50376
1991
128

50376
1991
128

USTL
FLANDRES ARTOIS

LABORATOIRE D'INFORMATIQUE FONDAMENTALE DE LILLE

N° d'ordre : 694

THESE

Nouveau régime

présentée à

l'Université des Sciences et Techniques de Lille Flandres Artois

pour obtenir le titre de

DOCTEUR en INFORMATIQUE

par

Isabelle DELIGNIERES-HANNEQUIN

PROPOSITION D'UN MODELE D'EVALUATION PARALLELE DE PROLOG

Soutenue le vendredi 8 Fevrier 1991 devant la commission d'examen

Membres du jury

Président	M. MERIAUX
Rapporteurs	M. TREHEL
	J.P. DELAHAYE
Directeur de thèse	B. TOURSEL
Examinateurs	G. GONCALVES
	P. LECOUFFE
	P. DEVIENNE

UNIVERSITE DES SCIENCES
ET TECHNIQUES DE LILLE
FLANDRES ARTOIS

DOYENS HONORAIRES DE L'ANCIENNE FACULTE DES SCIENCES

M.H. LEFEBVRE, M. PARREAU.

PROFESSEURS HONORAIRES DES ANCIENNES FACULTES DE DROIT ET SCIENCES ECONOMIQUES, DES SCIENCES ET DES LETTRES

MM. ARNOULT, BONTE, BROCHARD, CHAPPELON, CHAUDRON, CORDONNIER, DECUYPER, DEHEUVELS, DEHORS, DION, FAUVEL, FLEURY, GERMAIN, GLACET, GONTIER, KOURGANOFF, LAMOTTE, LASSERRE, LELONG, LHOMME, LIEBAERT, MARTINOT-LAGARDE, MAZET, MICHEL, PEREZ, ROIG, ROSEAU, ROUELLE, SCHILTZ, SAVARD, ZAMANSKI, Mes BEAUJEU, LELONG.

PROFESSEUR EMERITE

M. A. LEBRUN

ANCIENS PRESIDENTS DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE

MM. M. PAREAU, J. LOMBARD, M. MIGEON, J. CORTOIS.

PRESIDENT DE L'UNIVERSITE DES SCIENCES ET TECHNIQUES DE LILLE FLANDRES ARTOIS

M. A. DUBRULLE.

PROFESSEURS - CLASSE EXCEPTIONNELLE

M. CONSTANT Eugène	Electronique
M. FOURET René	Physique du solide
M. GABILLARD Robert	Electronique
M. MONTREUIL Jean	Biochimie
M. PARREAU Michel	Analyse
M. TRIDOT Gabriel	Chimie Appliquée

PROFESSEURS - 1ère CLASSE

M. BACCHUS Pierre	Astronomie
M. BIAYS Pierre	Géographie
M. BILLARD Jean	Physique du Solide
M. BOILLY Bénoni	Biologie
M. BONNELLE Jean-Pierre	Chimie-Physique
M. BOSCQ Denis	Probabilitéss
M. BOUGHON Pierre	Algèbre
M. BOURIQUET Robert	Biologie Végétale
M. BREZINSKI Claude	Analyse Numérique

M. BRIDOUX Michel	Chimie-Physique
M. CELET Paul	Géologie Générale
M. CHAMLEY Hervé	Géotechnique
M. COEURE Gérard	Analyse
M. CORDONNIER Vincent	Informatique
M. DAUCHET Max	Informatique
M. DEBOURSE Jean-Pierre	Gestion des Entreprises
M. DHAINAUT André	Biologie Animale
M. DOUKHAN Jean-Claude	Physique du Solide
M. DYMENT Arthur	Mécanique
M. ESCAIG Bertrand	Physique du Solide
M. FAURE Robert	Mécanique
M. FOCT Jacques	Métallurgie
M. FRONTIER Serge	Ecologie Numérique
M. GRANELLE Jean-Jacques	Sciences Economiques
M. GRUSON Laurent	Algèbre
M. GUILLAUME Jean	Microbiologie
M. HECTOR Joseph	Géométrie
M. LABLACHE-COMBIER Alain	Chimie Organique
M. LACOSTE Louis	Biologie Végétale
M. LAVEINE Jean-Pierre	Paléontologie
M. LEHMANN Daniel	Géométrie
Mme LENOBLE Jacqueline	Physique Atomique et Moléculaire
M. LEROY Jean-Marie	Spectrochimie
M. LHOMME Jean	Chimie Organique Biologique
M. LOMBARD Jacques	Sociologie
M. LOUCHEUX Claude	Chimie Physique
M. LUCQUIN Michel	Chimie Physique
M. MACKE Bruno	Physique Moléculaire et Rayonnements Atmosph.
M. MIGEON Michel	E.U.D.I.L.
M. PAQUET Jacques	Géologie Générale
M. PETIT Francis	Chimie Organique
M. POUZET Pierre	Modélisation - calcul Scientifique
M. PROUVOST Jean	Minéralogie
M. RACZY Ladislas	Electronique
M. SALMER Georges	Electronique
M. SCHAMPS Joel	Spectroscopie Moléculaire
M. SEGUIER Guy	Electrotechnique
M. SIMON Michel	Sociologie
Melle SPIK Geneviève	Biochimie
M. STANKIEWICZ François	Sciences Economiques
M. TILLIEU Jacques	Physique Théorique
M. TOULOTTE Jean-Marc	Automatique
M. VIDAL Pierre	Automatique
M. ZEYTOUNIAN Radyadour	Mécanique

PROFESSEURS - 2ème CLASSE

M. ALLAMANDO Etienne	Composants Electroniques
M. ANDRIES Jean-Claude	Biologie des organismes
M. ANTOINE Philippe	Analyse
M. BART André	Biologie animale
M. BASSERY Louis	Génie des Procédés et Réactions Chimiques

Mme BATTIAU Yvonne	Géographie
M. BEGUIN Paul	Mécanique
M. BELLET Jean	Physique Atomique et Moléculaire
M. BERTRAND Hugues	Sciences Economiques et Sociales
M. BERZIN Robert	Analyse
M.BKOUCHE Rudolphe	Algèbre
M. BODARD Marcel	Biologie Végétale
M. BOIS Pierre	Mécanique
M. BOISSIER Daniel	Génie Civil
M. BOIVIN Jean-Claude	Spectroscopie
M. BOUQUELET Stéphane	Biologie Appliquée aux enzymes
M. BOUQUIN Henri	Gestion
M. BRASSELET Jean-Paul	Géométrie et Topologie
M. BRUYELLE Pierre	Géographie
M. CAPURON Alfred	Biologie Animale
M. CATTEAU Jean-Pierre	Chimie Organique
M. CAYATTE Jean-Louis	Sciences Economiques
M. CHAPOTON Alain	Electronique
M. CHARET Pierre	Biochimie Structurale
M. CHIVE Maurice	Composants Electroniques Optiques
M. COMYN Gérard	Informatique Théorique
M. COQUERY Jean-Marie	Psychophysiologie
M. CORIAT Benjamin	Sciences Economiques et Sociales
Mme CORSIN Paule	Paléontologie
M. CORTOIS Jean	Physique Nucléaire et Corpusculaire
M. COUTURIER Daniel	Chimie Organique
M. CRAMPON Norbert	Tectolique Géodynamique
M. CROSNIER Yves	Electronique
M. CURGY Jean-Jacques	Biologie
Melle DACHARRY Monique	Géographie
M. DEBRABANT Pierre	Géologie Appliquée
M. DEGAUQUE Pierre	Electronique
M. DEJAEGER Roger	Electrochimie et Cinétique
M. DELAHAYE Jean-Paul	Informatique
M. DELORME Pierre	Physioliogie Animale
M. DELORME Robert	Sciences Economiques
M. DEMUNTER Paul	Sociologie
M. DENEL Jacques	Informatique
M. DE PARIS Jean Claude	Analyse
M. DEPREZ Gilbert	Physique du Solide - Cristallographie
M. DERIEUX Jean-Claude	Microbiologie
Melle DESSAUX Odile	Spectroscopie de la réactivité Chimique
M. DEVRAINNE Pierre	Chimie Minérale
Mme DHAINAUT Nicole	Biologie Animale
M. DHAMELINCOURT Paul	Chimie Physique
M. DORMARD Serge	Sciences Economiques
M. DUBOIS Henri	Spectroscopie Hertzienne
M. DUBRULLE Alain	Spectroscopie Hertzienne
M. DUBUS Jean-Paul	Spectrométrie des Solides
M. DUPONT Christophe	Vie de la firme (I.A.E.)
Mme EVRARD Micheline	Génie des procédés et réactions chimiques
M. FAKIR Sabah	Algèbre
M. FAUQUAMBERGUE Renaud	Composants électroniques

M. FONTAINE Hubert	Dynamique des cristaux
M. FOUQUART Yves	Optique atmosphérique
M. FOURNET Bernard	Biochimie Sturcturale
M. GAMBLIN André	Géographie urbaine, industrielle et démog.
M. GLORIEUX Pierre	Physique moléculaire et rayonnements Atmos.
M. GOBLOT Rémi	Algèbre
M. GOSSELIN Gabriel	Sociologie
M. GOUDMAND Pierre	Chimie Physique
M. GOURIEROUX Christian	Probabilités et Statistiques
M. GREGORY Pierre	I.A.E.
M. GREMY Jean-Paul	Sociologie
M. GREVET Patrice	Sciences Economiques
M. GRIMBLOT Jean	Chimie Organique
M. GUILBAULT Pierre	Physiologie animale
M. HENRY Jean-Pierre	Génie Mécanique
M. HERMAN Maurice	Physique spatiale
M. HOUDART René	Physique atomique
M. JACOB Gérard	Informatique
M. JACOB Pierre	Probabilités et Statistiques
M. Jean Raymond	Biologie des populations végétales
M. JOFFRE Patrick	Vie de la firme (I.A.E.)
M. JOURNEL Gérard	Spectroscopie hertzienne
M. KREMBEL Jean	Biochimie
M. LANGRAND Claude	Probabilités et statistiques
M. LATTEUX Michel	Informatique
Mme LECLERCQ Ginette	Catalyse
M. LEFEBVRE Jacques	Physique
M. LEFEBVRE Christian	Pétrologie
Melle LEGRAND Denise	Algèbre
Melle LEGRAND Solange	Algèbre
M. LEGRAND Pierre	Chimie
Mme LEHMANN Josiane	Analyse
M. LEMAIRE Jean	Spectroscopie hertzienne
M. LE MAROIS Henri	Vie de la firme (I.A.E.)
M. LEROY Yves	Composants électroniques
M. LESENNE Jacques	Systèmes électroniques
M. LHENAFF René	Géographie
M. LOCQUENEUX Robert	Physique théorique
M. LOSFELD Joseph	Informatique
M. LOUAGE Francis	Electronique
M. MAHIEU Jean-Marie	Optique-Physique atomique
M. MAIZIERES Christian	Automatique
M. MAURISSON Patrick	Sciences Economiques et Sociales
M. MESMACQUE Gérard	Génie Mécanique
M. MESSELYN Jean	Physique atomique et moléculaire
M. MONTEL Marc	Physique du solide
M. MORCELLET Michel	Chimie Organique
M. MORTREUX André	Chimie Organique
Mme MOUNIER Yvonne	Physiologie des structures contractiles
Mme MOUYART-TASSIN Annie Françoise	Informatique
M. NICOLE Jacques	Spectrochimie
M. NOTELET Françis	Systèmes électroniques
M. PARSY Fernand	Mécanique

M. PECQUE Marcel	Chimie organique
M. PERROT Pierre	Chimie appliquée
M. STEEN Jean-Pierre	Informatique

Je tiens à remercier

Monsieur Michel Mériaux de me faire l'honneur de présider ce jury et d'avoir accepté de remplacer Monsieur Eric Delattre dans des circonstances pénibles,

Monsieur Michel Trehel d'avoir accepté d'en être l'un des rapporteurs,

Monsieur Jean-Paul Delahaye d'avoir accepté de participer à ce jury en tant que rapporteur et d'avoir permis, grâce à ses critiques, d'améliorer ce document,

Monsieur Bernard Toursel d'avoir été l'initiateur de ce travail et d'en avoir suivi la progression, la favorisant par ses conseils,

Monsieur Gilles Goncalves de m'avoir suivie tout au long de ce travail, et de m'avoir apporté son soutien et sa grande disponibilité,

Monsieur Philippe Devienne et Monsieur Pierre Lecouffe de me faire l'amitié d'en être examinateurs.

Je remercie également les membres de l'équipe LOG_ARCH ainsi que mes collègues de l'EUDIL avec qui, il a été très agréable de travailler.

Je remercie tout particulièrement Monsieur Christian Dorémus sans qui la réalisation de ce document n'aurait pu être assurée.

Merci enfin à Monsieur Glanc qui en a assuré la reproduction.

INTRODUCTION **1**

PROGRAMMATION LOGIQUE

I. INTRODUCTION 8

II. PROLOG SEQUENTIEL. 8

II.1. Définitions. 8

II.2. L'unification 9
II.2.a. Définition. 9
II.2.b. Algorithme d'unification de Robinson. 9
II.2.c. Algorithme d'unification de Fages. 11

II.3. Le mécanisme de résolution 13

II.4. Evaluation de Prolog avec trois piles 19

III. PROGRAMMATION LOGIQUE ET PARALLELISME 22

III.1. Sources de parallélisme en programmation 22

III.2. Le parallélisme en programmation logique avantages et inconvénients 22
III.2.a. Le parallélisme OU 22
III.2.b. Le parallélisme ET 23
III.2.c. Le parallélisme de flot. 23

III.3 Modèles. 24
III.3.a. Modèles exploitant le parallélisme de flots. 24
A. Les langages gardés 24
B. Mise en œuvre des langages gardés. 27
III.3.b. Les modèles non déterministes. 28
A. Les modèles basés sur l'arbre ET/OU. 28
a) Le modèle de Conery 28
b) Le modèle COALA 30
B. Modèles Multiséquentiels 32
a) Le modèle SRI 33
c) Le modèle OPERA 36

IV. CONCLUSION 38

LE MODELE D'EVALUATION

I. INTRODUCTION....40

II. CARACTERISTIQUES DU MODELE....40

II.1 Le parallélisme OU restreint (DFOP)....40

II.2 Le pipeline ET....41

II.3 Le contrôle dirigé par la nécessité....42
II.3.a L'activation des nœuds....42
II.3.b Consommation des environnements retour....42
II.3.b.1 Le mécanisme d'activation restreinte du parallélisme OU....42
II.3.b.2 Le pipeline ET....45

III. LE MODELE FONCTIONNEL....48

III.1. La compilation du programme source....48

III.2. Primitives de synchronisation du modèle....49
III.2.a. Les primitives d'activation....50
III.2.b. La primitive demande de solutions....50
III.2.c. Les primitives résultats....51
III.2.d. La primitive de destruction....52

III.3. Le gérant d'un nœud....53

IV. CONCLUSION....55

LE GERANT D'UN NŒUD OU

I. INTRODUCTION....57

II. LA PRIMITIVE P_ACT....57

III. LA PRIMITIVE P_LOCK....58

IV. LA PRIMITIVE P_UNLOCK....59

IV.1. L'état du nœud est Attente_Double....60

IV. 2. L' état du nœud est Attente_Solution....60
IV.2.a. Le buffer d'environnements du nœud contient un environnement-échec....60
IV.2.b.La solution mémorisée n'est pas un échec....61

V. LA PRIMITIVE P_RES 63
V.1. L'état du nœud est Attente_Solution. 64
V.1.a. La table de liens est vide. 64
V.1.b. Un fils du nœud est encore actif. 64
V.2.Le nœud est en Attente_Double. 70
V.2.a. L'activité de tous les nœuds fils est nulle. 70
V.2.b. Un des fils du nœud est encore actif. 71
VI. LA PRIMITIVE P_ECHEC 74
VI.1.Le nœud fils responsable est le dernier fils du nœud. 75
VI.1.a. Le nœud est en Attente_Solution. 75
VI.1.b. Le nœud est en Attente_Double 76
VI.2. Un nœud fils est susceptible de transmettre une autre solution 76
VII. LA PRIMITIVE P_DEL 78
VIII. CONCLUSION 79

LE GERANT D'UN NŒUD ET

I. INTRODUCTION 80
II. LA PRIMITIVE P_ACT 80
II.1. L'unification échoue. 80
II.2. L'unification est un succès et le nœud n'a pas de fils (N1=0). 81
II.3. L'unification est un succès et le nœud possède au moins un fils. 83
III. LA PRIMITIVE P_LOCK 84
III.1. L'unification échoue. 84
III.2. L'unification est un succès et le nœud n'a pas de fils 84
III.3. L'unification est un succès et le nœud possède au moins un fils (N1 >=1). 85
IV. LA PRIMITIVE P_UNLOCK 86
IV.1. Un échec est mémorisé dans le buffer d' environnements 86
IV.2. L'environnement mémorisé n'est pas un échec. 87
IV.3. Le buffer d'environnements est vide. 90

V. LA PRIMITIVE P_RES.....90
V.1. L'émetteur de la primitive est le dernier fils du nœud.....91
V.1.a. L'attribut dernier de la primitive est vrai.....91
V.1.b. L'attribut dernier n'est pas positionné.....96
V.2. L'émetteur de la primitive n'est pas le dernier fils du nœud.....99
V.2.a. L'attribut dernier de la primitive est vrai.....99
V.2.b. L'attribut dernier de la primitive n'est pas vrai.....103
VI. LA PRIMITIVE P_ECHEC.....105
VI.1. L'échec obtenu est définitif.....106
VI.2. L'échec obtenu n'est pas définitif.....108
VII. LA PRIMITIVE P_DEL.....110
VIII. CONCLUSION.....110

LES PREDICATS A EFFET DE BORD

I. INTRODUCTION.....112
II. LE PREDICAT FAIL.....112
II.1. Rôle du prédicat FAIL.....112
II.2. Implantation en terme de primitives.....112
III. LE PREDICAT CUT.....113
III.1. Rôle du prédicat CUT.....113
III.2. Le CUT face au parallélisme du modèle solution retenue.....114
III.2.a. La destruction d'une arborescence principe général.....115
III.2.b. Les primitives supplémentaires nécessaires à la gestion du CUT.....116
III.2.c. Les états supplémentaires nécessaires à la gestion du CUT.....117
III.2.d. La primitive P_KILL.....118
III.2.e. La primitive P_ACK_KILL.....120
III.2.e.1. L'état du nœud est Suicide.....120
III.2.e.2. Le noeud est le noeud père du CUT.....121
III.2.f. La primitive P_CUT(Fk, Nœud).....124
III.2.g. La primitive P_UNLOCK.....130
III.2.h. La primitive P_RES.....131

III.2.i. La primitive P_ECHEC 138
III.2.j. Etude du comportement d'un nœud ET, père d'un CUT et activé en mode restreint 141
III.3. Mise en œuvre du CUT dans le modèle généralisé 144
IV. LES PREDICATS DE LECTURE ET D'ECRITURE 147
IV.1. Principes de la mise en œuvre des prédicats de lecture et d'écriture dans le modèle 147
IV.2. Implantation en terme de primitives 148
IV.2.a. Calcul de la valeur du booléen en fonction d'une primitive reçue par le nœud 148
IV.2.b Calcul de la valeur du booléen consécutif à l'émission d'une primitive par le nœud 150
V. CONCLUSION 150

SIMULATION

I.INTRODUCTION 152
II. SIMULATION ABSTRAITE DU MODELE 152
II.1. Limitations 152
II.2. Présentation du simulateur 153
II.2.a. Paramètres d'entrée 153
II.2.b. Paramètres de sortie 153
II.2.c. Architecture du simulateur 154
II.2.c.1. Architecture fonctionnelle du simulateur 154
II.2.c.2. Structures de données 156
II.3. Jeux d'essai 156
II.3.a. Les résultats obtenus sur la classe 1 157
II.3.b. Les résultats obtenus sur la classe 2 158
II.3.c. Les résultats obtenus sur la classe 3 159
II.3.c.1. Le nombre de nœuds varie 159
II.3.c.2. Le nombre de niveaux varie 160
III.1. Principe 162
III.2. Présentation informelle du calcul des triplets 162
III.2.a. Calcul de l'argument CID 162
III.2.b. Calcul de l'argument ORD 162
III.2.c. Calcul de l'argument NOD 163

III.3. Résultats..163
III.3.a. Etude des programmes de classe 2.........................164
III.3.b. Etude des programmes de classe 3.........................165

IV. CONCLUSION..167

CONCLUSION..168

REFERENCES BIBLIOGRAPHIQUES..171

INTRODUCTION

INTRODUCTION

L'évolution de la technologie a permis un accroissement important des performances des machines. Cette évolution présente néanmoins des limitations, notamment en ce qui concerne la vitesse de commutation des circuits. Aussi, si l'on veut améliorer la performance des traitements, une solution s'impose : les architectures parallèles. Cependant, écrire un programme destiné à être exécuté sur de telles architectures soulève des problèmes différents de ceux rencontrés lors de la programmation séquentielle. Une étape importante dans le développement de tels programmes réside dans le choix du langage de programmation adapté. Celui-ci doit non seulement avoir une sémantique claire, mais encore être intrinsèquement parallèle.

Les langages impératifs, tels Fortran, Pascal, C, ne répondent qu'imparfaitement au second critère énoncé. En effet, la plupart de ces langages ont été conçus à une époque où les capacités mémoire étaient encore relativement pauvres. Le programmeur, qui utilise l'un de ces langages, indique pas à pas à la machine ce qu'elle doit faire et gère lui-même la mémoire en manipulant des déclarations de types, des instructions de saut, des instructions d'affectation. Un programme rédigé dans l'un de ces langages est une suite séquentielle d'instructions à exécuter. Des tentatives de parallélisation de ces langages ont été effectuées. Ainsi, est né le Fortran vectoriel, dans lequel, les boucles DO sont parallélisées, et qui est implanté sur des calculateurs scientifiques. Néammoins, la machine virtuelle des langages impératifs est toujours d'architecture von Neumann.

Cette classe de langages est mal adaptée aux problèmes d'intelligence artificielle

Les langages de programmation logique sont quant à eux intrinsèquement parallèles. En effet, les langages de programmation logique sont déclaratifs (ils représentent les connaissances sur un sujet donné par un formalisme adéquat). A chaque étape, plusieurs choix sont possibles pour continuer. Une implantation séquentielle implique l'exploration successive de chacune de ces voies, entraînant des retours en arrière.

Une implantation parallèle laisse espérer une exploration simultanée, donc parallèle, de ces différentes possibilités.

Les langages de programmation logique sont en outre très séduisants grâce à un pouvoir d'expression élevé issu des mécanismes de logique mathématique auxquels ils font appel. Leurs applications en intelligence artificielle sont nombreuses. Ainsi, ils peuvent modéliser des connaissances sur les grammaires formelles, ou le traitement des langues naturelles...

