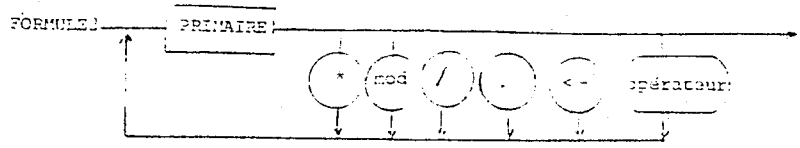

F1_GAUCHE
F2_GAUCHE
<
<=
>
>=
=
/=
F2_GAUCHE

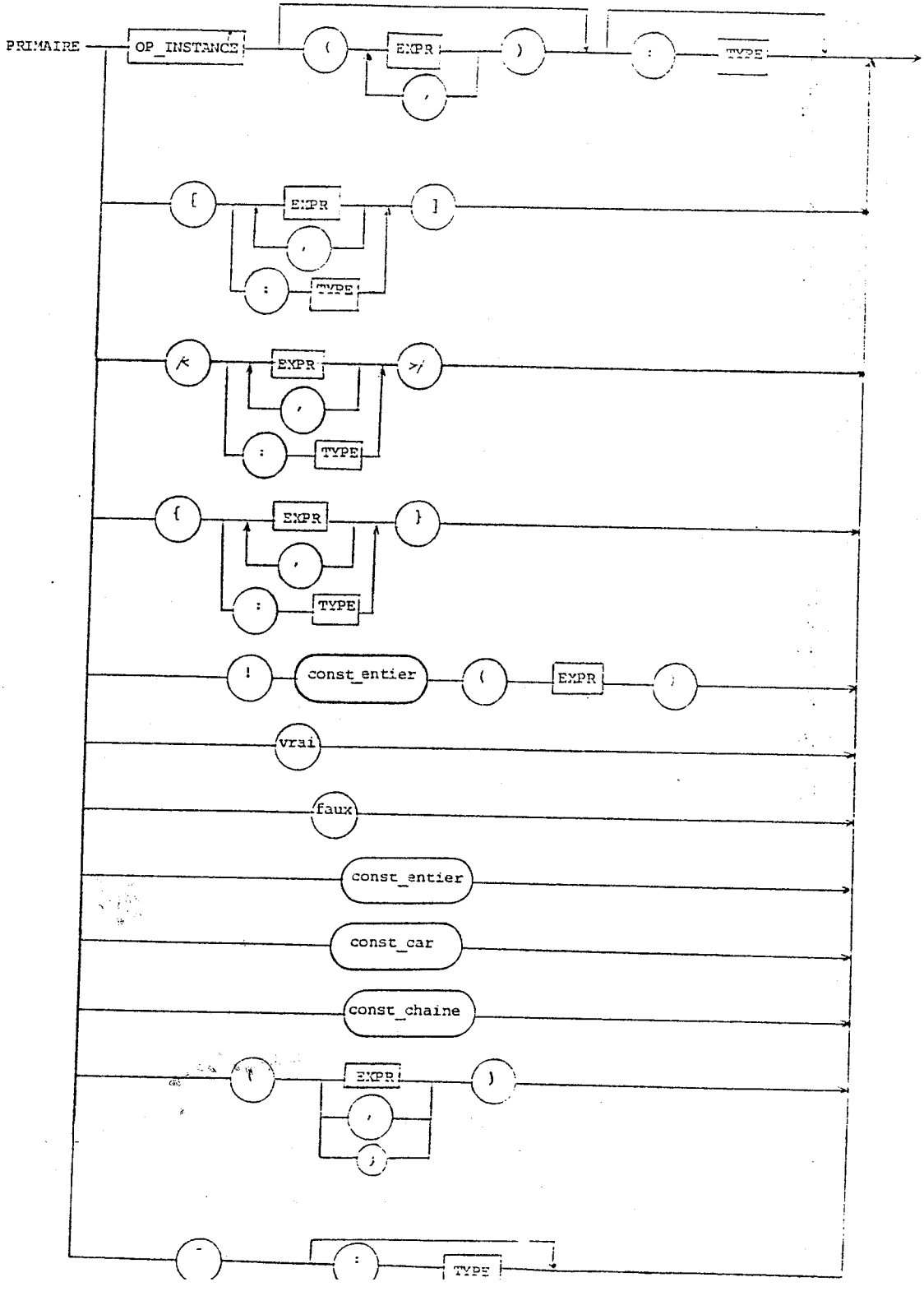

F2_GAUCHE
F2_GAUCHE
+
-
<+
+>

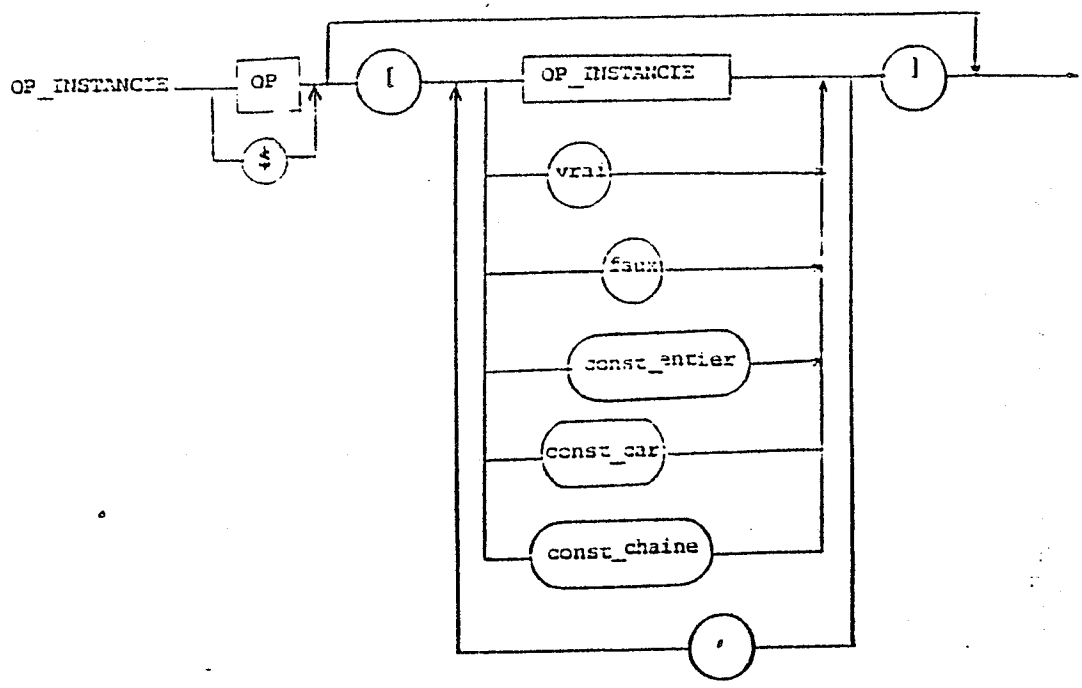

F3_GAUCHE
F_GAUCHE
*
mod
/
.
^
operateur


OP
opérateur
idf
et
ou
<
>
<=
>=
=
/=
+
-
<+
+>
*
mod
/
.
<-
non
#
P_GAUCHE
OP
(
EXPR_GAUCHE
)
,
:
TYPE
[
]
vrai
faux
const_entier
const_car
const_chaine
EXPR
si
alors
sinon
fsi
soit
dans
fsoit
=>
formule 0
FORMULE0
FORMULE 1


FORMULE1
FORMULE 2
<
<=
>
>=
=
/=
FORMULE 2
FORMULE2
FORMULE 3
+
-
-.
+.


FORMULE3
PRIMAIRE
*
mod
/
.
<-
opérateur


PRIMAIRE
OP_INSTANCE
(
EXPR
,
)
:
TYPE
[
EXPR
,
:
TYPE
]
/<
EXPR
,
:
TYPE
>/
{
EXPR
,
:
TYPE
}
!
const_entier
(
EXPR
)
vrai
faux
const_entier
const_car
const_chaine
(
EXPR
,
)
-
:
TYPE


OP_INSTANCIE
OP
$
[
OP_INSTANCIE
]
vrai
faux
const_entier
const_car
const_chaine
,

BIBLIOGRAPHIE

<ADA 82>
Dominique Le VERRAND
"Le langage ADA, manuel d'évaluation". Publié sous l'égide de l'AFCET avec le concours de l'Agence de l'Informatique, 1982.