L'objectif du projet N-ARCH [Gonc 88a] est la conception d'une machine parallèle, adaptée à l'exécution de programmes déclaratifs, selon des schémas de contrôle non von Neuman. L'architecture de cette machine consiste en un réseau de processeurs, interconnectés de telle manière à ce que tout processeur du réseau puisse être la racine d'un arbre de recouvrement du réseau. Les communications entre processeurs se font par échange de messages, chacun des processeurs possède une mémoire associative. Dans le cadre de ce projet, deux points particuliers ont été développés : l'un est relatif à la distribution de charge des objets dans le réseau [Arna 89], l'autre à l'étude d'une mémoire associative [Gonc 88b]. Dans le cadre du projet N-ARCH, divers travaux ont été réalisés, concernant la réalisation d'un émulateur du réseau N-ARCH sur transputer [Niar 90], l'étude des méthodes de répartition de bases de données dans un réseau [Nico 91], l'étude de nouveaux modèles d'évaluation parallèle de langages déclaratifs. Deux langages en l'occurrence, le langage fonctionnel FP [Deve 90] et le langage Prolog ont plus particulièrement été étudiés. Les travaux relatifs à Prolog ont donné naissance au projet LOG-ARCH.

La réflexion, qui a été menée dans le cadre du projet LOG-ARCH et qui a conduit au modèle d'évaluation présenté dans ce travail, repose sur un modèle sous-jacent de machine. L'architecture de cette machine est basée sur un réseau de processeurs coopérants : chaque processeur du réseau comporte une partie traitement et une partie communication :

- Chaque processeur de traitement dispose d'une mémoire locale et coopère avec les autres processeurs pour évaluer un programme Prolog distribué, par le biais d'émissions et de réceptions de messages.
- La partie communication assure l'émission et la réception de messages ainsi que leur routage via un nombre limité de canaux de communication.

Toutefois, le modèle d'évaluation proposé est plus général et peut être adapté aisément à d'autres types d'architectures.

L'objet de ce travail étant la proposition d'un modèle d'évaluation parallèle de Prolog, nous présenterons dans un premier temps les notions fondamentales de ce langage inventé en 1971, par A. Colmerauer et P. Roussel.

Nous rappelerons ainsi les notions de clauses de Horn, de prédicats, puis décrirons l'algorithme d'unification de Robinson [Robi 65], ainsi que la stratégie de résolution mise en œuvre en Prolog. Nous montrerons ainsi que cette stratégie de résolution aboutit au double choix systématique d'une règle de sélection d'une clause et de celle d'un but. Nous définirons enfin la notion de retour arrière, ou backtracking, et présenterons le parcours d'un arbre ET/OU.

Dans un second temps, nous nous intéresserons aux différentes sources de parallélisme offertes par la programmation logique. Plusieurs études ont permis de dégager les plus importantes d'entre elles qui sont respectivement le parallélisme OU, le parallélisme ET et le parallélisme de flot.

Ces trois sources de parallélisme ont donné à deux classes de modèles : la premiere est celle des systèmes gardés, dont la base est le parallélisme de flot, la seconde est celle des modèles non-déterministes qui exploitent soit le parallélisme OU, soit le parallélisme ET, soit une combinaison des deux.

Les modèles gardés présentent les trois caractéristiques communes suivantes :

- Chaque clause d'un programme contient une garde qui spécifie la (voire les) condition(s) à satisfaire, nécessaire(s) à l'exécution du corps de la clause.
- Le non-déterminisme inhérent à Prolog est abandonné au profit du don't care non-determinism. Le non déterminisme est restreint au choix d'une seule clause parmi un ensemble de clauses qui définissent un prédicat. Ce choix n'est jamais remis en cause (la notion de retour arrière a disparu).
- Les processus sont synchronisés par indication du mode d'accés aux arguments des prédicats.

Les modèles non-déterministes, qui visent à produire l'ensemble des solutions répondant à un problème donné, quant à eux, se divisent en deux sous-classes : les modèles théoriques et les modèles multi-séquentiels. Le modèle de Conery [Cone 81], basé sur l'élaboration dynamique d'un arbre ET/OU appartient à la première de ces sous-classes. Les modèles multi-séquentiels cherchent à contrôler le développement du parallélisme OU qui entraîne la gestion d'un volume important d'informations : à cette fin, ils maximisent le degré de parallélisme en bornant le nombre de processus simultanément créés. La borne, généralement considérée, est le nombre de ressources disponibles.

La présentation du langage Prolog, ainsi que des modèles gardés et des modèles non-déterministes figurent dans le premier chapitre.

Notre attention a été plus particuliérement attirée par le fait que la plupart des modèles parallèles, qu'ils soient gardés ou non déterministes, ne se préoccupent pas de l'ensemble des solutions obtenues suite à l'exécution d'un programme. Ainsi, l'abandon par les langages gardés du non-déterminisme au profit du don't care non-déterminism implique que les ensembles de solutions construits lors des exécutions respectives d'un même algorithme codé d'une part en Prolog, d'autre part dans un des langages gardés diffèrent tant par le nombre de solutions obtenues que par l'ordre de production de ces solutions. Les modèles multi-séquentiels, quant à eux, s'ils visent essentiellement l'efficacité, soit ne traitent pas ce problème, soit le résolvent en introduisant un opérateur explicite (que le programmeur est amené à gérer).

Afin d'éviter ce problème, le modèle proposé, qui repose sur l'élaboration dynamique d'un arbre ET/OU, exploite un parallélisme OU restreint, ainsi qu'un pipeline ET d'activation, assortis d'un mécanisme de verrouillage.

Ainsi, lors de la phase d'activation, un nœud OU n'active que le premier de ses fils. La réception d'une première solution évaluée par ce fils, entraîne l'activation en mode restreint du fils suivant. Ce nœud fils est alors activé, mais la pose d'un verrou de solutions interdit à ce fils toute remontée de solutions vers le nœud OU. Ce dernier lève le verrou de solutions dès la réception de la dernière solution évaluée par le premier fils. Le mécanisme est itéré sur l'ensemble des fils du nœud.

Un nœud ET, quant à lui, dialogue successivement avec l'ensemble de ses fils. Ainsi, il active d'abord son premier fils, puis sur réception d'une première solution évaluée par ce fils, active le fils suivant éventuel. Le mécanisme est itéré jusqu'à ce que le dernier fils soit activé. Chaque retour de solution évaluée par un fils, a entraîné sur ce dernier la pose d'un verrou de solutions qui laisse le nœud actif, mais l'empêche d'envoyer une autre solution au nœud ET. Le nœud ET lève ce verrou dès qu'il peut traiter une nouvelle solution élaborée par le fils concerné.

Ces trois mécanismes (parallélisme OU restreint, pipeline ET et mécanisme de verrouillage) assurent d'une part que le modèle respecte la sémantique opérationnelle de Prolog et d'autre part que l'extraction du parallélisme soit automatique. Le chapitre II détaille chacun de ces mécanismes et introduit le formalisme nécessaire à la description des algorithmes qui gèrent les nœuds OU et les nœuds ET, respectivement détaillés dans les chapitres trois et quatre.

Ces algorithmes décrivent la vie d'un nœud, de sa création à sa destruction. L'interprétation d'une primitive par le nœud assure la transition entre deux étapes consécutives. Entre chaque étape, le nœud peut être amené à changer d'état, à émettre une (voire plusieurs) primitive, à effectuer des traitements locaux. Six primitives et cinq états suffisent à la gestion des nœuds.

Les primitives sont réparties en quatre classes distinctes. Les deux premières classes de primitives, en l'occurrence les primitives d'activation et les demande de solutions, sont des primitives descendantes, car elles sont appliquées par un nœud père à un nœud fils. A l'inverse, les primitives de la troisième classe, la classe de résultat, sont des primitives ascendantes qui transmettent à un nœud père une solution (voire un échec) évaluée par un nœud fils. La quatrième classe se réduit à la primitive particulière qu'est la primitive de destruction du nœud : en effet, la seule initiative réelle d'un nœud étant son autodestruction, le nœud est à la fois l'émetteur et le destinataire de cette primitive.

Les cinq états qu'un nœud peut revêtir sont exclusifs les uns des autres. L'état initial d'un nœud est l'état créé, l'état final d'un nœud est l'état résolu. Les trois autres états déterminent la classe de la primitive attendue par le nœud. Les nœuds père et fils du nœud travaillant de façon asynchrone, l'attente du nœud est relative soit à une primitive

demande de solutions, soit à une primitive résultats, soit à une primitive de chacune de ces classes.

Le chapitre quatre montre que la gestion d'un nœud ET est une opération plus complexe que celle d'un nœud OU. En effet, à tout moment, un nœud OU dialogue avec au plus un de ses fils (les autres sont activés en mode restreint), alors qu'un nœud ET peut dialoguer simultanément avec plusieurs de ses fils afin de maintenir amorcé le pipeline ET.

Afin de rendre possible l'exécution de tout programme Prolog existant, selon le modèle d'évaluation, il s'avère nécessaire d'introduire des prédicats dits à effets de bord. Le chapitre cinq est consacré à la prise en compte de trois d'entre eux qui sont le prédicat CUT et les prédicats de lecture et d'écriture. Le prédicat CUT permet de contrôler l'expansion de l'arbre de recherche en coupant certains choix en attente. Notre modèle, étant un modèle d'évaluation parallèle de Prolog, implique le développement anticipé de branches OU qui, lorsque le CUT est rencontré, deviennent inutiles. Le problème consiste à détruire des branches, voire des arborescences dynamiques : en effet, les nœuds qui constituent ces entités sont créés et détruits sur échange de primitives. Il faut notamment éviter la destruction de primitives que la destruction prématurée d'un nœud priverait de destinataires. Aussi, un nœud ne peut-il se détruire qu'aprés s'être assuré de ne plus recevoir de primitives en provenance de son père et de ses fils. Le principe adopté repose sur la mise en place de primitives supplémentaires dont d'une part l'ordre de destruction émis par un nœud vers l'un de ses fils et d'autre part l'accusé de réception correspondant. Le dialogue entre le nœud à détruire et son père prend donc fin lorsque le nœud a reçu l'ordre de se détruire et a émis un accusé de réception. Le nœud propage ensuite l'ordre de destruction à ses propres fils et se met en attente des accusés de réception correspondants. Lorsqu'ils lui sont tous parvenus, le nœud peut se détruire.

Lors de l'exécution d'un programme Prolog, les prédicats de lecture et d'écritures sont des opérations qui s'effectuent selon un ordre donné, que le développement parallèle de branches OU dans le modèle proposé peut perturber. Or, le modèle implique qu'à tout moment une seule branche est active (sur cette branche n'est posé aucun verrou de solutions). Sur cette seule branche peuvent être exécutés les prédicats de lecture et d'écriture. Aussi, à tout instant, un nœud doit savoir s'il se situe ou non sur la branche active. A cette fin, on associe à chaque nœud

un booléen dont la valeur est modifiée à chaque interprétation de primitive par le nœud. Le chapitre cinq détaille le calcul de cette valeur booléenne.

Le chapitre six décrit une première évaluation du modèle. Cette évaluation a été menée dans deux directions différentes. Lors de la première phase, une simulation abstraite a été réalisée en langage C sur station SUN. Cette simulation a un double but : d'une part, elle permet de vérifier le parcours de l'arbre ET/OU obtenu (donc de s'assurer que la sémantique opérationnelle de Prolog a été respectée), d'autre part elle a donné les premières estimations quant au taux moyen de parallélisme (en nombre de nœuds simultanément actifs) développé. La seconde action consiste en une simulation réelle, c'est-à-dire en l'observation de comportement de programmes réels, exécutés selon le schéma d'évaluation proposé. L'observation a été réalisée d'aprés les traces de l'exécution des programmes, obtenues grâce à l'introduction systématique de prédicats espions dans chaque règle de programme. Ces deux simulations ont principalement montré que respecter la sémantique opérationnelle de Prolog, et plus particulièrement construire des ensembles de solutions ordonnées identiques à ceux produits par Prolog, est une contrainte forte. Cette contrainte est d'autant plus forte lorsque le nombre de solutions est important.

PROGRAMMATION LOGIQUE

PROGRAMMATION LOGIQUE.

I. INTRODUCTION.

Dans un premier temps, nous décrirons les deux mécanismes sur lesquels repose l'exécution séquentielle d'un programme Prolog. Ces deux mécanismes sont respectivement l'unification et la résolution.

Dans un second temps, nous évoquerons les différentes sources de parallélismes supportés par la programmation logique. Nous évoquerons ensuite les deux approches qui ont été développées à partir des sources de parallélisme citées, à savoir:

- l'approche des langages gardés, qui sont des langages parallèles de programmation logique, intégrant un contrôle explicite.
- l'approche des modèles non déterministes, qui sont relativement compatibles avec le langage Prolog et qui n'intègrent pas de primitives de communication supplémentaires.

Nous terminerons en présentant différents modèles de calcul et architectures parallèles élaborées à partir des deux approches précédentes.

II. PROLOG SEQUENTIEL.

II.1. Définitions.

Prolog est un langage logique, doté de mécanismes de contrôle, tel le CUT. Il est constitué d'un ensemble de clauses de Horn, de forme générale :

P:-L1....Ln (n>=0)

où P est la tête de clause, et la suite des Li le corps de la clause. P et chacun des Li (0<=i<=n) sont des atomes. Un atome s'écrit R(t1,...,tm) où R est un symbole de prédicat, et les différents tj (1<=j<=m) sont des termes. Un terme est soit :

- une constante (ex. : a,b,c...)
- une variable (ex. : x,y,...)
- un terme structuré, qui est soit un p-uple <t1,...,tp>, soit une fonction f(t1,...,tp) où f est un symbole de fonction. Dans chacune de ces représentations, chaque tj (1<=j<=p) est lui-même un terme.

Si le corps de la clause est vide, la clause est un fait. Un prédicat (ou paquet) est constitué de l'ensemble des clauses de Horn, qui ont même atome de tête. L'ordre des clauses dans le paquet a son importance.

Exécuter un programme Prolog, revient à déterminer l'ensemble des solutions satisfaisant un but initial. Un interpréteur Prolog repose sur deux mécanismes essentiels qui sont respectivement l'unification et la résolution, mécanismes que nous allons maintenant décrire.

II.2. L'unification.

II.2.a. Définition.

L'unification détermine l'instance commune de deux termes t1 et t2 si elle existe. Dans le premier algorithme d'unification proposé par Robinson [Robi 65], l'instance commune de deux termes t1 et t2 est l'unificateur le plus général de ces deux termes où :

- un unificateur de deux termes t1 et t2 est une substitution ß, c'est-à-dire un ensemble fini de couples (variable, terme) où une même variable n'apparaît pas en partie gauche de deux couples ou plus, telle que ß(t1)=ß(t2). Si un tel unificateur existe, les termes t1 et t2 sont unifiables.

Considérons l'exemple suivant où ß est donné par {(X, f(a,b)), (Y, g(Z))}. Trois variables, en l'occurrence X, Y, et Z, figurent dans cette substitution. Les variables X et Y qui apparaissent dans la partie gauche des couples sont dites variables liées (respectivement aux termes f(a,b) et g(Z)), la variable Z qui n'apparaît en partie gauche d'aucun couple, est dite variable libre.

- l'unificateur le plus général est l'unificateur g, tel que pour tout unificateur f, il existe une substitution ß telle que ß=fog (l'opérateur o est l'opérateur de composition).

II.2.b. Algorithme d'unification de Robinson.

L'algorithme proposé par Robinson est le suivant :

```
fonction  unifier(t1, t2):(bool,ß );
données     t1, t2 : termes;
résultats   bool : booléen;  (* bool est vrai ssi t1 et t2 sont unifiables*)
              ß : substitution;  (* Si bool est vrai alors ß est le plus grand
            unificateur de t1 et t2*)

début
si (t1 est une variable x et t2 le terme t)
ou (t2 est une variable x et t1 le terme t)
alors si x=t
      alors début
            bool=vrai;
            s=ø
```

```
            fin
      sinon si occur(x, t)
            alors bool=faux
            sinon  début
                   bool=vrai;
                   ß = {(x,t)}
                   fin
            fsi
      fsi
sinon (* t1 et t2 sont deux termes structurés respectivement égaux à
f(x1,...,xn) et g(y1,....,yp) *)
      si f<>g
      alors bool=faux
      sinon si n<>p
            alors bool=faux
            sinon  début
                   i=1; bool=vrai;ß=ø;
                   tq (i<=n) et bool
                     faire (bool,ß1) = unifier(ß(xi),ß(yi));
                         si bool alors ß=ß1oß fsi
                     fait
                   fin
            fsi
      fsi
fsi
fin
```

Dans cet algorithme, la fonction occur(x,t) vérifie l'appartenance de la variable x au terme t. La fonction occur(x,t) retourne vrai si la variable x apparait dans le terme t ; dans le cas inverse, la valeur retournée est 'faux'. Cette fonction effectue donc le test d'occurrence. Un algorithme, qui inclut ce test, impose donc qu'une variable ne peut être liée à un terme la contenant. C'est le cas de l'algorithme de Robinson.

Il est à remarquer que la plupart des systèmes Prolog n'incluent pas le test d'occurrence, qui se révèle être une opération coûteuse: en effet, il faut parcourir complètement le terme t afin de déterminer si la variable x figure ou non dans ce terme. Cependant, la suppression de ce test peut entraîner la construction de termes infinis. Deux types de solutions [Boiz 89] pallient ce problème :

- l'une laisse à l'utilisateur l'initiative d'utiliser ou non le test d'occurrence [Mart 82, 83]
- l'autre, mis en oeuvre dans Prolog II [Colm 82] [VanC 82] [Kano 82], prend en compte les termes infinis, au lieu de les rejeter. La notion d'unification cède la place à la résolution d'un système d'équations sur

les arbres rationnels, c'est à dire les arbres possédant un ensemble fini de sous-arbres.

II.2.c. Algorithme d'unification de Fages.

Cet algorithme manipule des termes rationnels, c'est-à-dire des termes dont le nombre de sous-termes distincts est fini. Ces derniers sont représentés par des Graphes Ordonnés, notés GO, pouvant contenir des circuits.

Les termes sont construits à partir des ensembles suivants :

- F, l'ensemble fini des fonctions, dont l'arité est connue.
- V, l'ensemble des variables.

La figure 1 présente l'arbre représentant le terme f(x,g(y),z)

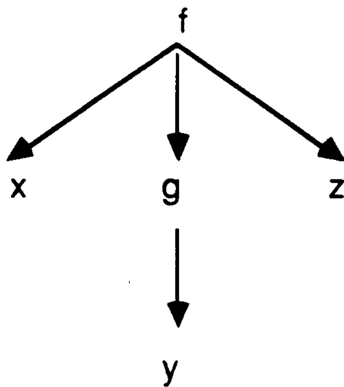


Figure 1 : Représentation d'un terme fini.

La représentation du terme infini f(a,f(a,f.............) est donc :

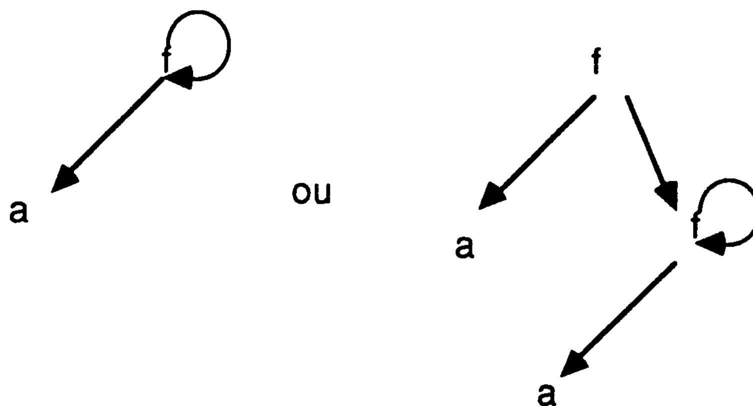


Figure 2 : Représentation d'un terme infini.

A tout sommet x d'un tel graphe G, on associe :

- etiq(x) : l'étiquette de x symbole de variable ou de fonction.
- arité(x) : l'arité de x. L'arité d'une variable est nulle.
- succ(x,i) : la fonction qui retourne le ième successeur du sommet x.
- var(x) : la fonction booléenne qui retourne vrai si x est une variable, faux sinon.
- g[x<-y] : la fonction qui substitue le sommet y au sommet x dans le graphe.
- x<-y : la substitution de la variable x par le terme représenté par le terme y.

L'algorithme d'unification devient donc :

```
fonction  unificateur(m,n)
début
si m=n
alors retourner ø
sinon si var(m)
     alors  début
            g[n-<m] ;
            m-<n;
            fin
      sinon si etiq(m)=etiq(n)
             alors  début
                    g<-g[m<-n];
                    ß=ø;
                    pour i variant de 1 à m
                       faire ß <- unificateur(succ(m,i), succ(n,i)) o ß  fait
                    retourner ß
                    fin
             sinon erreur
            fsi
     fsi
fsi
fin
```

La terminaison de cet algorithme est assurée car le nombre de sommets non remplacés diminue avant chaque appel récursif. En effet, le sommet m, qui n'est pas une variable, est remplacé par le sommet n (lui-même non variable) avant le calcul récursif du plus grand unificateur des successeurs de m et n.

De plus, lors du remplacement d'un sommet par un autre, le graphe initial n'est pas détruit : des chaînes d'indirection sont créées. Les substitutions peuvent alors être défaites. A chaque sommet m, est associé ind(m), qui vaut n, si n est substitué à m, et ø dans le cas contraire. Le calcul du représentant de m consiste donc à parcourir cette chaîne afin de trouver le sommet dont ind(n) est nul. La fonction repr(m) suivante réalise ce calcul.

```
fonction repr(m)
début
si ind(m)=0
alors retourner m
sinon début
      ind(m)<-repr(ind(m);
      retourner ind(m)
      fin
fsi
fin
```

L'algorithme d'unification devient le suivant :

```
procédure unificateur(m,n)
début
si m<>n
alors si var(m)
      alors ind(m):=n
      sinon si var(n)
            alors ind(n):=m
            sinon si etiq(m)=etiq(n)
                  alors début
                        ind(m):=n;
                        pour i variant de 1 à arité(m)
                        faire unifier(repr(succ(m,i)),repr(succ(n,i))) fait
                        fin
                  sinon erreur
                  fsi
            fsi
      fsi
fsi
fin
```

II.3. Le mécanisme de résolution.

La stratégie de résolution de Prolog est une SLD-résolution [Apt 82] [Llyo 82], c'est-à-dire, une méthode qui utilise une stratégie de sélection du littéral.

Dans un interpréteur Prolog, à chaque étape sont appliquées deux règles de sélection. Ainsi, seront systématiquement considérés d'une part le premier des buts de la liste courante des buts à effacer (le but le plus à gauche) , d'autre part la première des clauses qui constituent la définition d'un prédicat. Les clauses seront considérées selon leur ordre d'apparition dans le paquet.

Détaillons une étape de résolution associée à la liste de buts $B_1..B_n$. Cette étape se déroule en trois phases :

-1) En premier lieu, le butl B_1 est sélectionné.
-2) Puis, la première clause, C<- L1..Lp, du paquet qui définit B1 est élue. L'unification entre d'une part B_1 et d'autre part la tête de la régle ainsi déterminée a lieu. Si l'unification réussit, le but B_1 est enlevé de la liste des buts et remplacé par la queue de la règle C-> L_1,.. , L_p , non sans avoir effectué les substitutions nécessaires à l'unification sur l'ensemble de la liste des buts. Remarquons que si la queue de la règle concernée est vide, le but B1 est seulement effacé, les substitutions étant réalisées comme précédemment.
-3) La première phase est de nouveau effectuée sauf si :
a) la liste courante des buts est vide : un succès a alors été obtenu.
b) un échec est survenu lors de la démonstration d'un des buts.

Ces deux situations n'arrêtent pas l'exécution du programme. Ainsi, une étape de résolution a entraîné le choix systématique d'une unique clause parmi celles qui constituent un paquet. Les autres clauses sont restées en attente, ouvrant des possibilités non encore exploitées. Il y a alors eu création d'un point de choix en attente, associé au but en cours de résolution. En cas d'échec, lors de la résolution d'un but, le dernier point de choix créé est examiné entraînant la sélection de la première des clauses restantes du paquet concerné. Ce mécanisme constitue le retour arrière Prolog, encore appelé backtracking. L'examen complet de l'ensemble des choix en attente met un terme à l'exécution du programme.

Cette stratégie de résolution permet de définir le parcours d'un arbre SLD : celui-ci s'effectue en profondeur d'abord, et de gauche à droite.

De plus, elle permet de définir le parcours d'un arbre ET/OU dans lequel figurent deux types de nœuds :

- les nœuds OU, qui résolvent un littéral unique ;
- les nœuds ET, qui résolvent l'ensemble des littéraux qui constituent un corps de clause.

Nous allons expliquer la manière dont est parcourue l'arbre ET/OU en présentant un exemple.Ainsi, la figure 4 présente l'arbre ET/OU associé au programme Prolog de la figure 3. Le but à résoudre est gpar(luc, T).

```
gpar(X, Y) :- père(X, Z) par(Z, X).
par(X, Y) :- père(X, Y).
par(X, Y) :- mère(X, Y).
père(jean, luc).
père(jean, paul).
père(luc, marie).
mère(lise, yves).
mère(marie, guy).
:- gpar(luc, T).
```

Figure 3 : Un programme Prolog.

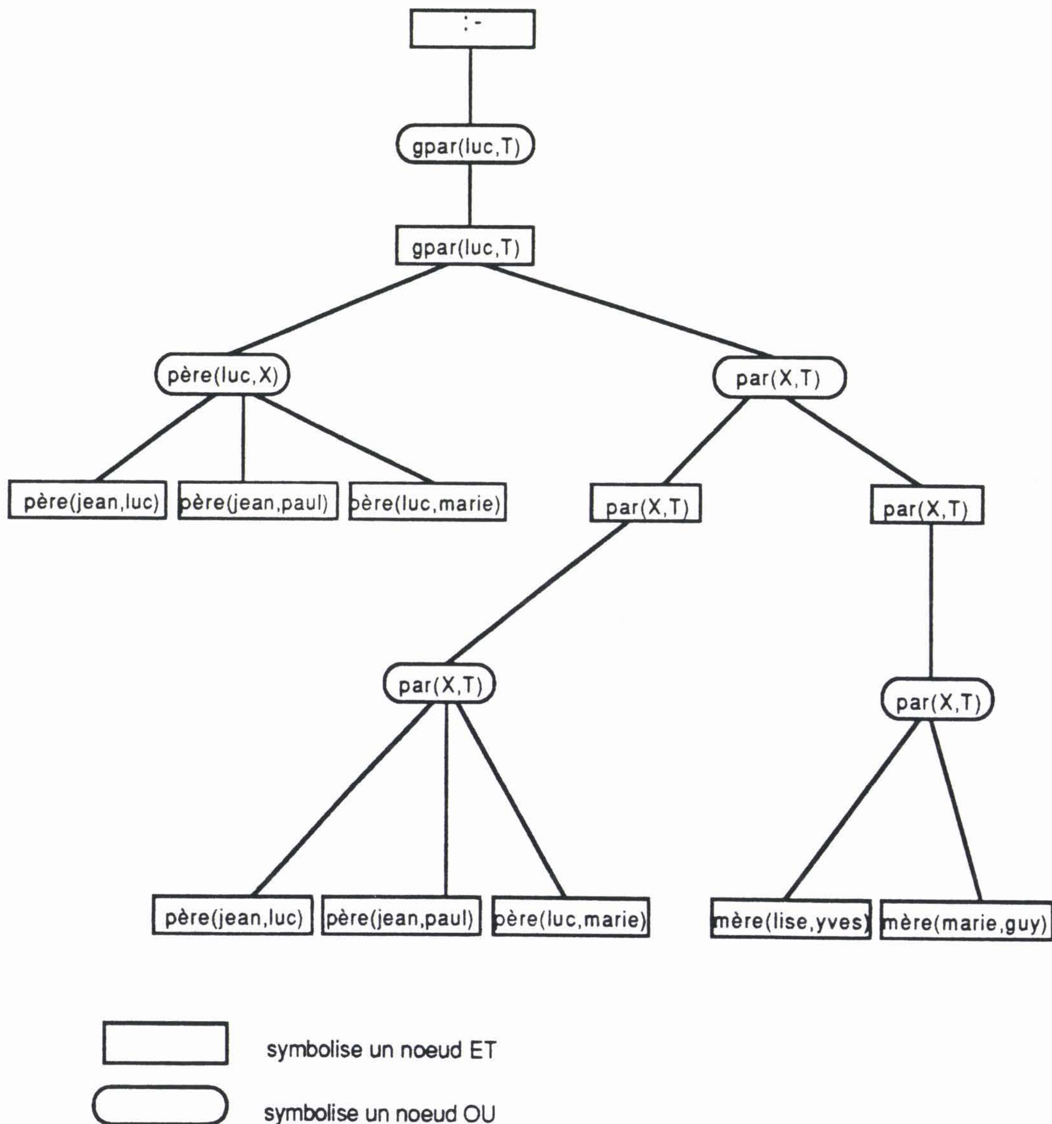


Figure 4 : Arbre ET/OU.

La figure 5 montre le parcours de l'arbre ET/OU précédemment présenté. Les branches ET, issues du nœud gpar(luc,T) sont parcourues de gauche à droite conformément à la règle de sélection du but : ainsi, sont successivement effacés les buts père(luc,X) et par(X,T). Les branches OU issues respectivement des nœuds père(luc,T), père(X,T) et mère(X,T) sont successivement parcourues car aucun succés n'est évalué (s'il existe) en dehors de celui produit par la dernière alternative. Dans cet exemple, aucun retour en arrière n'est effectué car les règles père et mère qui entraînent un succés, apparaissent en dernière position de leurs paquets respectifs.

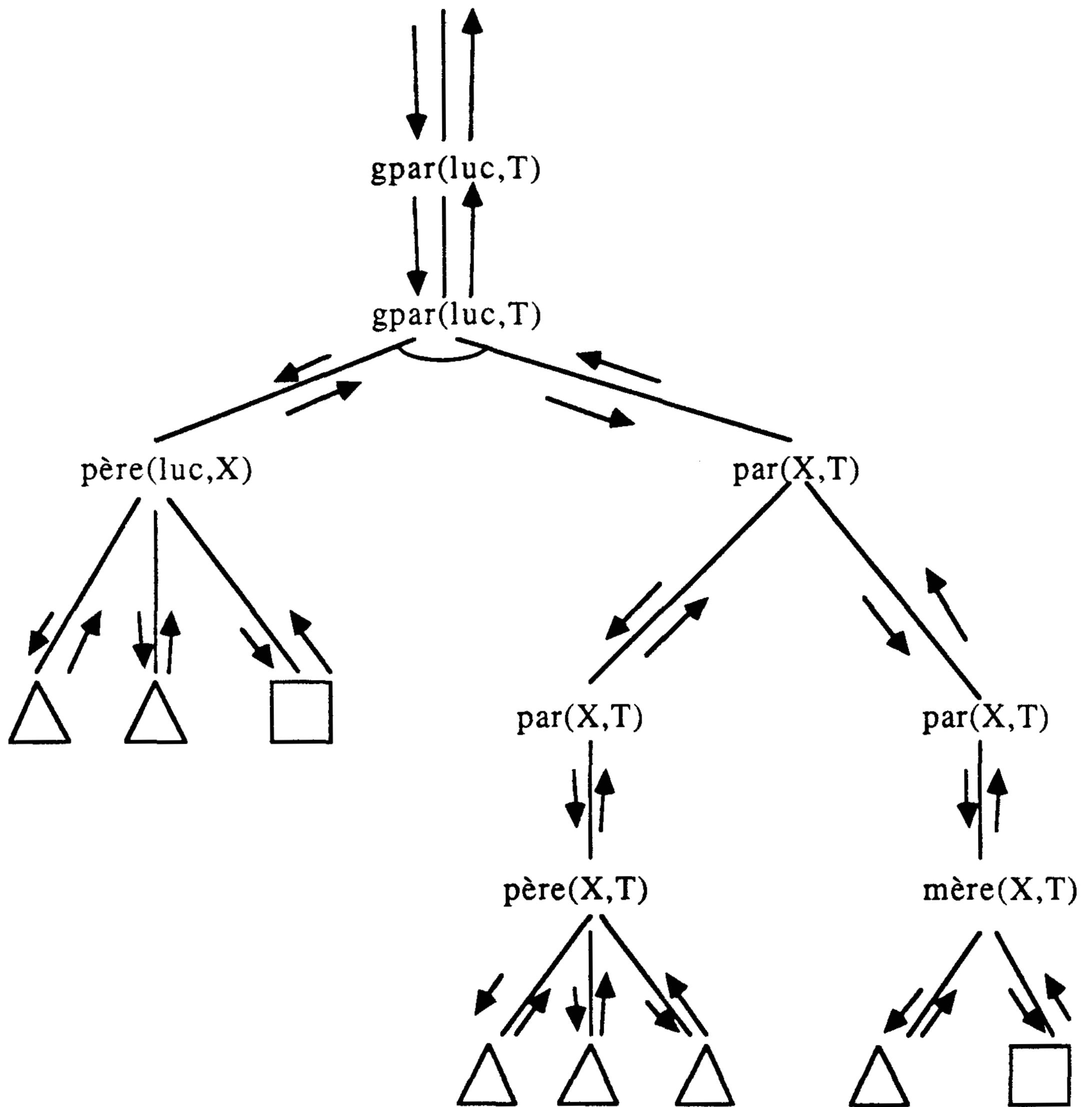


Figure 5 : Parcours de l'arbre ET/OU.

□ est la résolvante vide obtenue en cas de succès.
△ symbolise l'échec évalué lors de la résolution.

Si, pour le même programme Prolog, on considère le but gpar(jean,T), le parcours de l'arbre ET/OU associé se décompose en les étapes décrites dans les figures 5.1 à 5.4. La première alternative qui définit le prédicat père donne un succés. Il reste deux autres alternatives à évaluer ultérieurement (elles sont symbolisées par les pointillés):

<u>Etape n°1</u>

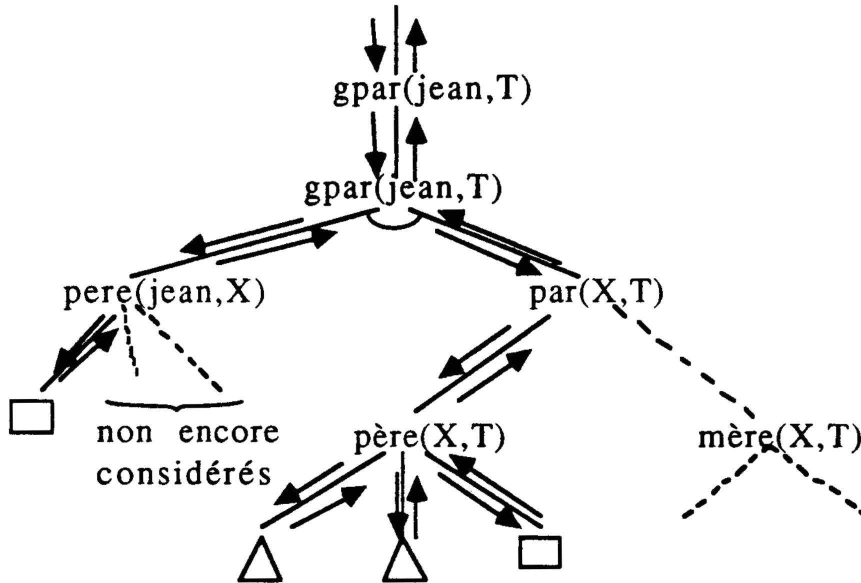


Figure 5.1

<u>Etape n° 2</u>: La solution (T=marie) est évaluée. Comme toutes les alternatives définissant le prédicat par n'ont pas été traitées, l'évaluation se poursuit par l'examen du prédicat mère.

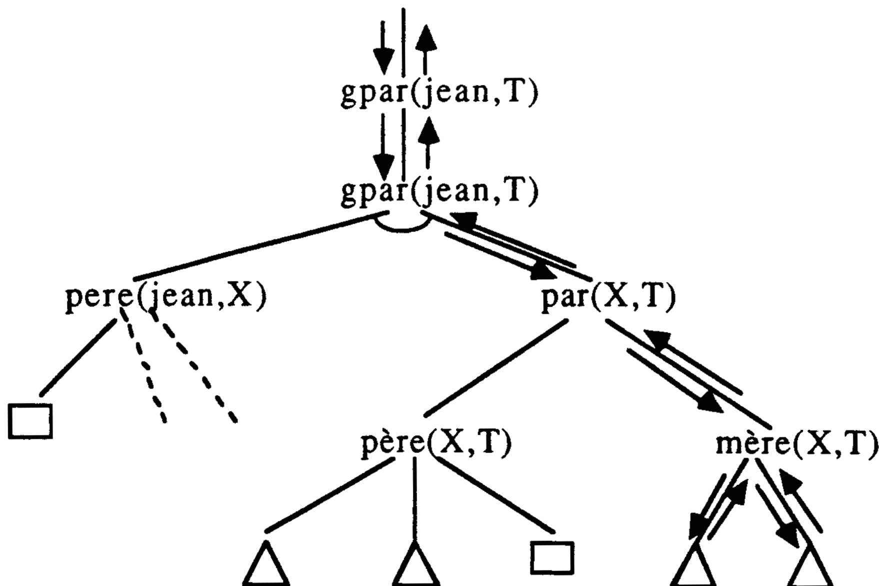


Figure 5.2

<u>Etape n° 3</u>: Un retour arrière est ensuite effectué afin de prendre en considération le premier des faits, en l'occurrence père(jean,paul), qu'il reste à envisager.

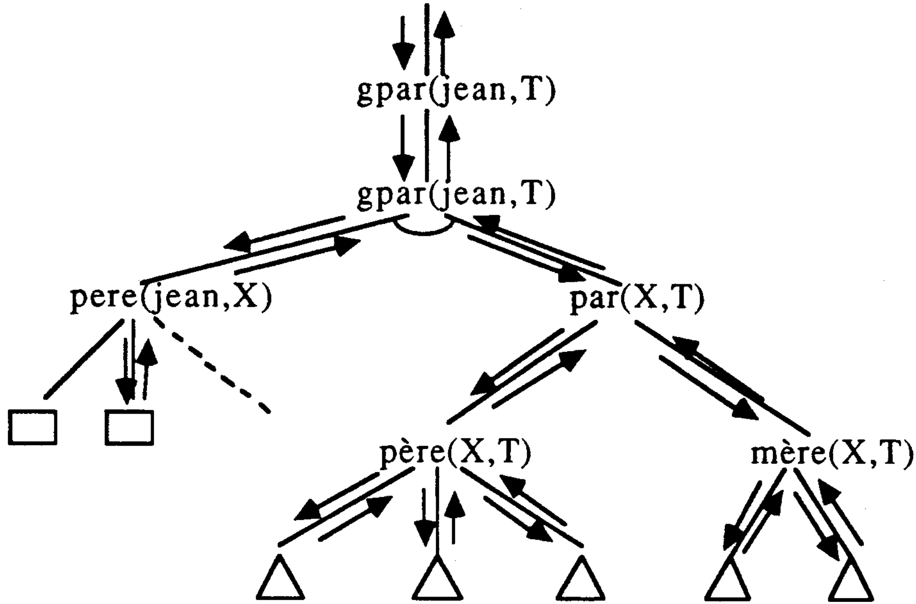


Figure 5.3

<u>Etape n° 4</u>: La troisième alternative père(luc,marie) est considérée : elle conduit à l'échec. L'exécution du programme est terminée.

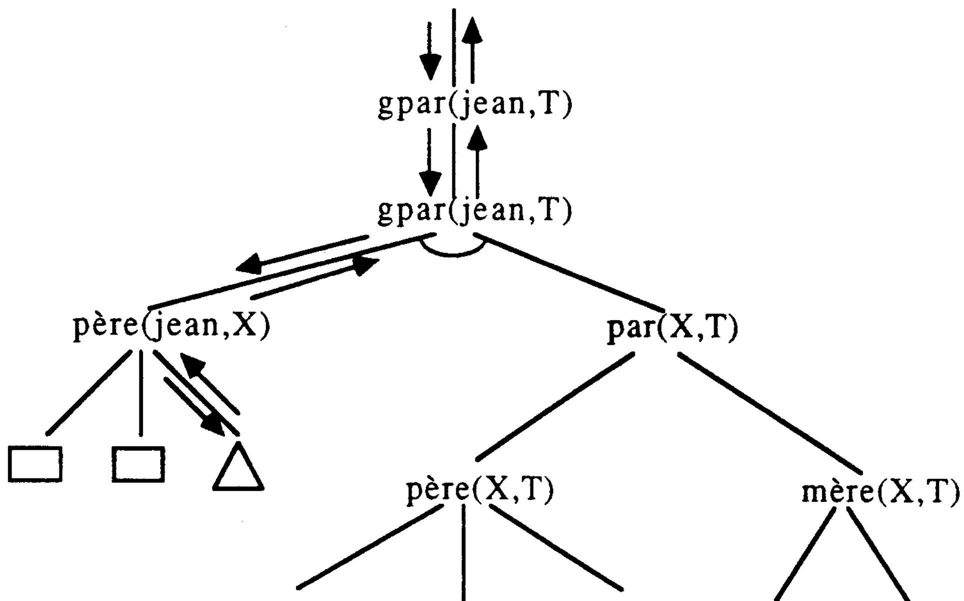


Figure 5.4

II.4. Evaluation de Prolog avec trois piles.

L'évaluation de Prolog nécessite les trois piles suivantes :

- la pile locale
- la pile de restauration (trail)
- la pile globale ou la pile de recopie (heap).

La pile locale contient deux types d'objets, qui sont d'une part les environnements et d'autre part les points de choix.

* Un environnement reçoit les valeurs des variables qui figurent dans la clause. Chaque variable est numérotée. On accède donc à la variable par simple déplacement dans le vecteur. Lors de l'unification, la liaison d'une variable à un terme est réalisée en mémorisant la référence à ce terme.

* Un point de choix est créé lors de l'appel d'un but si plusieurs alternatives sont susceptibles de satisfaire l'unification avec le but. Il contient toutes les informations nécessaires à la restauration du contexte antérieur au calcul, dans l'éventualité d'un retour arrière. Ces informations se divisent en deux groupes : la reprise et la continuation. La reprise gère le retour arrière. Outre le bloc de choix sur la pile locale, et l'accès aux alternatives non encore envisagées, elle désigne le sommet de la pile de restauration. La continuation gère l'avancée en permettant de retrouver la prochaine suite de buts à prouver (après examen complet de la clause) ainsi que le bloc d'activation qui détient l'environnement à considérer. On retrouve ce type d'informations dans les blocs déterministes, c'est-à-dire ceux créés lorsque la clause candidate à l'unification est la dernière clause envisageable du paquet.

La pile de restauration manipule les références aux variables liées lors de l'unification et remises à l'état libre lors du retour arrière. Elle est donc mise à jour lors des retours arrière.

L'unification, en Prolog, implique la construction de nouveaux termes, dits instances, à partir d'anciens, dits modèles. Deux mécanismes distincts, en l'occurrence le partage des structures et la recopie des structures, peuvent être mis en place afin de générer ces nouveaux termes.

* Dans le partage des structures [Boye 72], l'instance d'un terme structuré est un couple (modèle, environnement) où le modèle est celui du terme et l'environnement décrit l'ensemble des valeurs des variables qui y figurent. En fait, deux instances d'un même modèle partagent ce modèle, mais diffèrent par l'environnement. La liaison d'une variable à un terme structuré se fait par affectation d'un doublet (modèle, environnement).

L'utilisation d'un tel algorithme suscite l'apparition d'une nouvelle pile comme le suggère D. Warren [Warr 77]. Cette pile est la pile globale qui mémorise les variables dont la durée de vie pourrait être supérieure à celle de la clause où elles figurent.

* Dans la recopie de structure [Brun76, 80], la création d'un nouveau terme est effectuée par recopie du modèle initial du terme. Ce mode de représentation des termes repose sur les deux principes suivants :

- la création d'une nouvelle instance de terme implique la recopie du modèle initial du terme, suivie de la liaison de la variable avec cette copie.
- L'accès à une instance de terme déjà existante entraîne la seule liaison entre la variable et la copie du terme.

Cet algorithme nécessite une pile supplémentaire, à savoir la pile de recopie. Cette zone supporte la recopie des termes structurés : en particulier, y seront alloués autant d'emplacements mémoire que nécessaires à la recopie du terme.

En 1983, David H. Warren [Warr 83] proposait une machine virtuelle, la WAM, qui devint par la suite la base de nombreux travaux , notamment dans le domaine de la compilation Prolog.

- Les termes, manipulés par la WAM, sont représentés par recopie de structures. Outre la zone de code qui contient les instructions et les données nécessaires à la représentation du programme source la WAM utilise une zone de travail divisée en trois piles, à savoir la pile locale, la pile de recopie et la pile de restauration. Elles sont organisées ainsi que le montre la figure 6.

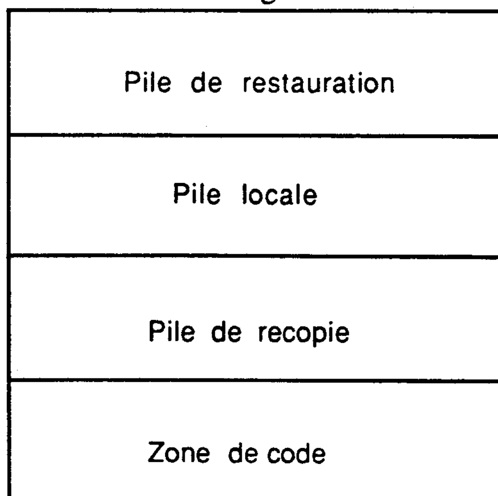


Figure 6 : Structure de l'espace de travail WAM.

- La WAM intègre en outre un certain nombre d'optimisations

dont l'une des plus intéressantes est le double niveau d'indexation des clauses. Cette opération élimine de façon immédiate des alternatives dont on sait qu'elles ne satisferont pas l'unification avec le but appelant. Les points de choix correspondants ne seront donc pas créés. Ce procédé repose sur l'analyse du premier argument du but qui rappelons le, est soit une variable, une constante, une liste ou un terme fonctionnel. Examinons ces différentes possibilités :

* S'il est une variable, toutes les clauses du paquet conviennent.

* S'il est une constante, seules les clauses dont le premier argument du littéral de tête est une constante ou une variable sont à examiner.

* S'il est une liste (respectivement un terme fonctionnel), les clauses présentant une variable ou une liste (respectivement un terme fonctionnel) en tant que premier argument de littéral de tête sont les seules à être traitées.