<Ack-Den 79>
W.B. ACKERMAN et J.B. DENNIS
"VAL - preliminary reference manual". MIT Laboratory for Computer Science, 1979.

<Ark 84>
Equerem ARKAXHIU
"Un environnement et un Langage Graphique pour la Spécification de Processus Parallèles Comunicants". Thèse de 3ème. cycle, juillet 1984, INPG.

<Arv-Gos 78>
ARVIND, K.P. GOSTELOW et W. PLOUFFE
"An Asynchronous Programming Language and Computing Machine". University of California at Irvine, 1978.

<Aub 77>
R. AUBIN
"Strategies for Mechanizing Structural Induction". Proc. 5th Int. Joint Conf. on Artificial Intelligence, Cambridge, Massachusets, Août 1977.

<Bac 78>
John BACKUS
"Can Programming be Liberated from the Von Neuman Style? A Functional Style and its Algebra of the Programs". Communications of the ACM, Août 1978, Vol.21, No.8.

<Ber 79>
Didier BERT
"La programmation générique. Construction de logiciel, spécification algébrique et vérification". Thèse Docteur es Sciences, Université de Grenoble, 1979

<Ber 82a>
Didier BERT
"Generic Programming: A tool for designing Universal Operators. Application to Program Algebra". R.R.No.336, IMAG Novembre 1982.

<Ber 82b>
Didier BERT
"Refinements of Generic Specifications with algebraic tools". R.R.No.335, IMAG Novembre 1982.

<Ber 83>
Didier BERT
"Manuel de Référence de LPG, Version 1.2". R.R.No.408, IMAG Decembre 1983.

<Boy-Moo 77>
R.S. BOYER et J.S. MOORE
"A lemma Driven Automatic Theorem Prover for Recursive Function Theory". Proc. 5th Int. Joint Conf. on Artificial Intelligence, Cambridge, Massachusetts, Août 1977.

<Bur 81>
Rod BURSTALL
"An informal Introduction to Specifications Using CLEAR". International Summer School Theoretical Foundations of Programming Methodology, Munich 1981.

<Bur-Dar 77>
R.M. BURSTALL et J. DARLINGTON
"A transformation System for Developing Recursive Programs". Journal of ACM 24, 1, Janvier 1977.

<Bur-Gog 81>
Rod BURSTALL, J.A. GOGUEN
"Algebras et Theories: An Introduction for Computer Scientist". International Summer School Theoretical Foundations of Programming Methodology, Munich 1981.

<Bur-Mac 81>
Rod M.BURSTALL, D.B. MacQUEEN, D.T. SANNELLA.
"HOPE: an experimental applicative language". Internal Report CSR-62-80, University of Edimburgh, 1981.

<Cam 74>
R. CAMPBELL et A. HABERMANN
"The specification of process synchronization by Path Expressions". LNCS 16, 1974.

<Chu 41>
A. CHURCH
"The Calculi of Lambda-Conversion". Princeton U. Press, Princeton, New Jersey, 1941.

<Cla-Gla 80>
J.W. CLARKE, P.J.S. GLADSTONE, C.D. MacLEAN, A.C. NORMAN
"SKIM - the S, K, I reduction machine". Proceedings LISP conference, Standford 1980.

<Cla-Gre 83>
K.L. CLARK, S. GREGORY
"PARLOG: a parallel logic programming language". Research report DOC-83/5, Imperial College, London 1983.

<Col-Kan 83>
A. COLMERAUER, H. KANOUI, M. VAN CANEGHEM
"PROLOG, bases theoriques et developpements actuels". Techniques et Sciencies Informatiques, Vol.2, No.4, 1983.

<Cis 81>
Maria del Pilar CISNEROS
"Spécification de Processus Génériques". Rapport de DEA, Octobre 1981, INPG.

<COR 81>
CORNAFION
"Systèmes informatiques repartis. Concepts et techniques". Chapitre 4, pp. 99, Ed. DUNOD, 1981.

<Cur 58>
H.B. CURRY et R. FEYS
"Combinatory Logic". Vol.1, North Holland Pub. Co., Amsterdam, 1958.

<Dar-Her 81>
J. DARLINGTON, P. HENDERSON et D.A. TURNER
"Lectures Notes of SRC/CREST". 1981 Summer Course on Functional Programming at a Newcastle University. Publié par Cambridge University Press.

<Den 79>
J.B. DENNIS
"The varieties of Data Flow Computers". MIT Computation Structures Group, Memo 183, 1979.

<Fer 83>
Jean Claude FERNANDEZ
"Structures Mathématiques pour l'étude du Parallélisme". R.R.390, IMAG 1983.

<Gen 80>
H.J. GENRICH, K.LAUTENBACH et P.S. THIAGARAJAN
"An overview of Net Theory". Proc. Advanced Course on General Net Theory of Processes and Systems, LNCS, 1980.

<Gog 74>
J.A. GOGUEN
"On Homomorphisms, Correctness, Termination, Undfoldment and Equivalence of Flow Diagram Programs". Journal of Commputer and System Sciences, No.8, PP.333-365, 1974.

<Gog-Bur 81>
J.A.GOGUEN, R.M. BURSTALL
"The Semantics of CLEAR, a specification langage". International Summer School Theoretical Foundations of Programming Methodology, Munich, 1981.

<Gur-Wat 80>
J. GURD et I. WATSON
"A Data Driven System for High Speed Parallel Computing". Computer Design, 1980.

<Hal 79>
C. HEWITT, G. ATTARDI et H. LIEBERMAN
"Specifiying and proving properties of guardians for distributed systems". LNCS 70, 1979.

<Hen-Mor 76>
P. HENDERSON et J. MORRIS
"A lazy evaluator". Proc. 3rd ACM Symp. on Principles of Programming Languages, Atlanta, Georgia, 1976.

<Hoa 78>
C.A.R. HOARE
"Communicating sequential processes". Comm. ACM, Vol.21, No.8, 1978.

<Hoa 80>
C.A.R. HOARE
"A model for communicating sequential processes". PRG-22, Oxford, 1980.

<Hoa 81a>
C.A.R. HOARE et Al.
"A theory of communicating sequential processes". PRG-16, Oxford, 1981.

<Hoa 81b>
"A calculus of total correctness for communicating processes". PRG-23, Oxford, 1981.

<Jac 78>
Paul JACQUET
"Les types génériques. Propositions pour un mecanisme d'abstraction dans les langages de programmation". Thèse de 3ème. cycle, Université de Grenoble I.

<Jor 81a>
Philippe JORRAND
"Bases for the Specification of Communicating Processes". ICS 81, London, April 1981.

<Jor 81b>
Philippe JORRAND
"Specification of Communicating Processes and Process Implementation Correctness". International Symposium on Programming, 5th.Colloquium, Turin, April 1982, Proceedings LNCS 137.

<Jor 83>
Philippe JORRAND
"Projet SPARC". Non publié.

<Jor 84a>
Philippe JORRAND
"Communicating Processes as Term Algebres". A paraître.

<Jor 84b>
Philippe JORRAND
"FP2 : Functional Parallel Programming based on Term Substitution". LIFIA, à paraître.

<Kah 77>
G. KAHN et D. MacQUEEN
"Coroutines and networks of parallel processes". Proc. IFIP Congress, North Holland, 1977.

<Kun 80>
H.T. KUNG
"The structure of parallel algorithms". Advances in Computers, Vol.19, pp.65-112.

<Kun 81>
H.T. KUNG
"Why Systolic Architectures?". Departement of Computer Science, Carnegie-Mellon University.

<Lis-Zil 77>
Barbara LISKOV, Stephen ZILLES
"An introduction to Formal Specifications of Data Abstractions". Currents Trents in Programming Methodology, Vol.I, Software Specification and Design, Ed. R.T. Yeh Prentice Hall, Inc.