Suite à cette analyse, le paquet de clauses est divisé en sous-paquets en vue d'une sélection future. Une seconde indexation réorganise ces sous-paquets. Cette seconde indexation varie selon la nature du premier argument de la tête de clauses :

* Dans le cas d'une variable, toutes les clauses conviennent.

* On indexe les constantes par elles-mêmes.

* On distingue les termes fonctionnels par leurs symboles fonctionnels.

* On distingue les listes vides des listes non vides.

Ce double niveau d'indexation d'une part permet un gain de temps en éliminant des unifications inutiles, d'autre part minimise la taille de la pile locale en diminuant le nombre d'alternatives à examiner.

III. PROGRAMMATION LOGIQUE ET PARALLELISME.

III.1. Sources de parallélisme en programmation

La programmation logique offre plusieurs sources de parallélisme [Cone 81], dont les trois principales sont :

- *le parallélisme OU* : plusieurs, voire toutes les clauses qui constituent un prédicat sont exécutées parallèlement.
- *le parallélisme ET* : plusieurs, voire tous les buts de la liste courante de buts sont évalués simultanément.
- *le parallélisme de flot* (noté stream parallelism) : Les dépendances de données entre sous buts d'une clause sont énoncées explicitement. Les buts sont lancés en parallèle mais se synchronisent en fonction de ces dépendances de données.

D'autres sources potentielles existent, parmi lesquelles :

- le parallélisme d'unification : on tente l'instanciation parallèle des arguments d'une tête de clause.
- le parallélisme dit d'accès à la de base de faits : ce parallélisme est relatif à l'ensemble des faits du programme. Il peut être considéré comme un sous-ensemble du parallélisme OU.

Dans la suite de cette étude nous nous concentrerons sur les trois premières sources de parallélisme. Dans un premier temps, nous exposerons les principaux avantages et inconvénients de chaque type de parallélisme. Puis, nous présenterons quelques modèles définis à partir de ces types de parallélisme.

III.2. Le parallélisme en programmation logique: avantages et inconvénients.

III.2.a. Le parallélisme OU

Le parallélisme OU présente les deux avantages suivants : il offre un gros grain de parallélisme puisqu'il se situe au niveau des clauses (alors que celui procuré par le parallélisme ET se situe au niveau des sous-buts) et nécessite peu de synchronisation. Toutefois, il n'est exploitable que dans des programmes non déterministes. Enfin, il entraîne la gestion d'un volume important d'informations. En effet, une même variable peut être liée plusieurs fois lors de l'activation simultanée de différentes clauses alternatives. Dans le cadre d'une exécution séquentielle, une cellule contenant une variable X, serait référencée plusieurs fois. Deux écritures successives dans la même cellule sont séparées par un retour en arrière, pendant lequel la cellule est de nouveau initialisée. Par contre, une exécution parallèle entraîne

l'utilisation de plusieurs cellules mémoire, situées dans des environnements distincts.

III.2.b. Le parallélisme ET

Le parallélisme ET présente l'avantage majeur d'être présent dans tous les types de programmes, qu'ils soient déterministes ou non.

A l'inverse, il soulève un problème important, connu comme étant le problème des variables partagées par différents buts de la liste courante de buts.

Ainsi, si nous considérons la clause suivante : p(X,Y) :- q(X,Z) , r(Z,Y)., nous constatons que les littéraux q et r partagent la variable Z et par conséquent doivent accéder à la même instance de cette variable. Or une évaluation simultanée et indépendante de ces deux littéraux, libre de tout contrôle, pourrait instancier la variable Z à deux valeurs distinctes. Le problème peut d'ailleurs être étendu si lors de l'exécution, les variables X et Y sont liées à un même terme.

Différents types de solutions ont été envisagées :

- une évaluation indépendante et simultanée des buts associée à un mécanisme de jointure : ce mécanisme est peu efficace car il ralentit considérablement l'exécution du programme.
- une synchronisation au niveau des variables, énoncée par le programmeur : cette approche est celle des langages gardés, et sera détaillée dans un paragraphe ultérieur.
- une synchronisation au niveau des sous-buts : les buts qui partagent une variable sont exécutés séquentiellement. Cette technique rend nécessaire la détection de l'indépendance entre buts. Ainsi, Conery [Cone 83]. propose un modèle où les dépendances entre buts sont calculées dynamiquement et représentées par des graphes. Ces graphes indiquent les relations de production/consommation des variables entre les différents sous-buts. Degroot [Degr 84] propose une analyse mi-statique, mi-dynamique des clauses. La compilation génère des tests simples, qui lors de l'exécution permettent de déterminer si les sous-buts sont, ou non, indépendants. Cette méthode a donné naissance au parallélisme ET restreint [Degr 84, 87] [Herm 86].

III.2.c. Le parallélisme de flot.

Les modèles basés sur le parallélisme de flot obéissent à un schéma de type producteur consommateur : un littéral produit une instance d'une variable, instance qui sera consommée par d'autres littéraux à sa-

tisfaire et dont l'exécution était jusqu'alors suspendue. La synchronisation des différents processus créés lors de l'évaluation des littéraux apparaît donc comme le seul problème majeur à résoudre.

A partir de ces trois sources de parallélisme, ont été définies deux grandes classes de modèles [Chas 89a][Chas 89b] :

- les langages gardés qui reposent sur le parallélisme de flot
- les modèles non-déterministes qui visent à calculer l'ensemble des solutions répondant au problème initial. Ces modèles intègrent soit le parallélisme OU, soit le parallélisme ET, soit les deux types de parallélisme.

III.3 Modèles.

III.3.a. Modèles exploitant le parallélisme de flots.

A. Les langages gardés.

Les langages gardés dérivent des travaux de Clark sur le langage relationnel [Clar 81]. Ces langages ont une syntaxe, une sémantique déclarative et une sémantique opérationnelle commune.

Syntaxe des langages gardés :

$$H:- G_1.....G_n \mid B_1...B_m \quad (n,m>=0)$$

où :

- H est la tête de clause
- $G_1...G_n$ (n>=0) est la garde de la clause , c'est-à-dire l'ensemble des conditions à satisfaire afin d'exécuter le corps de la clause.
- | est l'opérateur d'engagement , encore appelé 'commit operator'.
- $B_1...B_m$ (m>=0) est le corps de la clause.

Sémantique déclarative des langages gardés:

Pour toute valeur de variable de la clause,

H est vrai si et seulement si pour tout i (0<=i<=n) G_i est vrai et et tout j (0<=j<=m) B_j est vrai.

Sémantique opérationnelle des langages gardés :

Le calcul est organisé hiérarchiquement en un arbre de processus ET/OU. Lors de la résolution d'un but, toutes les clauses alternatives sont examinées. Dès que réussissent successivement une unification et l'évaluation de la garde correspondante, l'examen des autres clauses est définitivement suspendu. Seule, se poursuit l'exécution du corps de la clause ainsi élue. La garde apparaît donc comme la synchronisation nécessaire entre les processus OU. La synchronisation entre les processus ET est quant à elle réalisée selon un mécanisme du type produc-

teur/consommateur. Le programmeur énonce explicitement des relations du type producteur/consommateur, par l'intermédiaire de primitives, voire d'opérateurs spécifiques à chacun des langages gardés. Ces primitives précisent le mode d'accès aux variables. Ces modes d'accès sont respectivement l'accès en lecture (la variable ne peut être instanciée), l'accès en écriture (la variable peut être instanciée). Dans le premier cas, la variable est une variable dite en entrée, dans le second cas, la variable est dite en sortie. Ces primitives assurent ainsi une communication asynchrone entre les buts parallèles. Elles diffèrent d'un langage à l'autre. Nous nous attarderons particulièrement sur les langages suivants: Parlog [Clar 85, 84], Concurrent Prolog [Shap 83,87], et Guarded Horn Clause [Ueda 85,86].

En Parlog, une déclaration de mode est associée à chaque prédicat. Une variable en sortie est notée "?", une variable en entrée est notée "^". Le programme merge (tri de deux listes) écrit en Parlog est le suivant :

```
mode merge(?, ?, ^).
merge([X|XS], YS, [X|ZS]) :- true| merge(XS, YS, ZS).
merge(XS,[Y|YS],[Y|ZS]) :- true| merge(XS, YS, ZS).
merge([], YS, YS):-true|true.
merge(XS,[],XS):-true|true.
```

Notons que Parlog possède outre les opérateurs précédents :

- l'opérateur "&" qui force l'exécution séquentielle de deux littéraux successifs.
- l'opérateur "." qui permet l'exécution parallèle de deux clauses séparées par cet opérateur.
- l'opérateur ";" qui marque la fin d'un paquet de clauses exécutables en parallèle. Les clauses qui se situent après ce paquet ne sont évaluées que si l'évaluation des gardes des clauses du paquet a échoué.

En Concurrent Prolog, seules sont annotées les variables accessibles en lecture (c'est-à-dire que l'on ne peut instancier) par le symbole "?". Ces variables appelées variables read-only peuvent figurer à un endroit quelconque : tête de clause, corps de clause, but. Le programme merge devient le suivant :

```
merge([X|XS],YS,[X|ZS]) :- true | merge(XS?, YS, ZS).
merge(XS,[Y|YS],[Y|ZS]) :- true | merge(XS, YS?, ZS).
merge([],YS,YS) :-true|true.
merge(XS,[],XS):-true|true.
```

et la question est merge(XS?,YS?,Z).

Au contraire de Parlog, l'évaluation des gardes en Concurrent Prolog

peut instancier des variables du but. Toutefois, ces instances ne doivent pas être visibles avant la sélection de la clause, dont l'exécution se poursuivra. Aussi, la gestion d'environnements multiples est nécessaire afin de préserver le parallélisme OU dans l'évaluation des clauses alternatives. Ce mécanisme autorise des instanciations cachées et temporaires des variables du but jusqu'à l'élection d'une clause. Il y a donc gestion d'environnements multiples car locaux à chaque clause.

Guarded Horn Clause ne requiert aucune annotation supplémentaire. Toutefois, toute tentative d'instanciation d'une variable de l'appelant lors d'une unification, ou d'une évaluation d'une garde entraîne la suspension du littéral la demandant. Le programme merge s'écrit alors :

```
merge([A|X],Y,OZ):-true| OZ=[A|Z] merge (X,Y,Z).
merge(X,[A|Y],OZ):-true| OZ=[A|Z] merge (X,Y,Z).
merge([],Y,OZ) :- true | OZ=Y.
merge(X,[],OZ) :- true|OZ=X.
```

La présentation de ces langages amène les constations suivantes :

- Les langages gardés ont abandonné le non-déterminisme des langages de programmation logique au profit du **don't care non determinism.** En effet, le choix d'une alternative dans un paquet n'est jamais remis en cause même en cas d'échec. Toute notion de retour en arrière est donc éliminée.

- Le parallélisme OU est restreint à l'évaluation des têtes et gardes de clauses.

- Le parallélisme OU de ces langages, même restreint, entraîne le problème classique de gestion d'environnements multiples. Afin de pallier ce problème, le concept de garde sûre a été introduit. Une garde est sûre si elle n'instancie aucune variable de l'appelant. Le concept est étendu aux programmes, aux langages. Un programme est sûr si toutes les clauses qui le constituent sont sûres. Un langage est sûr si tous les programmes écrits en ce langage sont sûrs. Aussi, Parlog et GHC sont des langages sûrs : ainsi, en Parlog, les clauses non sûres sont détectées à la compilation et rejetées. De même, en Guarded Horn Clause, la notion de sûreté est garantie lors de l'exécution par le mécanisme de suspension des buts précédemment décrit. Concurrent Prolog, quant à lui, nécessite une gestion d'environnements multiples. Afin de pallier ce problème, un nouveau langage, en l'occurrence, Safe Concurrent Prolog, SCP, a été développé. Ce langage supporte le parallélisme OU et le parallélisme ET. La différence principale entre CP et SCP est une annotation, notée "^", appelée "write enable". Cette annotation distingue les variables de sortie des variables d'entrée. Les variables de sorties, annotées "^",

apparaissent en tête de clause, toutes les autres variables sont des variables d'entrée. L'annotation write enable désigne donc statiquement les variables qu'une garde peut instancier. Ce mécanisme proche de celui de Parlog assure que toute clause écrite en SCP est sûre.

B. Mise en œuvre des langages gardés.

Les langages gardés font l'objet de nombreuses études. Ainsi, Levy [Levy 86] et Crammond [Cram 86] ont développé des modèles de calcul parallèle pour ces langages. Des implantations sur machine mono-processeurs et multi-processeurs ont été réalisées. Dans le cadre de cette étude, nous nous intéresserons plus particulièrement aux travaux de l'ICOT. Nous évoquerons deux projets distincts : le premier maintenant abandonné, visait la définition d'une machine dataflow parallèle, le second définit la machine PIM, Parallel Inference Machine.

* PIM_D [Ito 83] est une machine parallèle DATAFLOW qui intégre les trois types de parallélisme suivants : le parallélisme ET, le parallélisme OU, et le parallélisme à l'unification. Le programme Prolog est compilé en un graphe dataflow dont les nœuds sont les opérations élémentaires à effectuer. Sur ce graphe circulent des jetons, en l'occurrence des opérandes. Le graphe est exploré en paralléle, un opérateur s'exécutant dés qu'il reçoit ses opérandes.

L'architecture développée se compose de grappes. Une grappe est elle-même constituée de quatre modules de mémoires et de quatre modules de traitement interconnectés. Un module de traitement se divise lui-même en plusieurs unités qui sont respectivement l'unité de contrôle des instructions (UCI), l'unité de d'exécution, l'unité de mémoire locale, et d'un buffer qui enregistre les paquets résultats. L'UCI prend un paquet en entrée, détecte la présence éventuelle des opérandes de l'instruction concernée et envoie le cas échéant l'instruction à l'unité d'exécution correspondante. Celle-ci décode l'instruction, l'exécute, et génère un paquet résultat qui est envoyé dans le réseau. Si l'un (au moins) des opérandes manque, les opérandes déjà présents sont mémorisés en attendant d'être pris en compte une nouvelle fois.

Un tel modèle présente un très fin grain de parallélisme. Toutefois, il entraîne la gestion d'un volume important d'informations, notamment la création de paquets en grand nombre.

* La machine PIM quant à elle se compose d'environ 12 grappes d'une dizaine de processeurs chacune. Le langage de base de cette machine est le langage KL1, dérivé du langage GHC. Ce langage se distingue par la platitude de ses gardes; En effet, celles-ci ne font appel qu'à des prédicats prédéfinis (=, <, >), évitant par ainsi la création d'un système hiérarchisé de gardes. Le projet suscite deux études distinctes,

en l'occurrence l'exécution de KL1 sur un élément de la machine PIM, et celle de KL1 en milieu distribué.

- La première approche est celle de PIM-Cluster [Sato 87], qui propose une exécution parallèle de KL1, exécution qui exploite au maximum les mémoires locales de chaque processeur. A cette fin, le processeur gère une file d'attente des processus prêts à être exécutés. Lorsque cette file d'attente devient vide, le processeur accède alors à un processus relevant d'un autre processeur. Les processus pouvant être exécutés par des processeurs autres que celui qui les a créés, sont estampillés par un pragma, noté @, et placés dans une file d'attente spéciale, accessible des autres processeurs. La méthode est efficace à condition que les pragmas soient associés à des processus donnant lieu des calculs importants [Chas 89a].

- La seconde approche est celle de la machine Multi-Psi, où le langage KL1 est exécuté sur un réseau de machines PSI. Ces machines PSI, séquentielles, mono-utilisateur et multi-traitements, peuvent supporter jusqu'à l'exécution simultanée de 64 processus. Ces machines n'ont pas de mémoire commune : en conséquence, elles doivent accéder à la mémoire locale des autres machines. Il faut limiter de tels accès qui se révèlent coûteux. La solution proposée par Ichioshy [Ichi 87] consiste à utiliser des pragmas explicites. A l'aide de ces derniers, le programmeur distingue les processus qui peuvent être exportés de ceux qui sont à exécuter localement, par le processeur qui les a créés. La gestion dynamique de la répartition des tâches est ainsi simplifiée. Le travail du programmeur est quant à lui plus complexe.

Actuellement, une première version de cette machine, Multi_PsiI, fonctionne. Une seconde version Multi-PsiII, est à l'étude. Elle est basée sur une vingtaine de machines PsiII, supposées être trois fois plus rapides que les machines précédentes. En particulier, le nombre de processus s'exécutant simultanément n'est plus limité.

III.3.b. Les modèles non déterministes.

Ces modèles visent à produire l'ensemble des solutions répondant à un problème donné. On distingue deux approches principales : les modèles consistant en un parcours de l'arbre ET/OU (voire du graphe de Kowalski) à l'aide de processus communiquant par messages, et les modèles multi-séquentiels.

A. Les modèles basés sur l'arbre ET/OU.

a) Le modèle de Conery

Un des premiers modèles théoriques conçus est celui de Conery [Cone

81, 83] qui intègre à la fois du parallélisme ET et du parallélisme OU. L'interprète parallèle développé est basé sur la notion de processus indépendants, qui communiquent par messages. Un processus OU est créé afin de résoudre un littéral unique, et un processus ET est créé pour résoudre la conjonction de sous buts dans le corps d'une clause. Lors de la résolution d'un sous but bi, un processus OU crée ses descendants, en l'occurrence des processus ET. Ces derniers résoudront le corps des clauses, dont la tête s'unifie avec bi, en créant autant de processus OU que de littéraux dans le corps de clause.

Le parallélisme OU et le parallélisme ET sont obtenus en traitant parallèlement les descendants respectifs de plusieurs des descendants des processus OU et des processus ET (sinon tous).

Initialement, le parallélisme ET n'est pas exploité : les sous buts sont satisfaits selon leur ordre d'apparition (les processus OU correspondants sont donc créés un à la fois). Si l'un des processus fils retourne un échec, le processus ET procède au backtracking en sollicitant de nouveau le prédécesseur du fils responsable de l'échec.

Le parallélisme ET est ensuite exploité : l'ensemble des processus fils d'un processus ET sont créés, entraînant ainsi l'apparition de processus inutiles ou ne se terminant pas, et la gestion délicate des solutions transmises par des sous buts partageant certaines variables.

Une solution intermédiaire consistant à déterminer les processus à exécuter séquentiellement, et ceux à exécuter parallèlement est alors proposée. Elle repose sur les deux idées principales suivantes :

- A chaque variable d'une clause, correspondent deux types de littéraux : un littéral unique, producteur de l'instance de la variable, et des littéraux consommateurs de cette instance. Conery évoque trois règles de recherche des littéraux producteurs et consommateurs :

  a) La tête de clause est le principal générateur des instances d'une variable.

  b) La plupart des interpréteurs ont des procédures prédéfinies qui sont unidirectionnelles.

  c) Le littéral le plus à gauche du corps de clause, dans lequel on trouve une variable non instanciée, est le producteur de cette variable. Cette règle utilisée lors de l'échec des deux premières garantit l'existence d'un littéral producteur pour chaque variable.

L'identification des littéraux producteurs et consommateurs entraîne

la construction d'un graphe dataflow. Celui-ci sert à ordonner la création des processus OU : un processus OU n'est engendré que si pour chaque variable consommée par le littéral existe une instance.

- Un ordre linéaire des littéraux, et la mise en oeuvre de messages RESET, assure une bonne gestion des retours arrière. Lors de la réception d'un message RESET par un processus, ce dernier retourne la première des solutions qu'il a déterminées et mémorisées.

L'échec d'un processus OU entraîne la recherche du premier littéral Li producteur susceptible de fournir une instance. Ce littéral reçoit un message REDO. De plus, tous les littéraux successeurs de Li reçoivent un message RESET (sont de nouveau activés) s'ils ne consomment pas les instances produites par Li, ou sont remplacés par de nouveaux processus OU dans le cas contraire.

b) Le modèle COALA

Un second modèle COALA (Calculateur Orienté Acteur pour la Logique et ses Applications) [Perc 86] [Dura 86] est étudié à Toulouse. Ce modèle d'interprétation répartie de Prolog s'appuie sur les deux concepts principaux suivants :

- le graphe de connexion [Kowa 79]
- la notion d'acteur.

Le programme source est en premier lieu précompilé en un graphe de connexion ET/OU. Les nœuds de ce graphe sont les clauses du programme, les arcs du graphe relient deux littéraux unifiables, qui sont respectivement le littéral en cours d'évaluation et la tête de clause candidate à l'unification. Chaque arc est étiquetté par les liaisons des variables, liaisons résultant de l'unification entre les extrémités de l'arc.

Le graphe initial de connexion du programme Prolog dont la question est :- p(a,c), est donné en figure 7.

```
p(X,Y) :- q(X,Z), r(Z,Y).
q(a,b).
q(b,c).
r(a,b).
r(b,c).
r(c,d).
```

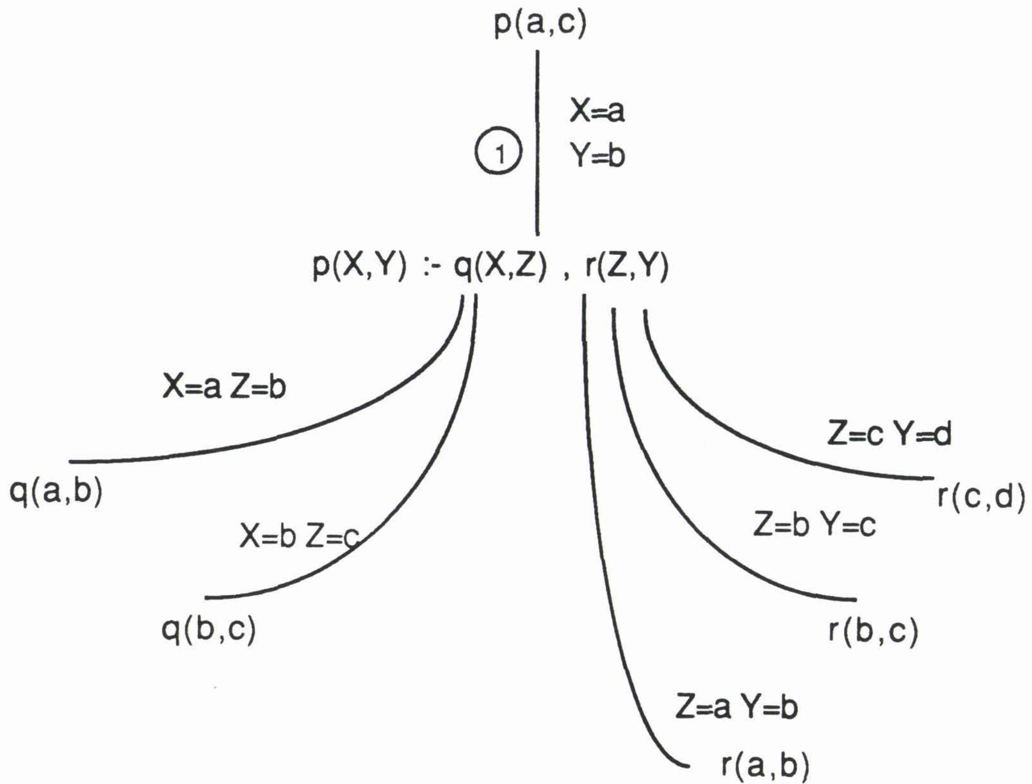


Figure 7 : Graphe initial.

La résolution modifie alors dynamiquement le graphe par ajout et suppression d'arcs. Ainsi, la sélection de l'arc 1, dans le graphe de la figure 7 entraîne l'obtention du graphe suivant (figure 8).

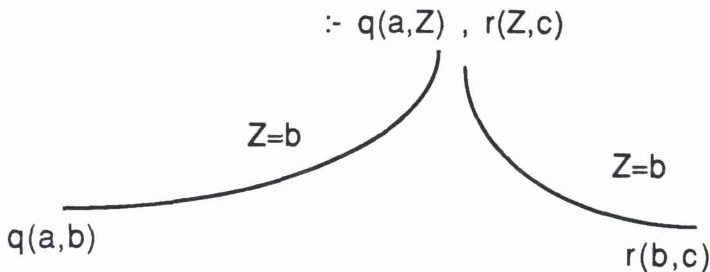


Figure 8 : Graphe obtenu aprés sélection de l'arc 1.

La résolution se poursuit en choisissant un arc issu de la résolvante <- q(a,Z), r(Z,c) et se termine quand le graphe ne comporte plus aucun arc. L'apparition de la clause vide souligne la découverte d'une solution au problème initial.

Le parallélisme ET, et le parallélisme OU sont liés à la possibilité de traiter simultanément plusieurs arcs. Le parallélisme OU s'obtient par le choix parallèle des arcs issus d'un même littéral, le parallélisme ET par l'unification parallèle de l'environnement de l'arc choisi, avec les environnements des arcs de la résolvante.

B. Modèles Multiséquentiels.

Les modèles multiséquentiels ont un but principal, en l'occurrence le contrôle de la création des processus OU en fonction de la disponibilité des ressources, à savoir les processeurs et mémoires. A cette fin, le degré maximal de parallélisme, c'est-à-dire le nombre de processus OU simultanément actifs, est borné par le nombre des ressources disponibles.

Ces modèles reposent sur une stratégie mixte : en effet, la stratégie mise en oeuvre est à la fois séquentielle et parallèle. Ainsi, chaque processus est une instance de la machine de Warren, et adopte en conséquence une stratégie séquentielle, de recherche en profondeur d'abord, indépendante des autres processus. Au cours de sa résolution, il produit des points de choix, et ne développe que la première des résolvantes laissant les autres en attente. Ces résolvantes seront considérées ultérieurement, par un processus redevenu libre. Ce dernier part alors à la recherche d'une tâche à effectuer disponible. Il détermine un processus qui a créé un (voire plusieurs) point(s) de choix, laissant ainsi du travail à réaliser. Le transfert de ces résolvantes (voire des structures de données les concernant) apparaît comme la seule communication possible entre deux processus distincts, les amenant ainsi à partager une structure de données communes. Le choix d'une alternative par un processus libre constitue la stratégie parallèle évoquée précédemment.

Deux des objectifs communs de ces modèles sont de tenter de maximiser le temps passé à exécuter des tâches, d'autre part de minimiser le temps consacré au scheduling.

Nous allons maintenant décrire trois modèles multiséquentiels parmi ceux existants. Chacun de ces modèles présentent des caractéristiques spécifiques. Le premier modèle, SRI [Warr 87], est l'un des premiers modèles multiséquentiel à avoir été étudié, et a donné naissance au projet Gigalips. Le second modèle, PEPSys [Syre 87] [Baro 88], est l'un de ceux qui intègrent du parallélisme OU et du parallélisme ET restreint. Le troisième modèle, OPERA [Bria 90] étudie l'implantation d'un système multiséquentiel sur une machine sans mémoire commune en l'occurrence la machine SUPERNODE.

Avant d'aborder la présentation de ces modèles, nous préciserons les notions suivantes: liaisons profondes et superficielles, liaisons universelles et conditionnelles.

Ainsi, une liaison est profonde si la valeur de la liaison est mémorisée par le processeur qui réalise la liaison. Au contraire, une liaison est superficielle si la valeur de la liaison est écrite dans la cellule de la variable qui était libre. De même, une liaison est universelle lorsque la

variable concernée est liée et créée avant la rencontre d'un point de choix. A l'inverse, une liaison est conditionnelle si la variable créée est liée après la rencontre de ce point de choix.

a) Le modèle SRI

Dans le modèle SRI[Warr 87], on peut considérer de façon abstraite une étape de calcul comme un arbre comprenant des arcs et des nœuds. Chaque nœud est étiqueté par des tâches, des triplets (C,G,B) où :

- C représente une liste de clauses
- G est une liste de buts
- B est une liste de liaisons

Exécuter une tâche consiste à tenter la résolution des buts G dans le contexte B avec les clauses C. Les nœuds de l'arbre de recherche correspondent aux points de choix de la machine WAM, augmentés d'un certain nombre d'informations supplémentaires (pointeur sur le nœud père, ...). La racine de l'arbre est étiquetée par la tâche $(C_0,G_0, [])$ où G_0 est le but initial, C_0 la liste de clauses candidates à l'unification et [] la liste vide.

Chaque processeur virtuel maintient un tableau de liaisons afin de mémoriser les liaisons conditionnelles (c'est-à-dire les liaisons concernant les variables partagées avec une autre branche de l'arbre). Les liaisons universelles sont, quant à elles, effectuées en remettant à jour la valeur dans la cellule de la variable.

Un processeur virtuel redevenu inactif se déplace le long de l'arbre de recherche afin de commencer une nouvelle tâche et doit modifier son tableau de liaisons en conséquence. En effet, les deux tâches anciennes et récentes ont un nœud ancêtre commun. Toutes les liaisons de variables effectuées avant la rencontre de cet ancêtre (c'est-à-dire les liaisons universelles) doivent être conservées. Au contraire, toutes les liaisons réalisées entre le nœud ancêtre et le nœud labellé par l'ancienne tâche sont à retirer du tableau de liaisons. A leurs places seront copiées les liaisons faîtes lors du parcours entre le nœud ancêtre et celui étiquetté par la nouvelle tâche.

Ce modèle présente donc l'avantage d'être proche du modèle séquentiel. En particulier, les accès aux variables et les liaisons sont des opérations en temps constant. A l'inverse, le changement de tâches qui entraîne d'une part la défection des liaisons conditionnelles, d'autre part la recopie des liaisons relatives à la nouvelle tâche peut se révéler coûteux. En effet, le coût de changements de nœuds est proportionnel à la distance les séparant, ou plus précisément au nombre de liaisons sur le chemin les reliant.

b) Le modèle PEPSys

PEPSys (Parallel ECRC Prolog System)[Syre 87][Baro 88] est développé à l'ECRC depuis 1984. Il a entraîné la définition d'un langage parallèle de programmation logique, d'un modèle de calcul, d'une machine abstraite et donné lieu à plusieurs évaluations du système, dont une implémentation réelle sur MX500.

- Le langage PEPSys présente les caractéristiques principales suivantes :

* Une intégration de deux types de module : des modules séquentiels, qui utilisent Prolog séquentiel, y compris les effets de bord et des modules parallèles qui expriment le parallélisme OU du programme (y sont regroupées les clauses d'un prédicat).

* Un parallélisme ET restreint, indépendant, exprimé par un opérateur explicite, '#'. A l'aide de cet opérateur, le programmeur désigne explicitement les sous-buts qui ne partagent pas de variable, et qui peuvent donc être exécutés en parallèle.

- Le modèle PEPSys combine du parallélisme OU, du parallélisme ET et la séquentialité. Deux types de problèmes, à savoir le contrôle du parallélisme et la gestion des liaisons doivent être résolus.

* Le parallélisme OU n'est développé qu'en présence de ressources disponibles. Ainsi, un nœud qui exécute un prédicat OU parallèle crèe un point de branchement. Ce dernier a un rôle double : d'une part, il indique aux processus devenus libres le travail encore disponible, d'autre part il permet au processus créateur un retour en arrière. Tous les processus maintiennent une liste de points de branchement. Un processeur redevenant libre consulte sa liste et prend en compte la première tâche qu'il trouve. Si la branche exécutée par le processus échoue, ce dernier effectue un retour arrière sur le dernier point de branchement. Si ce dernier propose une alternative à traiter, le processus s'en empare. S'il ne reste aucun travail à effectuer, le processus attend que tous ses fils aient terminé leur tâche: il convient, en effet, de leur permettre l'accès aux variables qu'il détient.

Un processus OU qui prend en charge une nouvelle tâche est susceptible de lier des variables de la branche principale, c'est-à-dire des variables globales aux différentes branches. Or, seule la première clause est autorisée à effectuer ces liaisons: les autres n'en ont pas le droit. Afin de mémoriser les liaisons des variables globales, d'ailleurs régies par un mécanisme de liaison profonde, on associe une table de hachage, notée hash-window à chaque processus. Les liaisons des variables

locales sont quant à elles, gérées par un mécanisme de liaison superficielle. Toutes ces liaisons (locales ou globales) sont datées par un OBL (Or Branch Level). Cet OBL représente le nombre de points de choix rencontrés par le processus. Avant de manipuler une variable globale, un processus doit découvrir si la variable a fait ou non l'objet d'une liaison, et si tel est le cas reconnaître la validité de la liaison (c'est-à-dire, déterminer si la liaison est postérieure ou antérieure au point de branchement). Les OBL de la variable liée et de la hash-window sont comparés. Si aucune liaison superficielle n'est détectée, on accède à la hash-window du processus courant afin de vérifier qu'une liaison profonde, valide pour le processus, ne s'y trouve. En cas d'échec, la chaîne ascendante des hash-windows est alors parcourue.

* Le parallélisme ET restreint est régi par un mécanisme similaire à celui mis en œuvre pour le parallélisme OU. Le sous-but "gauche" d'un nœud ET est traité par le même processus que celui évaluant le nœud , les sous-buts "droits" ne sont examinés que si un ressource libre existe. Chacune des branches, gauches et droites, est à même de générer des solutions multiples. On réalise alors incrémentalement le produit croisé de ces solutions au fur et à mesure de leur "arrivée"; cette opération est d'ailleurs rendue plus complexe par l'asynchronisme du calcul des solutions dans les branches gauches et droites. Un processus, noté join_cell (ou cellule de jonction), est alors créé afin de poursuivre les deux branches ET parallèles. La figure 9 illustre le parallélisme ET développé: un but p possède deux sous-buts l et r, qui produisent deux solutions chacun.

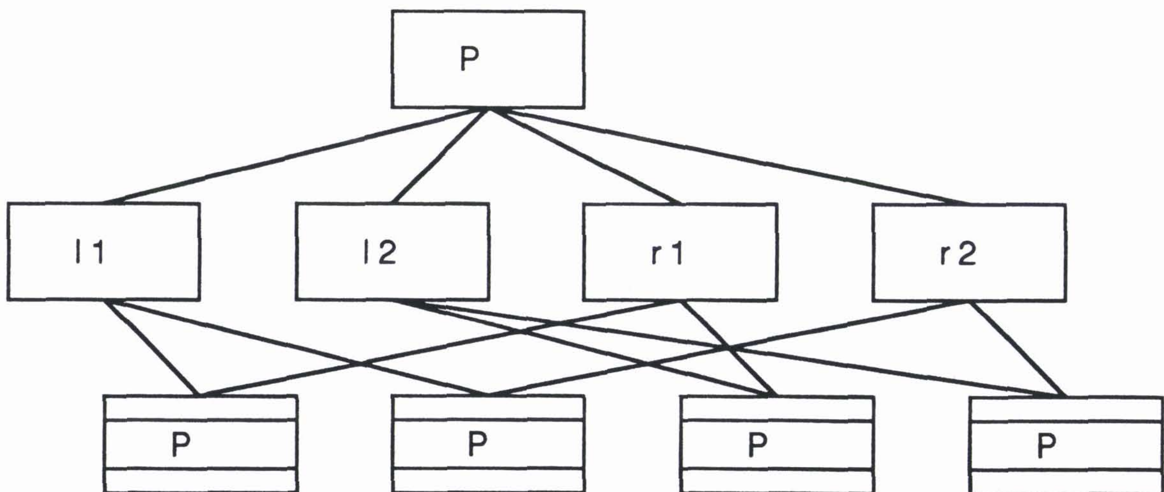


figure 9 : Parallélisme ET

Le produit croisé de ces solutions est ensuite mis en œuvre: les cellules à double barre symbolisent le "join cell", le littéral qui apparaît, dans ces cellules (dans notre exemple, le but p) indique la dernière

fenêtre de hachage commune.

La cellule de jonction se compose notamment d'un triplet de pointeurs vers des hash-windows qui sont respectivement les dernières rencontrées sur les branches gauches et droites, et la dernière de l'ancêtre commun. Ainsi, si l'on cherche une variable, on parcourt en premier lieu la branche droite jusqu'à l'ancêtre commun. Un échec aboutit au parcours de la branche gauche, un second échec au parcours de la branche ancêtre commune.

Le modèle PEPSys apparaît donc comme un modèle très complet qui intègre à la fois du parallélisme OU et du parallélisme ET. Le prix de cet aspect complet du modèle est la mise en place de mécanismes coûteux tels la recherche de liaisons au travers d'une chaîne de hash-windows pouvant être longue.

c) Le modèle OPERA

L'objectif d'OPERA [Bria 90] est la conception et l'implantation d'un système multiséquentiel OU parallèle sans mémoire commune. Les problèmes à résoudre, identiques à ceux posés les deux précédents projets, sont la définition d'une part d'une machine abstraite proche de la WAM, et d'autre part des règles de coopération entre processeurs.

- La machine abstraite étudiée, la TWAM (Transputer Warren Abstract Machine) possède un nombre de piles supérieur à celui de la WAM. Ainsi, si la pile de restauration reste inchangée, les piles locale et globale sont divisées en deux piles, respectivement la pile locale et la pile de choix et la pile globale et la pile variable. La pile locale ne contient plus que les environnements de clause et les points de choix étant mémorisés dans la pile de choix. La pile globale est réduite aux termes et valeurs, les variables logiques résidant en pile variable. La partition de la pile locale rend possible la distinction entre le travail en cours et le travail en attente. Celle de la pile globale facilite l'opération de déliaisons des variables liées postérieurement au point de choix concerné par l'exportation (et ce, par un mécanisme de datation des variables).

- Une communication entre TWAMs s'établit toujours entre une TWAM active, dite émettrice et une TWAM redevenue inactive, dite réceptrice, sur requête de cette dernière. Dès l'émission d'une copie des données qui décrivent les résolvantes à exporter, la TWAM émettrice reprend le travail en cours. La TWAM réceptrice ne peut commencer à travailler qu'à la fin du transfert des données, non sans avoir préalablement réinitialisé la pile des variables. Un conflit peut d'ailleurs survenir lorsque d'une part la TWAM émettrice termine sa résolvante avant la fin du transfert et d'autre part la résolvante émise était la seule

restant en attente.

Une partie contrôle centralisée, l'ordonnanceur ou scheduler, détermine une TWAM exportatrice à la demande d'une TWAM inactive, qui minimise le coût des transferts. Il effectue ce choix à l'aide d'informations (telles le nombre de points de choix en attente associée à une TWAM) recueillies en provenance des différentes TWAMs, et assure en outre, l'émission des ordres d'importation et d'exportation des données nécessaires à l'exécution de la tâche élue. Les informations traitées sont regroupées dans un descripteur d'état construit par un processus espion ,"SPY", associé à chaque TWAM. Celle-ci (figure 10) est composée des deux processus SOLVER et EXPORT.

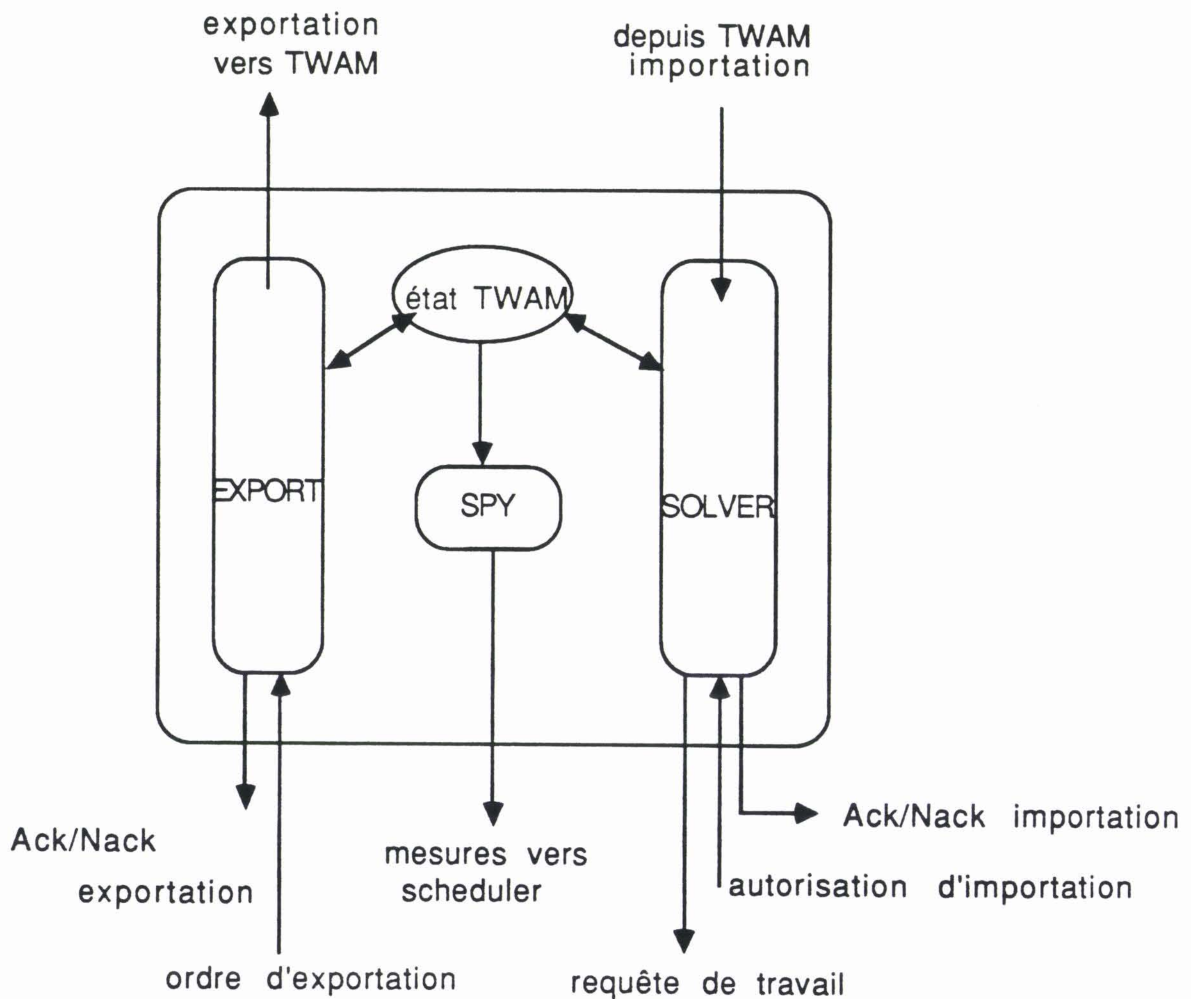


Figure 10 : Architecture d'une TWAM.

Le premier exécute séquentiellement le programme Prolog, le second réalise les transferts de données. Tous deux manipulent la pile de choix : le processus SOLVER produit les points de choix qu'il consomme selon la stratégie séquentielle, lors de retours en arrière. Le processus EXPORT

consomme ces points de choix en ordre chronologique lorsque la TWAM est élue pour exporter du travail. Le processus EXPORT considère donc les points de choix les plus anciens de la TWAM, c'est-à-dire ceux les plus proches de la racine de l'arbre de recherche.

Ce modèle, développé en C Prolog, a pour machine cible, SUPERNODE, machine multiprocesseur sans mémoire commune, constituée d'un réseau reconfigurable de transputers. Elle présente la particularité de pouvoir fonctionner en régime de connexion à la demande ; ainsi, les liaisons entre deux transputers sont établies au fur et à mesure de l'exécution du programme. Or, le modèle suppose l'éventualité de communications entre deux TWAMs quelconques à un instant donné. Il apparaît donc naturel de projeter une TWAM sur un transputer. C'est un commutateur de type Crossbar, commandé par un transputer spécialisé, dît transputer de contrôle, qui assure la reconfiguration complète du réseau en établissant (ou rompant) les liaisons physiques entre processeurs. Dès lors, le placement du scheduler devient immédiat : il convient de le placer sur le transputer de contrôle.

Le principal intérêt du projet OPERA réside dans le type de matériel utilisé : architecture sans mémoire commune, transputers et réseau reconfigurable. Il reste par contre de nombreux problèmes à résoudre dont notamment celui de la régulation de charge. Actuellement, un algorithme de type "naïf" est mis en oeuvre. Le scheduler reconnaît trois classes de TWAMS : les TWAMs inactives, les TWAMs muettes (celles qui n'ont pas de points de choix en attente), les TWAMs surchargées (celles qui ont au moins un point de choix en attente) et agit en conséquence. Les auteurs observeront les effets d'un tel mécanisme afin d'améliorer le degré de parallélisme obtenu par l'application de ce procédé.

IV. CONCLUSION.

La programmation logique offre de nombreuses sources de parallélisme. Les plus intéressantes d'entre elles, car les plus prometteuses, ont conduit soit à l'élaboration des systèmes gardés, s'il s'agit du parallélisme de flots, soit à l'élaboration des modèles non-déterministes et multi-séquentiels, s'il s'agit du parallélisme ET et du parallélisme OU.

Il est cependant un point sur lequel peu de modèles ont travaillé, à savoir l'ordre des solutions élaborées lors de l'exécution du programme.

Les modèles gardés ne tiennent pas compte de ce genre de problème. En effet, les langages gardés ont abandonné le non déterminisme inhérent à Prolog, en supprimant la notion de retour arrière. Aussi,

l'exécution d'un algorithme codé d'une part en Prolog, d'autre part dans l'un des langages gardés (quelle qu'elle soit) donne deux ensembles de solutions totalement distincts, aussi bien au point de vue du nombre de solutions que de leur ordre.

Les modèles non-déterministes, quant à eux, adoptent l'une des deux politiques suivantes : les uns ne se préoccupent l'ordre de ces solutions, les autres résolvent ce problème en introduisant un opérateur explicite.

Les chapitres suivants vont donc décrire un modèle d'exécution parallèle de Prolog, qui permet l'obtention d'un ensemble de solutions conforme à celui obtenu par l'exécution par un interpréteur Prolog séquentiel.

LE MODELE D'EVALUATION

LE MODELE D'EVALUATION.

I. INTRODUCTION.

Le projet LOGARCH définit d'une part un schéma d'évaluation parallèle de programmes Prolog, et d'autre part étudie une machine parallèle dédiée à l'exécution de programmes Prolog selon ce modèle d'évaluation.

Cette étude est consacrée à la définition d'un schéma d'exécution parallèle de Prolog. Ce schéma exploite le parallélisme OU. Ses trois principaux objectifs sont les suivants :

1. le parallélisme est exploité implicitement
2. la sémantique opérationnelle de Prolog est respectée
3. à tout moment, seul un nombre limité d'environnements est à gérer.