<Mag 79>
A. MAGGIOLO-SCHETTINI, H. WEDDE et J. WINKOWSKI
"Modelling a Solution for a control problem in distributed systems by restrictions". LNCS 70, 1979.

<Man 73>
Z. MANNA, S. NESS et J. VUILLEMIN
"Inductive methods for proving properties of programs". Comm. ACM 16, 8, Août 1973.

<May 83>
D. MAY
"OCCAM". SIGPLAN Notices, Vol.13, No.4, 1983.

<Mcc 60>
John McCARTHY
"Recursive functions of Symbolic Expressions and their Computation by Machine". Communications of the ACM, Avril 1960.

<Mcc 62>
John McCARTHY, P.ABRAHAMS, D.EDWARDS, T.HART et M.LEVIN
"LISP 1.5 Programmers Manual". MIT Computation Structures Group, Memo 183, Août 1979.

<Mil 79>
G. MILNE et R. MILNER
"Concurrent processes and their syntax". Journal of ACM, 26, 2, 1979.

<Mil 80>
Robin MILNER
"A calculus of communicating systems". Lecture Notes in Computer Science, No.92, 1980.

<Mil 82>
Robin MILNER
"Calculi for Synchrony and Asynchrony". Theoretical Computer Science 25, 1983, pp.267-310, North Holland.

<Mcj 75>
P. McJONES
"A Church-Rosser property of closed applicative languages". Rep. RJ 1589, IBM Res. Lab., San Jose, California, Mai 1975.

<Mun 83>
Traian MUNTEAN
"Introduction à OCCAM langage parallèle issu de CSP pour la programmation des systèmes de transinateurs". Laboratoire de Génie Informatique, Décembre 1983.

<Per 84>
Juan Manuel PEREIRA
"Processus communicants: un langage et ses modèles. Problèmes d'analyse". Thèse 3ème. cycle, LIFIA, INPG, 1984.

<Pet 80>
C.A. PETRI
"Introduction to General Net Theory". Proc. Advanced Course on General Net Theory of Processes and Systems, LNCS 1980.

<Rob-Sib 82>
J.A. ROBINSON, E.E. SIBERT
"LOGLISP: motivations, design and implementation". Logic Programming, Academic Press, 1982.

<Rob-Tch 81>
Y. ROBERT, M. TCHUENTE
"Calcul en parallèle sur des réseaux systoliques". Séminaire d'analyse numérique, IMAG, 5 Novembre 1981, Grenoble. Rapport No.368.

<Sco 72>
D. SCOTT
"Lattice-theoretic models for various type free calculi". Proc. Fourth Int. Congress for Logic, Methodology and the Philosophy of Science, Bucharest, 1972.

<Sha 83>
E.Y. SHAPIRO
"A subset of Concurrent PROLOG and its interpreter". Technical Report TR-003, ICOT Tokyo, 1983.

<Tur 76>
D.A. TURNER
"SASL Language Manual". St. Andrews University, 1976.

<Tur 81a>
D.A. TURNER
"The semantic elegance of applicative languages". ACM Conference on Functional Programming Languages, 1981.

<Tur 81b>
David TURNER
"The future of applicative programming". LNCS, Vol 123, Proceedings of 3rd. European Conference in Informatics, Munich, Octobre 1981.

<Tur 81c>
D.A. TURNER
"Functional Programming and Proof of Program Correctness". Rapport interne, Université de Kent, Canterbury, Angleterre.

<Turner 81d>
D.A. TURNER
"Aspects of the Implementation of Programming Languages". Oxford University D. Phil. Thesis, Fevrier 1981.

AUTORISATION de SOUTENANCE

VU les dispositions de l'article 3 de l'arrêté du 16 avril 1974,

VU les rapports de présentation de Monsieur JORRAND, Directeur de recherche

Mademoiselle CISNEROS GASCON Maria del Pilar

est autorisée à présenter une thèse en soutenance en vue de l'obtention du titre de DOCTEUR DE TROISIEME CYCLE, spécialité "Informatique".

Fait à Grenoble, le 15 octobre 1984

Le Président de l'I.N.P.-G

D. BLOCH
Président
de l'Institut National Polytechnique
de Grenoble

P.O. le Vice-Président,