Ces contraintes impliquent les conséquences suivantes:

- la syntaxe du langage est identique à celle de Prolog: le programme source est donc vierge de toute annotation supplémentaire, similaire à celles utilisées dans les langages gardés. Le parallélisme n'est pas à la charge du programmeur: il est inhérent au programme,
- l'arbre ET/OU est parcouru comme il l'est lorsque le programme Prolog est exécuté séquentiellement.
- l'ensemble des solutions obtenues est conforme à celui construit lors d'une exécution séquentielle. L'ordre de ces deux ensembles est identique.

Ce chapitre présente le modèle d'évaluation proposé. En un premier temps, nous exposerons les caractéristiques détaillées du modèle: en particulier, seront décrits les types de parallélismes mis en oeuvre, ainsi que la gestion des environnements. Puis, nous décrirons les primitives nécessaires à la construction de l'arbre ET/OU, et le comportement de chaque classe de nœuds de l'arbre.

II. CARACTERISTIQUES DU MODELE

II.1 Le parallélisme OU restreint (DFOP)

Le parallélisme OU, comme nous l'avons vu au chapitre précédent, consiste en l'activation simultanée de plusieurs fils d'un nœud OU, donc en l'examen parallèle des alternatives qui constituent un prédicat. Cette

méthode se révèle intéressante lorsque les programmes exécutés sont de type non déterministe, c'est-à-dire fournissent plusieurs solutions.

La plupart des modèles proposés reposent sur un parallélisme OU déclenché lors de la phase descendante du parcours de l'arbre. Ceci induit un parcours en largeur d'abord, parcours qui est contraire à celui imposé par la deuxième contrainte. De plus, ce type de parcours entraîne une explosion du nombre de processus OU créés. Ceci implique la gestion simultanée d'un volume important d'informations car l'activation parallèle des nœuds conduit à établir des liaisons multiples pour les mêmes variables, donc à créer simultanément plusieurs contextes d'activation.

Afin de satisfaire les contraintes énoncées précédemment et de contrôler la gestion du nombre d'environnements, notre modèle exploite un parallélisme OU restreint , qui se développe par le bas de l'arbre. Afin de mieux contrôler la gestion des environnements, seul un nombre limité de branches OU est simultanément activé. La mise en oeuvre de cette activation restreinte est réalisée en utilisant un mécanisme de verrouillage, décrit dans le paragraphe II.3.

II.2 Le pipeline ET.

Le parallélisme ET, détaillé dans le chapitre précédent consiste en l'activation parallèle des fils d'un nœud ET (donc en l'activation parallèle d'un corps de clause).

L'intérêt de cette méthode est de trouver une accélération à l'exécution de programmes déterministes pour lesquels existe au plus une solution.

Le parallélisme ET, soulève le problème des variables partagées par au moins deux littéraux. Aussi, notre modèle exploite un pipeline ET d'activation, géré de gauche à droite conformément à la régle de sélection des littéraux : on sélectionne d'abord le littéral le plus à gauche de la résolvante courante, puis le suivant... Une relation de type producteur-consommateur est établie entre deux nœuds fils successifs d'un nœud ET: le premier élabore une solution partielle, consommée par le successeur. La construction d'une solution est certes séquentielle; toutefois plusieurs solutions sont construites simultanément grâce à un mécanisme d'anticipation.

Notre modèle s'avère donc bien adapté aux programmes non déterministes.

II.3 Le contrôle dirigé par la nécessité.

L'activité d'un nœud obéit à un mode de contrôle dirigé par la nécessité.

II.3.a L'activation des nœuds

Initialement, aucun nœud n'existe en dehors du nœud but. La création d'un nœud survient suite à la demande explicite du nœud père. Cette demande est assurée par l'intermédiaire d'une primitive, émise par le nœud père, qui véhicule l'ensemble des arguments nécessaires à l'activation du nœud.

II.3.b Consommation des environnements retour : Introduction d'un mécanisme de verrouillage.

Définition : Un environnement solution est l'ensemble des arguments instanciés lors de la résolution. Si la résolution échoue, l'environnement est dit environnement-échec.

Un nœud détient un environnement-solution suite à la réception d'une primitive émise par l'un de ses nœuds fils. L'environnement-solution est donc celui déterminé par le sous-arbre de racine le nœud fils émetteur de la primitive.

Afin de limiter à tout instant le nombre d'environnements à gérer, l'accumulation d'environnements sur un même nœud est interdite. A tout moment, un nœud ne traite qu'un seul environnement-solution. Le nœud consomme d'abord l'environnement qu'il possède. Puis il demande explicitement une nouvelle solution à l'un de ses nœuds fils. La remontée des environnements solutions vers un nœud est donc directement liée à la vitesse de consommation des environnements solutions par ce nœud : entre deux demandes explicites de solutions émises par le nœud, la remontée des environnements-solutions vers ce nœud est suspendue.

Un tel mécanisme de consommation des environnements favorise l'éclatement de la pile d'environnement.

Ce mécanisme est implanté par l'utilisation d'un verrou, tant au niveau du parallélisme OU restreint, qu'à celui du pipeline ET. Nous allons maintenant décrire ce mécanisme de verrouillage.

II.3.b.1 Le mécanisme d'activation restreinte du parallélisme OU.

Le parallélisme OU restreint se développe en deux phases:

a) une phase d'activation, ou phase descendante

b) une phase de résultat, ou phase ascendante.

* **La phase d'activation** : Lors de cette phase, un nœud OU n'active qu'un seul de ses fils, celui qui est le plus à gauche. Ce mécanisme respecte la stratégie de choix des clauses utilisée dans les interpréteurs séquentiels : cette dernière implique l'examen des clauses selon leur ordre d'apparition dans le programme. On obtient le schéma suivant (figure 1).

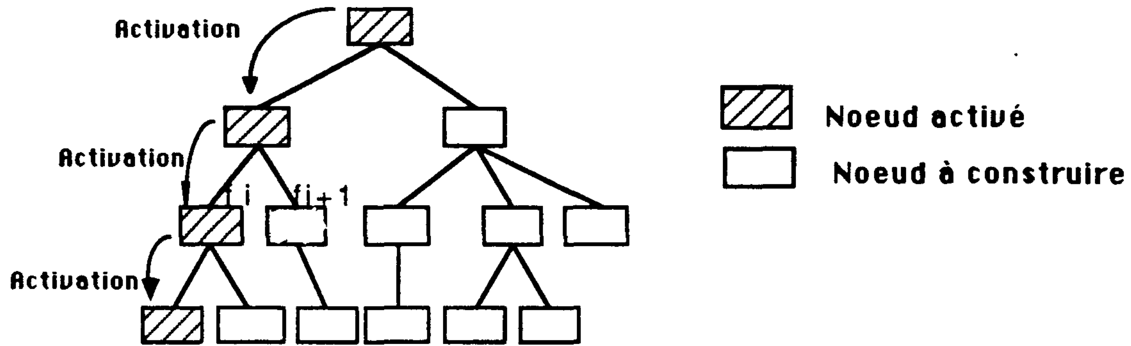


Figure 1 : Phase d'activation OU

La phase d'activation se termine dès qu'un nœud terminal est atteint. Ce nœud terminal, ainsi activé, évalue (ou non) un environnement solution (il a trouvé un échec), retourne ce résultat à son nœud père, puis il se détruit.

* **La phase de résultats.** : Un nœud OU reçoit le premier environnement solution du fils courant actif (soit f_i). Ce dernier reste actif, donc continue à élaborer des résultats. Mais il ne pourra transmettre une autre solution que sur demande explicite du nœud père. Il faut donc suspendre la remontée des solutions déterminées par ce nœud : ceci est réalisé grâce à la pose d'un verrou de solutions. Cette pose de verrou est consécutive à l'envoi par fi, de la solution courante (cf figure 2 : phase 1).

A la réception de ce premier environnement-solution, le nœud OU active alors le nœud fils suivant (f_{i+1}), s'il existe. Dès lors, deux branches indépendantes travaillent concurremment car une nouvelle phase d'activation a commencé dans le sous-arbre dont la racine est le nœud nouvellement activé (f_{i+1}).

Toutefois, le nœud OU ne peut prendre en considération les solutions élaborées par le nœud fils f_{i+1}, tant que le sous-arbre de racine f_i n' a pas été complètement exploré. Aussi, le nœud f_{i+1} est activé en mode restreint c'est-à-dire que l'activation s'accompagne de la pose d'un verrou de solutions, qui suspend toute remontée de solutions vers le père (cf figure 2 : phase 2).

Dès que le nœud OU a consommé la solution obtenue (en l'occurrence l'a retournée à son propre père), il peut traiter un nouvel environnement expédié par le nœud f_i : il lève le verrou de solution associé au nœud f_i (cf figure 2 : phase 3).

Il s'ensuit une phase de remontées de solutions entre le nœud f_i et le nœud OU : le nœud f_i envoie un environnement-solution au nœud OU, puis se bloque en remontée de solutions (le verrou de solutions est posé) et attend la prochaine levée de verrou par son père.

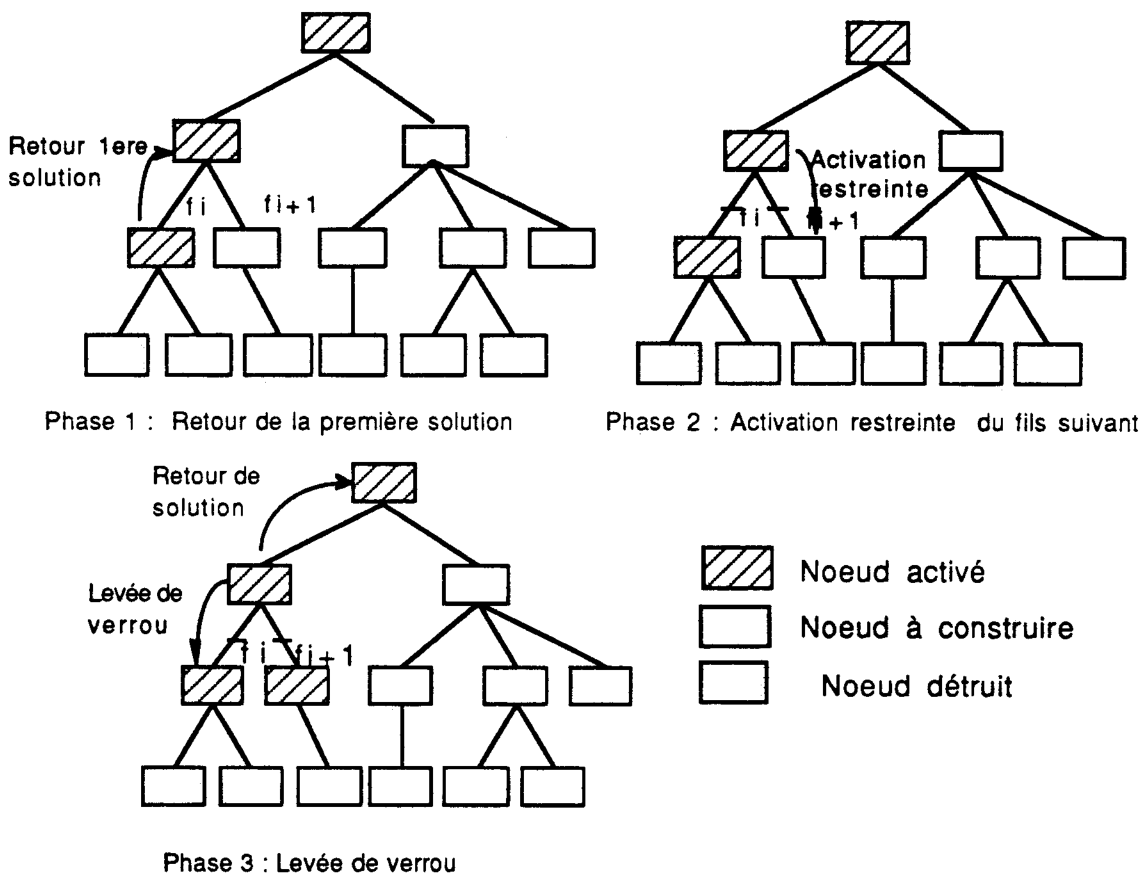


Figure 2 : Retour de la première solution

Cette séquence est itérée jusqu'à la transmission du dernier environnement solution du nœud fi jusqu'au nœud OU. Cette transmission assure que le sous-arbre de racine f_i a été complètement exploré. Aussi, d'une part le nœud f_i est détruit et d'autre part le verrou de solutions associé au nœud f_{i+1} est levé: le nœud OU prend alors connaissance des solutions élaborées par le sous-arbre de racine f_{i+1} (cf figure 3).

Le mécanisme décrit est récursif, permettant ainsi l'élaboration simultanée de plusieurs solutions. De plus, il présente un avantage double :

- les environnements restent distribués dans l'arbre de résolution
- la séquence de solutions obtenue est conforme à celle construite lors d'une exécution séquentielle du programme.

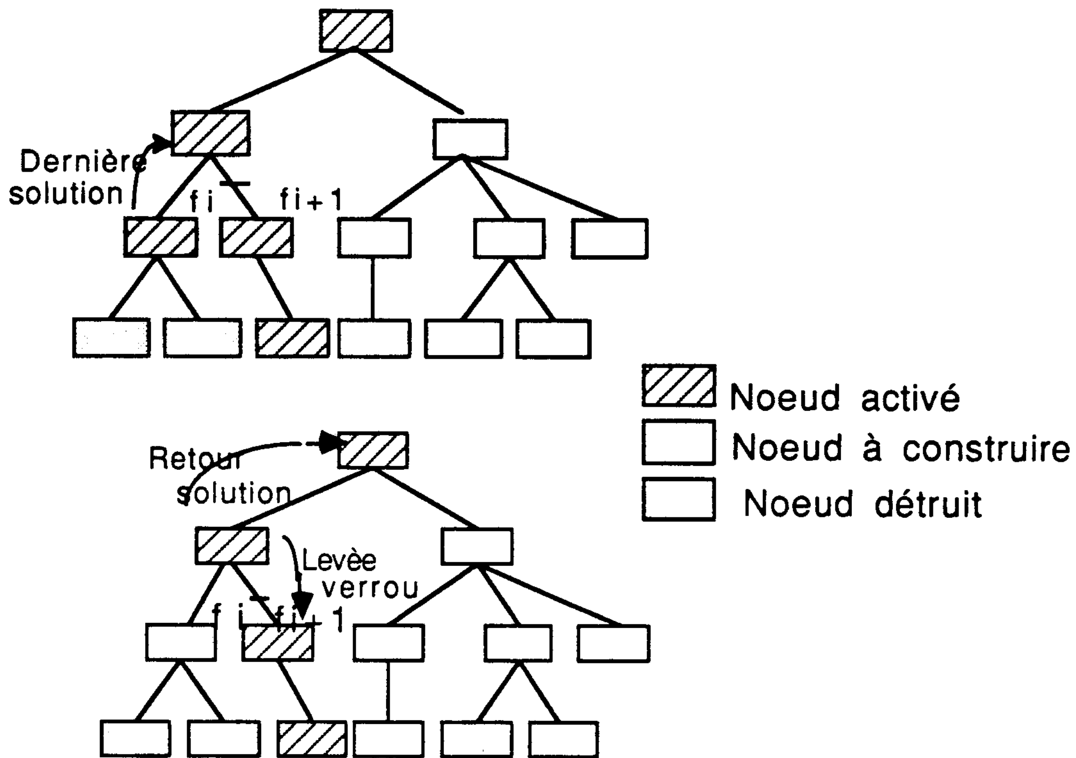


Figure 3 : Retour de la dernière solution

Les environnements-échecs nécessitent un traitement particulier : a priori, un échec n'est pas considéré comme définitif par le nœud qui le reçoit, et par conséquent n'est pas transmis au père du nœud. Ainsi, si le nœud f_i retourne un échec au nœud OU, ce dernier consulte le nœud fils suivant (en l'occurrence, f_{i+1}) afin d'obtenir un environnement autre qu' échec. Ce mécanisme est itéré jusqu'à obtention soit d'un environnement solution, soit d'un environnement échec définitif.

II.3.b.2 Le pipeline ET.

Un nœud ET qui vient d'être activé active le premier étage du pipeline ET, c'est-à-dire le premier de ses fils , f_1 puis se met en attente de la première solution calculée par f_1. Sur réception de cette solution, il active f_2, étage suivant du pipeline ET tandis que f_1, bien que restant actif, ne peut retourner d'autre solution que sur demande explicite du nœud ET. Au retour de la solution entre le nœud f_1 et le nœud ET, a succédé la pose du verrou de solution associé au nœud f_1, dont l'état devient activé en mode restreint (cf figure 4). Le mécanisme est itéré sur chaque retour de première solution, élaborée par les (n-1) premiers nœuds fils du nœud ET.

Lorsque le dernier étage du pipeline retourne sa première solution, le nœud ET dispose d'un environnement complet qu'il retourne à son propre père.

Il s'ensuit alors une phase de remontées de solutions entre le nœud f_n et le nœud père ET, similaire à celle décrite dans le paragraphe précédent : l'émission d'un résultat par le nœud f_n vers le nœud ET est suivie d'une pose de verrou de solutions associé au nœud f_n, le nœud f_n attend alors la prochaine levée de verrou par le nœud ET, et ce jusqu'à transmettre le dernier environnement solution.

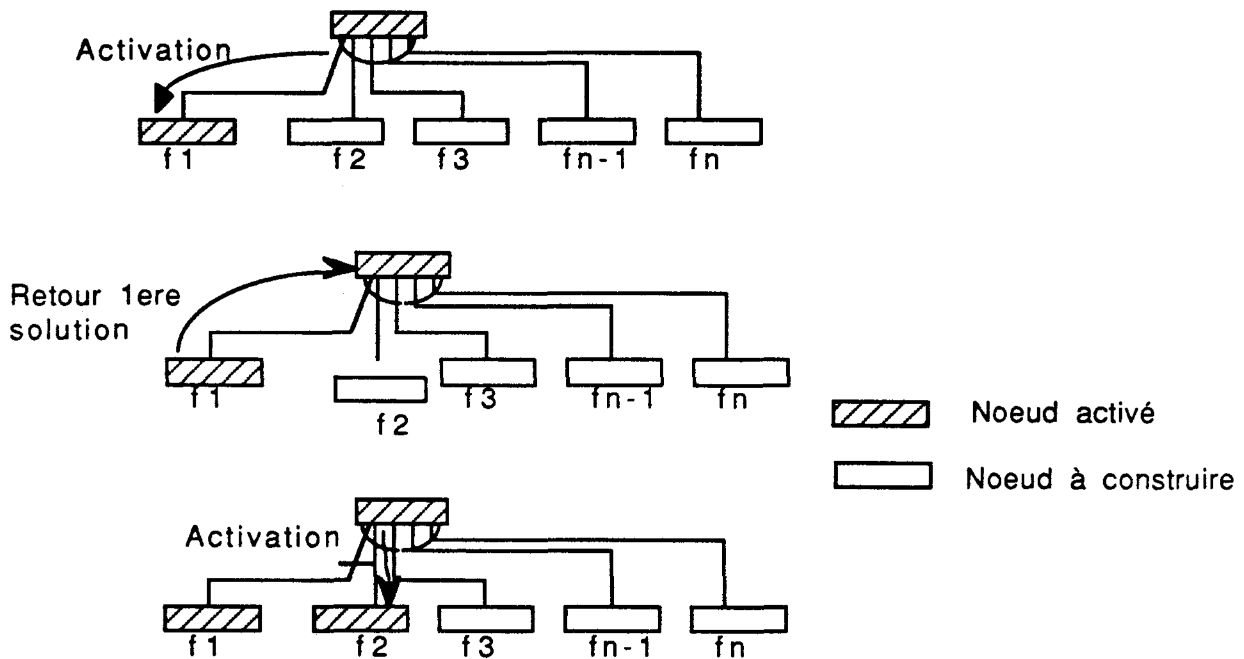


Figure 4 : Retour d'une première solution vers le nœud ET

Suite à cet événement, f_n redevient inactif il peut alors être détruit. Une nouvelle instance du nœud f_n est alors créée afin de consommer l'environnement suivant élaboré par le nœud f_{n-1}. Le nœud ET lève alors le verrou de solutions de f_{n-1}. f_{n-1} expédie la solution qu'il détient entraînant de nouveau la pose du verrou sur f_{n-1} et le nœud ET active le nœud f_n (figure 5).

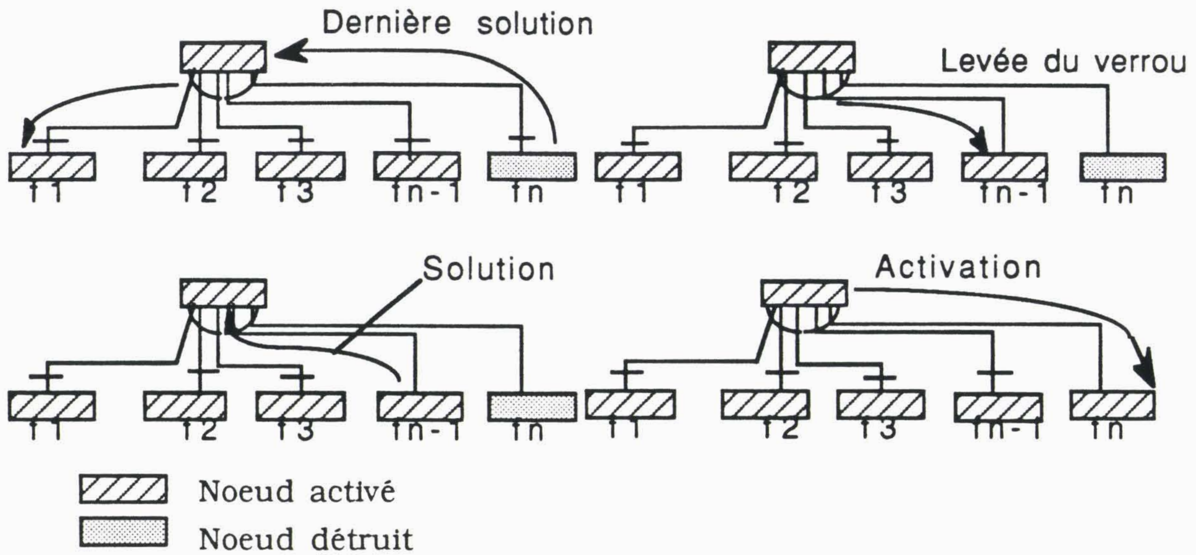


Figure 5 : Retour en arrière

Un mécanisme similaire est mis en place sur réception de la dernière solution d'un fils f_i (avec $i<n$). Une différence essentielle existe: le nœud f_i ne peut être activé qu'en mode restreint: l'activation s'accompagne donc de la pose d'un verrou. Le nœud ET lèvera le verrou lorsque le nœud f_{i+1} sera de nouveau inactif (figure 6).

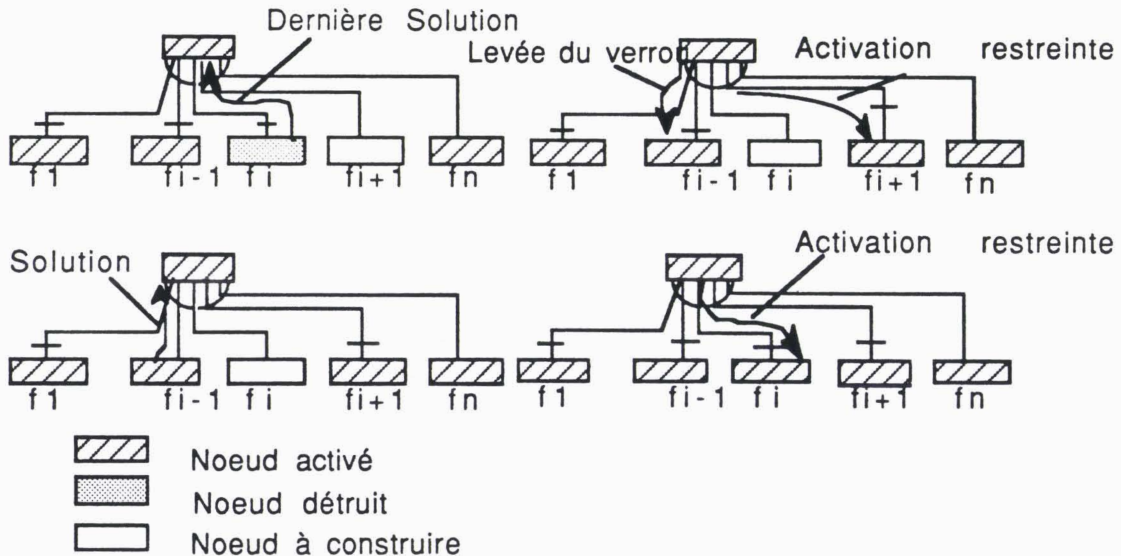


Figure 6 : Retour en arrière sur un fils quelconque

Ce mécanisme autorise l'anticipation du calcul des solutions : il est géré de droite à gauche, car il réalise le retour arrière inhérent à Prolog.

Remarque : La gestion du pipeline ET est conforme à la règle de sélection des littéraux en Prolog. Cette règle implique toujours le choix du littéral le plus à gauche du corps de clause.

III. LE MODELE FONCTIONNEL.

Notre modèle d'évaluation repose sur la construction dynamique d'un arbre ET/OU, tel que l'a introduit Conery (Cone 81). Dans un premier temps, nous détaillerons la compilation du programme source. Puis, nous décrirons les primitives de synchronisation nécessaires à la construction de l'arbre. Nous terminerons en présentant le comportement de chaque classe de nœud de l'arbre.

III.1. La compilation du programme source.

Le programme source est un ensemble de clauses de Horn constituées d'une tête de clause et d'un corps de clause. La tête de clause est réduite à un atome unique, un corps de clause est formé de plusieurs littéraux ou d'aucun. Les clauses d'un même paquet, c'est-à-dire les clauses dont l'atome de tête est le même, sont indicées selon leur ordre d'apparition dans ce paquet. Une clause est donc identifiée par la donnée d'un couple (Id_littéral, index) où Id_littéral et index désignent respectivement l'identificateur de l'atome de tête et le numéro d'ordre de la clause dans le paquet.

Une clause est compilée en un nœud ET, identifié par le couple (Id_littéral, index). Le nœud ET possède deux types de liens : un lien vers le nœud père, un lien par littéral du corps de clause.

Un prédicat est compilé en un nœud OU qui lui aussi a deux types de liens: un lien vers le nœud père, un lien par clause, qui définit le prédicat. A tout nœud correspond donc une table de liens où figurent les deux types de liens précédemment évoqués.

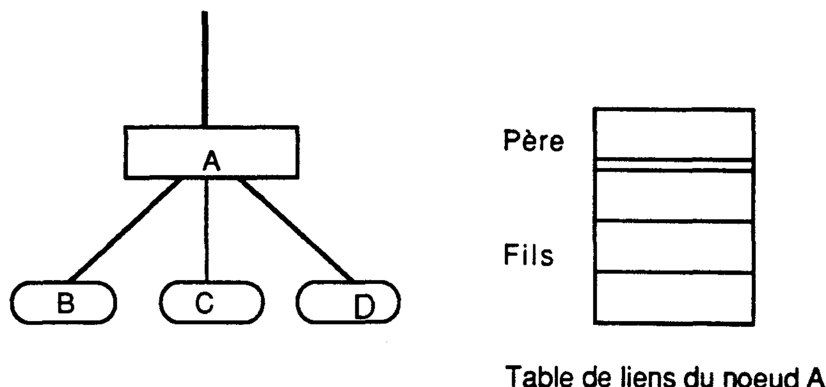


Figure 7 : Compilation d'une clause

La figure 7 présente la compilation de la clause a :-b, c, d.

A chacun des nœuds Et ainsi obtenu, est associé un environnement. Cet environnement est un vecteur, où figure l'ensemble des variables de la clause, numérotées selon leur ordre d'apparition dans la clause.

A chacun des liens fils est attachée la liste des argument du prédicat. Ainsi, la compilation de la clause : gpar(X,Y) :- par(X,Z) ,par(Z,Y) est donnée dans la figure 8.

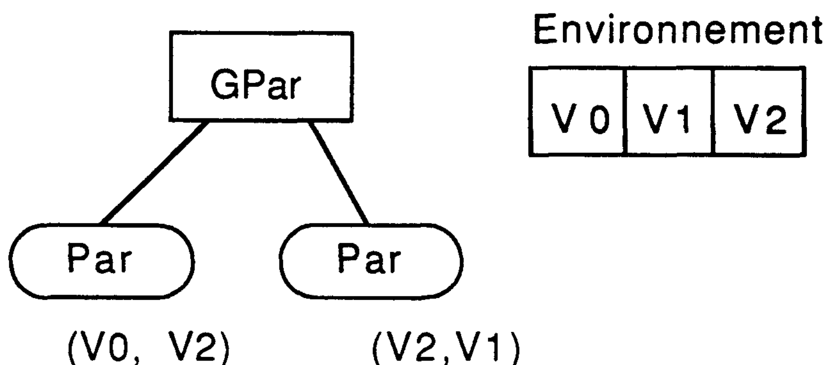


Figure 8 : Compilation d'une clause et environnement

De plus, chaque nœud est caractérisé par son état courant : l'état initial d'un nœud est l'état créé, l'état final l'état résolu. Les états intermédiaires tels les états actifs et en attente de communication seront décrits ultérieurement.

Comme nous l'avons vu précédemment, la construction de l'arbre est réalisée selon un mode de contrôle par nécessité. Un nœud n'agit que sur sollicitation extérieure, qui se traduit par la réception d'une primitive. Cette dernière est interprétée par le gérant du nœud, qui détermine ainsi le comportement futur du nœud. Dans un premier temps, nous détaillerons les primitives de synchronisation nécessaires à la construction de l'arbre ET/OU, puis l'algorithme du gérant d'un nœud.

III.2. Primitives de synchronisation du modèle.

L'étude précédente a fait apparaître quatre classes de primitives :

- les primitives d'activation
- la primitive demande de solution
- les primitives de résultat
- la primitive de destruction.

Chaque primitive étudiée sera donnée sous la forme suivante :

Id_prim(arg_1,...,arg_n)

où Id_prim est l'identificateur de la primitive et les différents arg_i (avec i compris entre 1 et n) les paramètres de la primitive.

II.2.a. Les primitives d'activation.

L'étude du parallélisme OU, ainsi que du pipeline d'activation a montré l'existence de deux types d'activation : l'activation immédiate et l'activation en mode restreint, qui sont respectivement exprimées par les primitives P_ACT et P_LOCK.

Ces deux primitives, liées à la phase d'expansion de l'arbre, créent une instance du nœud, à partir d'un modèle du graphe compilé. Cette instance devient l'un des nœuds fils du nœud qui a émis la primitive. Il y a alors association entre le lien père du nœud fils et le lien fils du nœud père (figure 9)

Les primitives P_ACT et P_LOCK ont les arguments identiques suivants :

- Id_émetteur, identificateur du nœud père. Cet argument désigne le nœud père à qui retourner les environnements qui seront déterminés.
- Id_destinataire est l'identificateur du modèle de nœud, dont une instance est à créer.
- L'ensemble des arguments nécessaires à l'activation du nœud.

La primitive P_ACT(Id_émetteur, Id_destinataire, Arguments) active le nœud nouvellement créé, sans positionner le verrou de solutions.

La primitive P_LOCK(Id_émetteur, Id_destinataire, Arguments) exprime l'anticipation du calcul des solutions, et l'ordonnancement de la séquence des solutions. Elle active l'instance du nœud nouvellement créé, en positionnant son verrou de solutions.

III.2.b. La primitive demande de solutions.

Cette primitive traduit donc la demande explicite de solutions d'un nœud père vers un nœud fils. Elle lève le verrou de solutions du nœud à qui elle est destinée. Cette primitive s'écrit sous la forme P_UNLOCK(Id_destinataire) où Id_destinataire identifie le nœud auquel la primitive est appliquée.

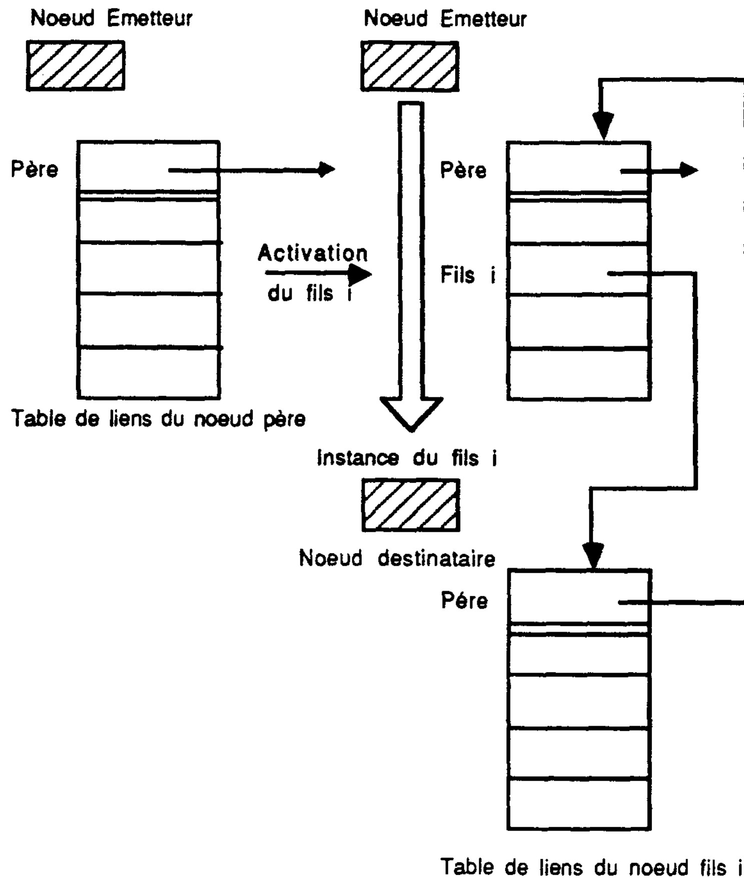


Figure 9 : Phase d'expansion de l'arbre.

III.2.c. Les primitives résultats.

Elles traduisent la phase de réduction de l'arbre, et sont au nombre de deux : P_RES, et P_ECHEC. La première retourne un environnement solution d'un nœud fils vers un nœud père, la seconde retourne l'échec du calcul vers le nœud père.

La primitive P_RES s'écrit P_RES(Id_émetteur, Id_destinataire, Attributs, Arguments). Id_émetteur est l'identificateur du nœud fils courant actif, Id_destinataire celui du nœud auquel la primitive est appliquée. Attributs est l'ensemble des deux informations suivantes : premier et dernier. Ces informations sont nécessaires à la détermination du type de la prochaine activation à effectuer sur le nœud fils suivant. Arguments est l'ensemble des arguments instanciés lors de la résolution.

La pose du verrou de solutions du nœud qui a émis la primitive est consécutive à l'émission de cette primitive.

La primitive P_ECHEC(Id_émetteur, Id_destinataire) entraîne la destruction du nœud émetteur de la primitive, de sa table de liens, ainsi

que la destruction du lien fils du nœud père.

Remarque: Les deux premières classes de primitives sont des primitives descendantes. Elles sont émises par un nœud père et appliquées à un nœud fils au contraire de la troisième classe de primitives qui sont des primitives ascendantes : émises par un nœud fils, elles sont appliquées au nœud père.

De plus, si l'attribut dernier de la primitive P_RES est vrai, le nœud émetteur de la primitive doit être détruit ainsi que sa table de liens : une primitive de destruction assure cette fonction. Le lien unissant le nœud émetteur et son nœud père est lui aussi détruit (figure 10). Il en est de même si la primitive traitée est une primitive d'échec.

III.2.d. La primitive de destruction.

La primitive de destruction s'écrit P_DEL. Elle ne prend aucun argument. Elle est émise par le nœud lui_même, consécutivement à l'envoi soit d'une primitive d'échec, soit d'une dernière primitive de résultat (l'attribut dernier est alors vrai) : le nœud est détruit ainsi que sa table de liens.

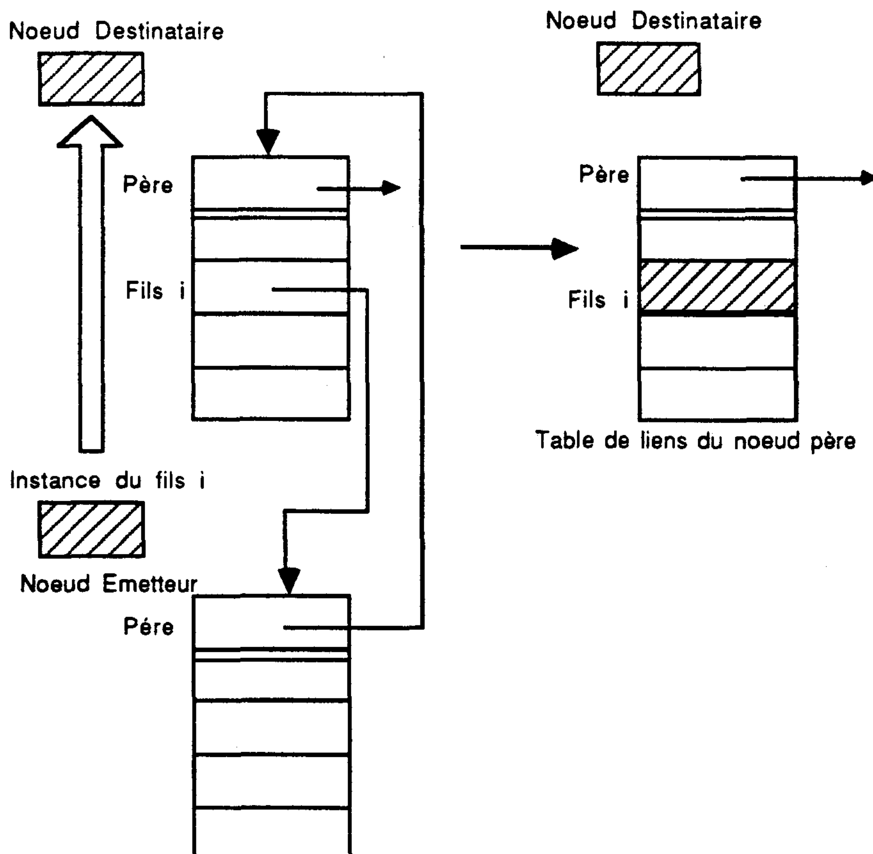


Figure 10 : Phase de réduction de l'arbre.

III.3. Le gérant d'un nœud.

L'arbre ET/OU est donc composé de deux types de nœuds distincts: les nœuds ET et les nœuds OU. Chacun de ces nœuds est représenté comme l'indique la figure 11.

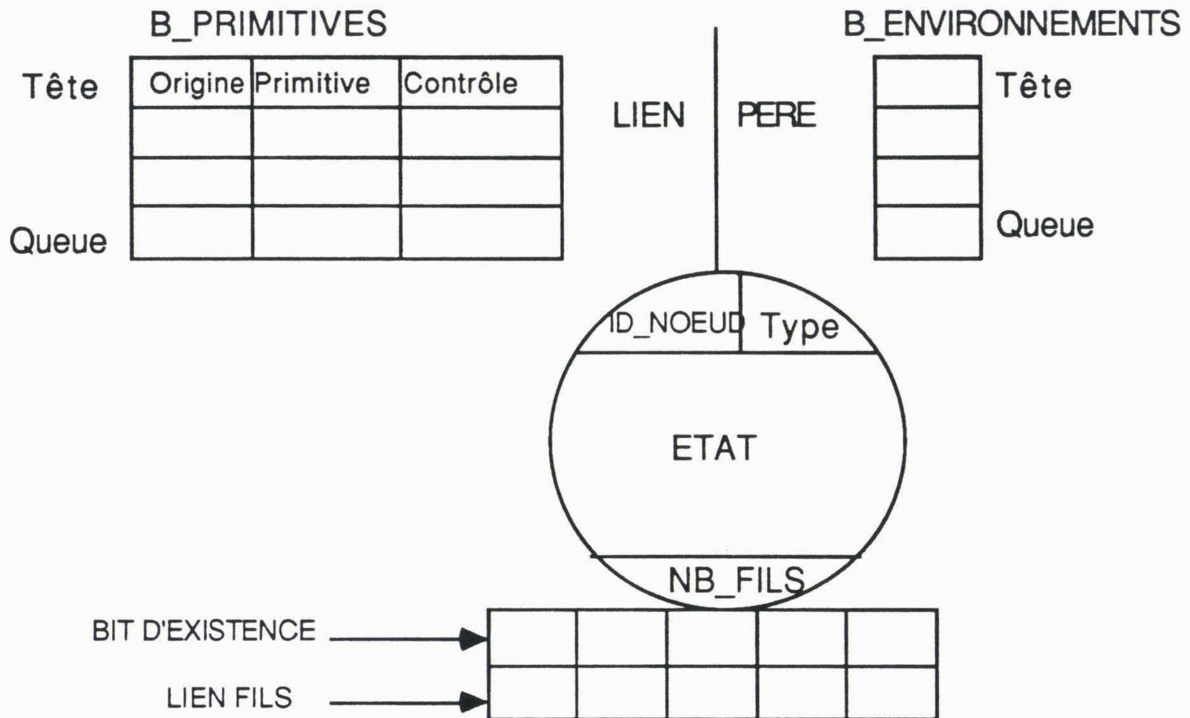


Figure 11 : Représentation d'un nœud.

B_PRIMITIVES est le buffer, où sont déposées les primitives appliquées au nœud. Ces primitives sont émises soit par le nœud père du nœud, soit par l'un des nœuds fils du nœud. Ce buffer peut contenir plusieurs primitives, qui sont interprétées dans l'ordre où elles sont déposées. Il est donc géré en file : une primitive est déposée en queue de file, et prélevée en tête de file. L'interprétation est réalisée mot à mot. Un emplacement du buffer est composé de trois parties différentes:

- une partie *origine* contenant l'identificateur de l'émetteur de la primitive,
- une partie *primitive* contenant l'identificateur de la primitive,
- une partie *contrôle*, qui spécifie les paramètres de contrôle de la primitive.

B_ENVIRONNEMENTS est le buffer qui contient les environnements solutions attendant d'être retournés au père, ainsi que les environnements d'activation.

ID_NOEUD est l'identificateur du nœud.

TYPE est le type du nœud ET ou OU.

NB_FILS représente le nombre total de nœuds fils.

ETAT est l'état courant du nœud.

L'état d'un nœud prend ses valeurs dans l'ensemble E={Créé, Attente_Solution, Attente_levée_verrou, Attente_Double, Résolu}, où

- **Créé** est l'état initial du nœud, précédant son activation
- **Attente_Solution** : le nœud attend une solution de l'un de ses nœuds fils.
- **Attente_Double** : le nœud attend conjointement une solution de l'un de ses nœuds fils, et une demande de solution émanant du nœud père.
- **Attente_Levée_Verrou** : le nœud attend une demande de solutions émanant du nœud père afin de lui retourner l'environnement solution mémorisé dans le buffer d' environnements.
- **Résolu** : état final du nœud, précédant sa destruction.

Tous les états, en particulier Attente_Solution, Attente_levée_verrou, Attente_Double, sont exclusifs les uns des autres.

Les *bits d'existence* et les *liens fils* constituent la *table d'existence*, dont le nombre d'entrées correspond au nombre de fils du noeud. Un bit d'existence est à zéro si le nœud fils n'existe pas ou plus, c'est-à-dire s'il n'a pas été activé ou s'il a été détruit. Lorsque le bit d'existence est à zéro, aucun lien fils n'existe.

A l'inverse, le bit d'existence est à un lorsque le nœud a été activé en mode restreint ou non.

Remarques :

1) Initialement, lors de la création du nœud, tous les bits d'existence sont à zéro. Il en est de même lorsque le nœud est dans l'état résolu.

2) La valeur du bit d'existence peut être indifférente dans l' interprétation de la primitive, la valeur adoptée est alors notée I(Indifférent).

Chacun des nœuds OU et des nœuds ET est régi par un gérant. Ce dernier effectue la séquence d'actions suivante: il prélève une primitive en tête du buffer de primitives du nœud, interprète le comportement du nœud en fonction de plusieurs facteurs dont l'état courant du nœud. Le traitement s'interrompt lorsque le nœud est en passe d'être détruit (en d'autres termes, le nœud est dans l'état résolu). L'algorithme du gérant d'un nœud est donc:

```
tant que état(nœud) <> résolu
faire
    tant que buffer_primitives(nœud)  vide
    faire
        attendre;
    fait;
    prélever(primitive,buffer_primitives);
    interpréter(primitive);
fait;
```

Les deux chapitres suivants sont respectivement consacrés à la description détaillée des gérants des nœuds OU et des nœuds ET.

IV. CONCLUSION.

Ce chapitre décrit le modèle d'évaluation parallèle que nous avons défini pour le langage Prolog. Ce modèle respecte la sémantique opérationnelle de Prolog; en effet, la règle d'activation des nœuds est conforme à la règle de sélection des buts et des clauses. Ainsi, un nœud active toujours en premier lieu le nœud fils le plus à gauche. Or, les nœuds OU (voire les nœuds ET) sont les transformés respectifs des littéraux d'un corps de clause (voire de la tête de clause). L'activation du nœud ET, fils le plus à gauche d'un nœud OU, correspond à la sélection de la première des clauses d'un paquet, et celle du nœud OU, fils le plus à gauche d'un nœud ET, à celle du premier littéral d'un corps de clause.

De plus, le modèle extrait automatiquement le parallélisme du programme. Ainsi, c'est le retour d'une première solution calculée par un nœud fils, vers un nœud OU, qui déclenche l'activation du nœud fils suivant de type ET (c'est-à-dire l'évaluation d'une nouvelle alternative du paquet). Cette opération ne requiert aucunement la présence d'un opérateur explicite de contrôle du parallélisme.

Les deux points précédents impliquent donc que notre modèle permet l'exécution de tout programme écrit en Prolog pur. L'obtention d'un modèle complet exige l'introduction de prédicats particuliers, tel que les prédicats à effet de bord comme le montrera le chapitre V.

En dernier lieu, si le mécanisme de verrouillage permet la production des solutions dans un ordre identique à celui obtenu lors d'une exécution séquentielle, il entraîne par contre **une diminution du taux moyen de parallélisme,** par rapport à un modèle ne tenant pas compte de cette contrainte. Ce point sera abordé dans le chapitre VI.

Le modèle proposé peut d'ailleurs être généralisé. Afin de clarifier la présentation, nous avons supposé égal à un le nombre de branches acti-

vées en mode restreint par un nœud OU. Ce nombre peut en fait varier, jusqu'à égaler le nombre de fils du nœud. Cette variante implique peu de modifications. Il est à remarquer que dans ce cas, notre modèle se rapproche d'un modèle OU total.

LE GERANT D'UN NŒUD OU

LE GERANT D'UN NŒUD OU.

I. INTRODUCTION.

Le chapitre précédent a montré que l'activité d'un nœud de l'arbre ET/OU, construit selon le schéma d'évaluation proposé, obéit à un mode de contrôle dirigé par la nécessité. La sollicitation extérieure à laquelle est soumise un nœud de l'arbre se manifeste sous forme d'une primitive, qui est interprétée par le gérant du nœud. Chaque type de nœud a son propre gérant.

Ce chapitre est consacré à l'étude du gérant d'un nœud OU. Nous détaillerons donc l'interprétation de chacune des primitives décrites auparavant. En particulier, nous nous attacherons à préciser les principaux critères (tels l'état courant du nœud, les contenus respectifs de la table de liens et du buffer d'environnements du nœud) dont dépend l'interprétation de la primitive courante, ainsi que les changements d'états du nœud, les mises à jour de la table de liens et la nature des primitives éventuellement émises.

II. LA PRIMITIVE P_ACT(Nœud_père, Nœud, Arguments).

Le nœud père exécute la primitive P_ACT (Nœud_père, Nœud, Arguments): il crée le nœud fils. L'état de ce nœud est initialisé à Créé. Dans le nœud qui exécute la primitive P_ACT, le bit d'existence correspondant au nouveau fils est positionné. Le lien fils du nœud père et le lien père du nœud fils sont associés. La primitive P_ACT et l'environnement d'activation sont respectivement déposés en queue des buffers de primitives et d'environnements du nœud fils. Le gérant du nœud prélève la primitive dans le buffer de primitives, puis construit une primitive d'activation destinée au premier de ses nœuds fils, soit Fils1. Le nœud attend alors la première des solutions évaluées par ce nœud fils : il est en Attente_Solution.

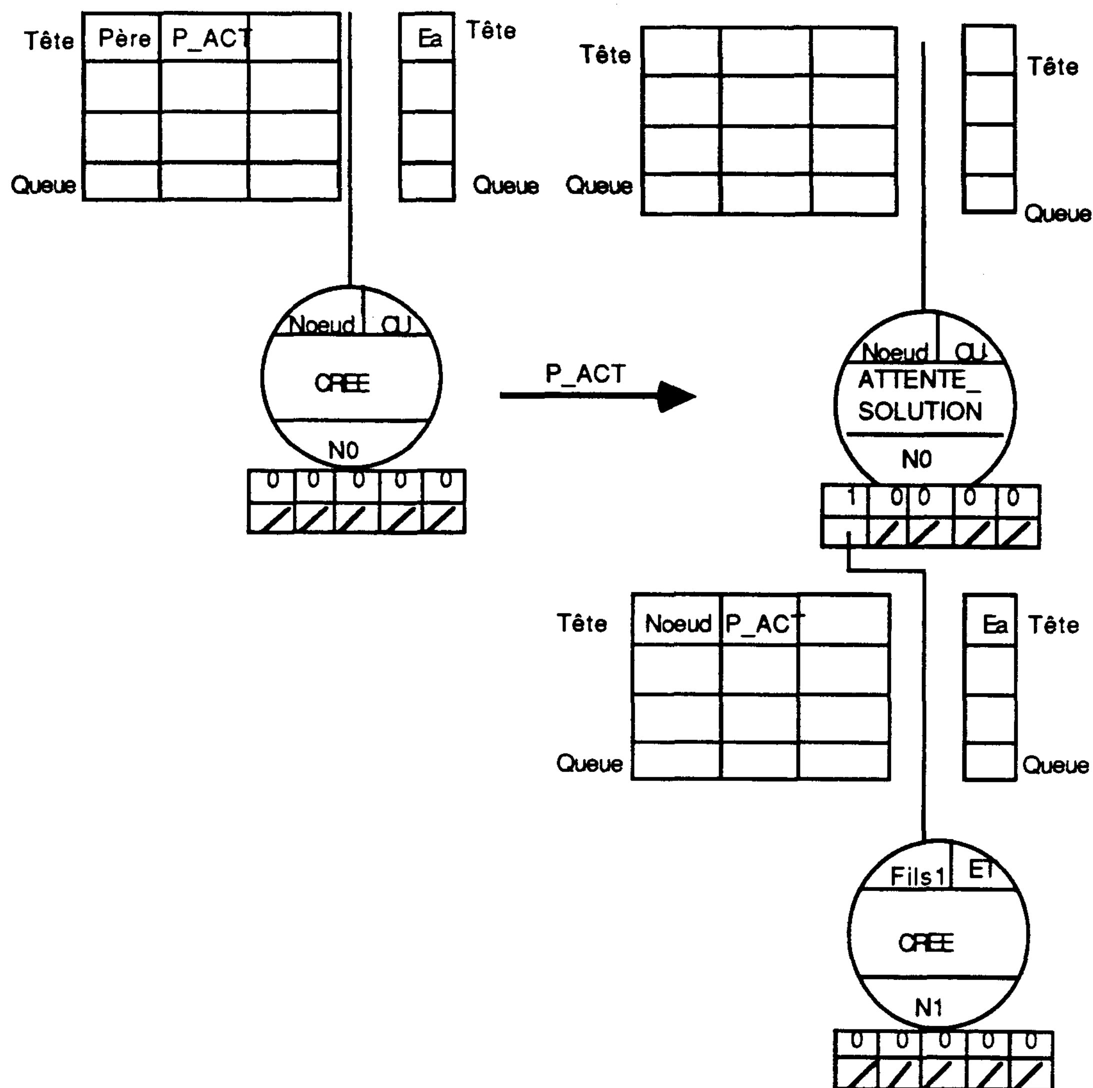


Figure 1 : Primitive P_ACT.

III. LA PRIMITIVE P_LOCK(Nœud_père, Nœud, Arguments).

Le nœud père exécute la primitive P_LOCK (Nœud_père, Nœud, Arguments) : il crée le nœud fils. L'état du nœud est initialisé à Créé, le bit d'existence est mis à un. Le lien fils du nœud père et le lien père du nœud fils sont associés. La primitive P_LOCK est déposée en queue du buffer de primitives, et l'environnement d'activation est déposé dans le buffer d'environnements. Le gérant du nœud prélève la primitive dans le buffer de primitives, puis construit une primitive P_ACT(Nœud, Fils1, E) qui sera appliquée au premier nœud fils créé Filsl. Le nœud attend alors la première des solutions évaluées par ce nœud fils, ainsi qu'une demande de solutions émanant du nœud père: il est en Attente_Double (figure 2)

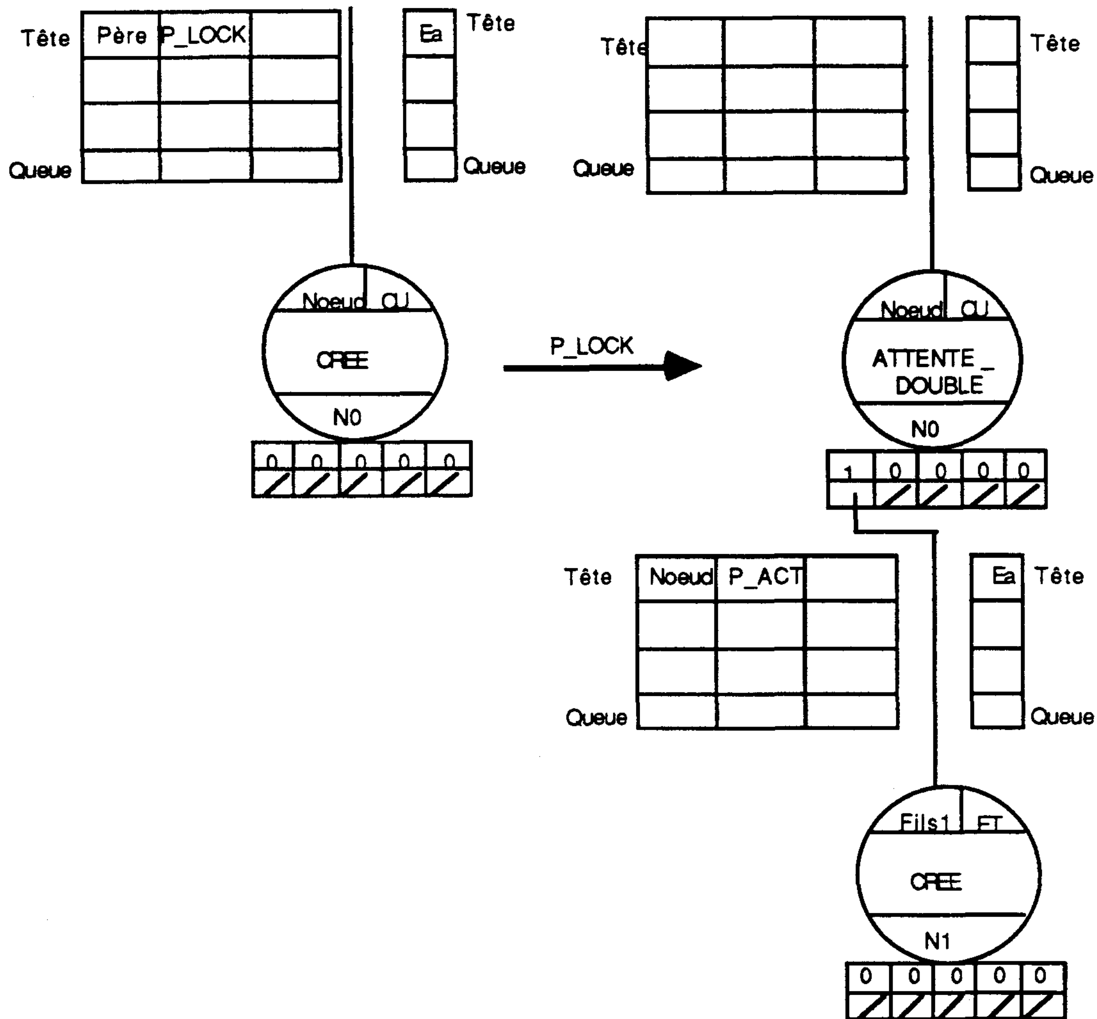


Figure 2 : Primitive P_LOCK.

Remarque : Les primitives d'activation en mode restreint ou non, sont les premières primitives qu'interprète un nœud. Etant les premières primitives reçues par le nœud, elles sont déposées en tête du buffer de primitives, qui était vide jusqu'alors.

IV. LA PRIMITIVE P_UNLOCK(Nœud).

Cette primitive concerne des nœuds existants. Elle traduit la demande explicite d'une solution, demande émanant du nœud père. L'état d'un nœud à qui est appliquée cette primitive est l'un des suivants :

- Attente_Double : le nœud attend conjointement une demande de solution et une solution construite par l'un de ses nœuds fils.
- Attente_Levée_Verrou : le nœud a mémorisé une solution évaluée par l'un de ses fils.

Examinons chacune de ces possibilités :

IV.1. L'état du nœud est Attente_Double.

Le nœud attend simultanément une demande de solution et une solution : seul l'état du nœud est modifié. En effet, le nœud reste en attente de solution évaluée par l'un de ses fils, solution nécessaire à la satisfaction de la demande de solution émanant du nœud père (figure 3.1).

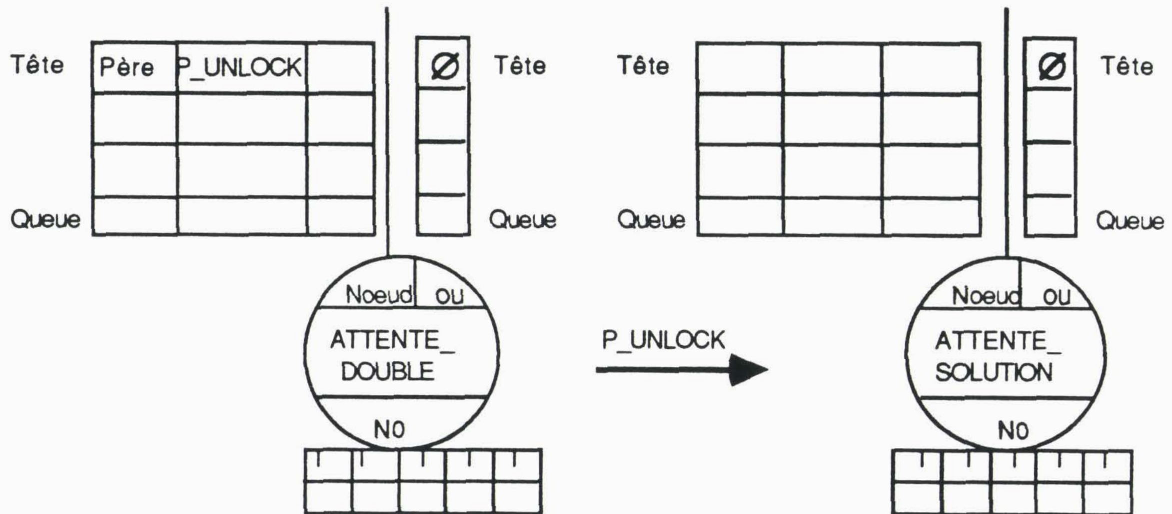


Figure 3.1

IV. 2. L' état du nœud est Attente_Solution.

IV.2.a. Le buffer d'environnements du nœud contient un environnement-échec.

L'échec évalué par le nœud est un échec définitif. En effet, avant de prendre en considération cet échec, le nœud s'est assuré qu' aucun autre de ses nœuds fils n'est en mesure de lui fournir une solution autre qu' échec. La table des liens fils est telle que chaque bit d'existence est à zéro, et chaque lien fils inexistant. Le nœud passe en état résolu. De plus, il construit deux primitives, qui sont en l'occurrence: une primitive d'échec, P_ECHEC(Nœud_Père, Nœud) qu'il place dans le buffer de primitives de son père, et une primitive de destruction qu'il dépose dans son propre buffer (Cf figure 3.2). L'environnement-échec est mis en queue du buffer d'environnements.

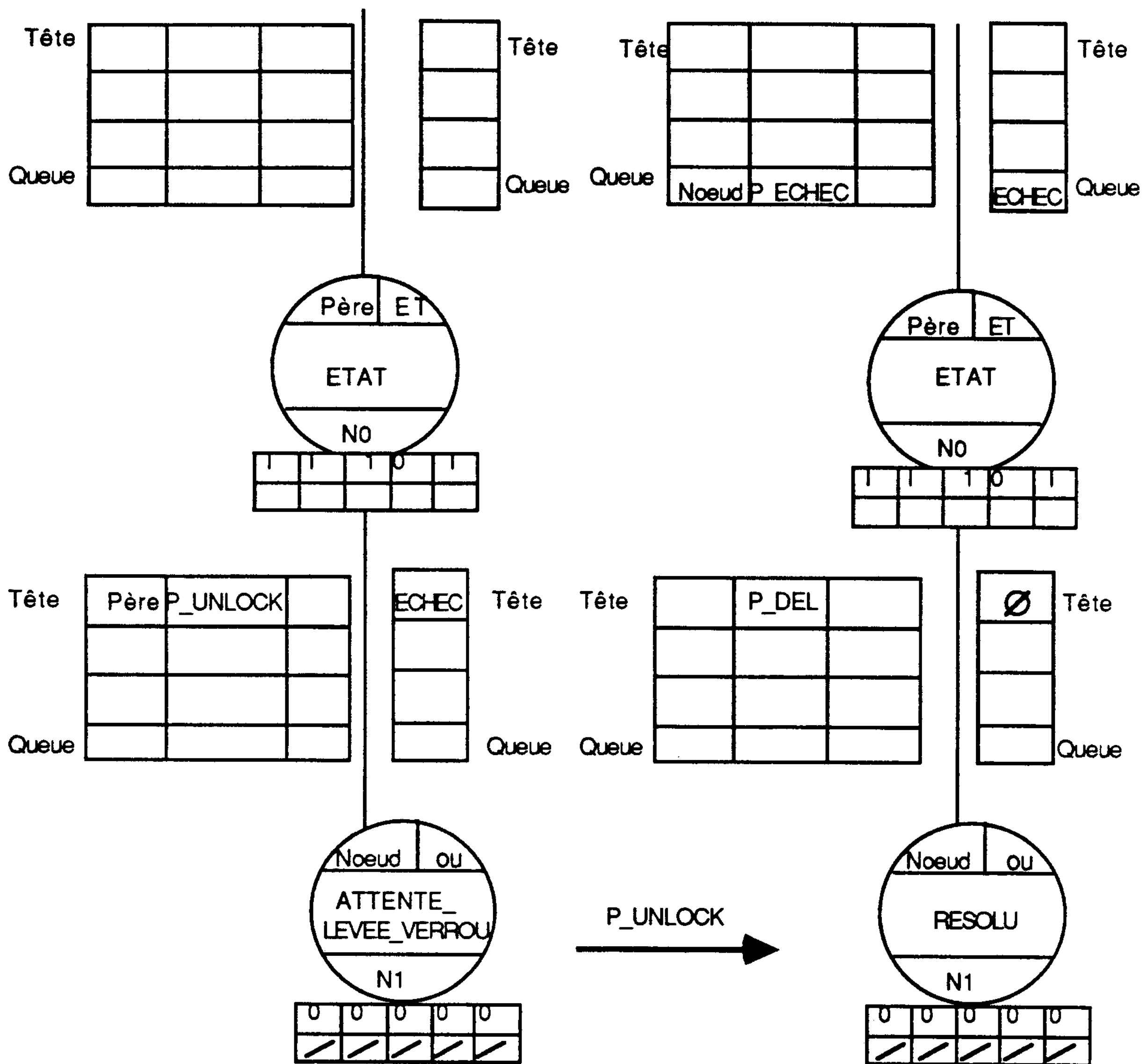


Figure 3.2 : le nœud est en situation d'échec.

IV.2.b.La solution mémorisée n'est pas un échec.

Cas 1. Toutefois, la table de liens est telle que chaque bit d'existence est à zéro, et chaque lien fils inexistant. Le sous-arbre dont le nœud est le père a donc été détruit, et le nœud a terminé son travail. En conséquence, son état devient résolu. Comme précédemment, il élabore deux primitives, dont l'une est appliquée au nœud père et l'autre à lui-même. La première, de type résultat, exprime le caractère définitif de la solution retournée au nœud père, la seconde entraîne la destruction du nœud (figure 3.3)

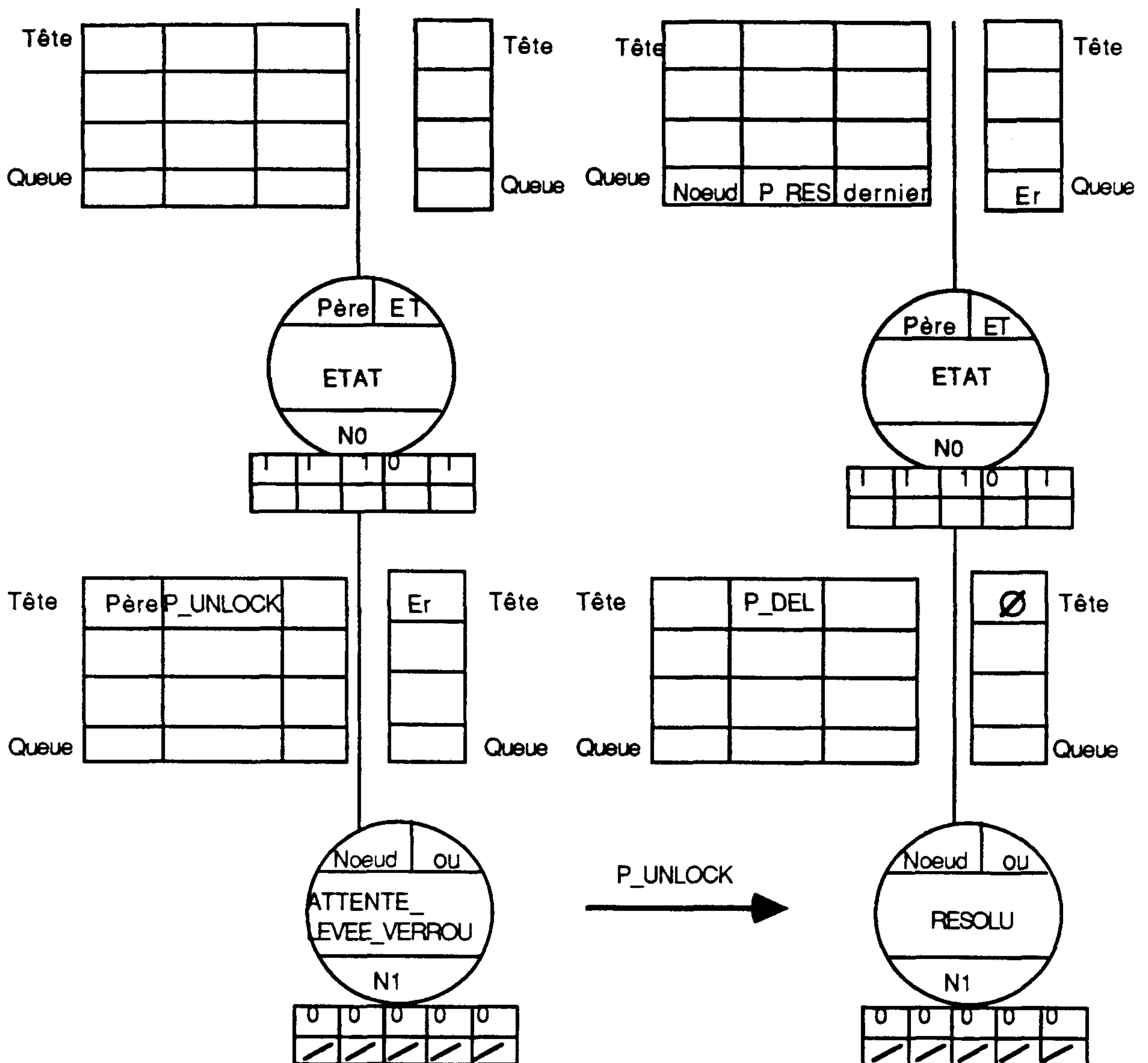


Figure 3.3 : le nœud a évalué une dernière solution.

Cas 2. L'un au moins des fils du nœud est encore actif (comme en témoigne la table des liens fils). Le nœud réalise alors les deux actions suivantes : il construit une primitive de type résultat à l'intention de son nœud père, ainsi qu'une primitive demande de solution à l'un de ses nœuds fils. L'indice du nœud fils concerné est le minimum des indices correspondant à un bit d'existence de valeur un.

Remarque : L'attribut premier est vrai si l'environnement solution retourné est le premier que contient le buffer d'environnements.

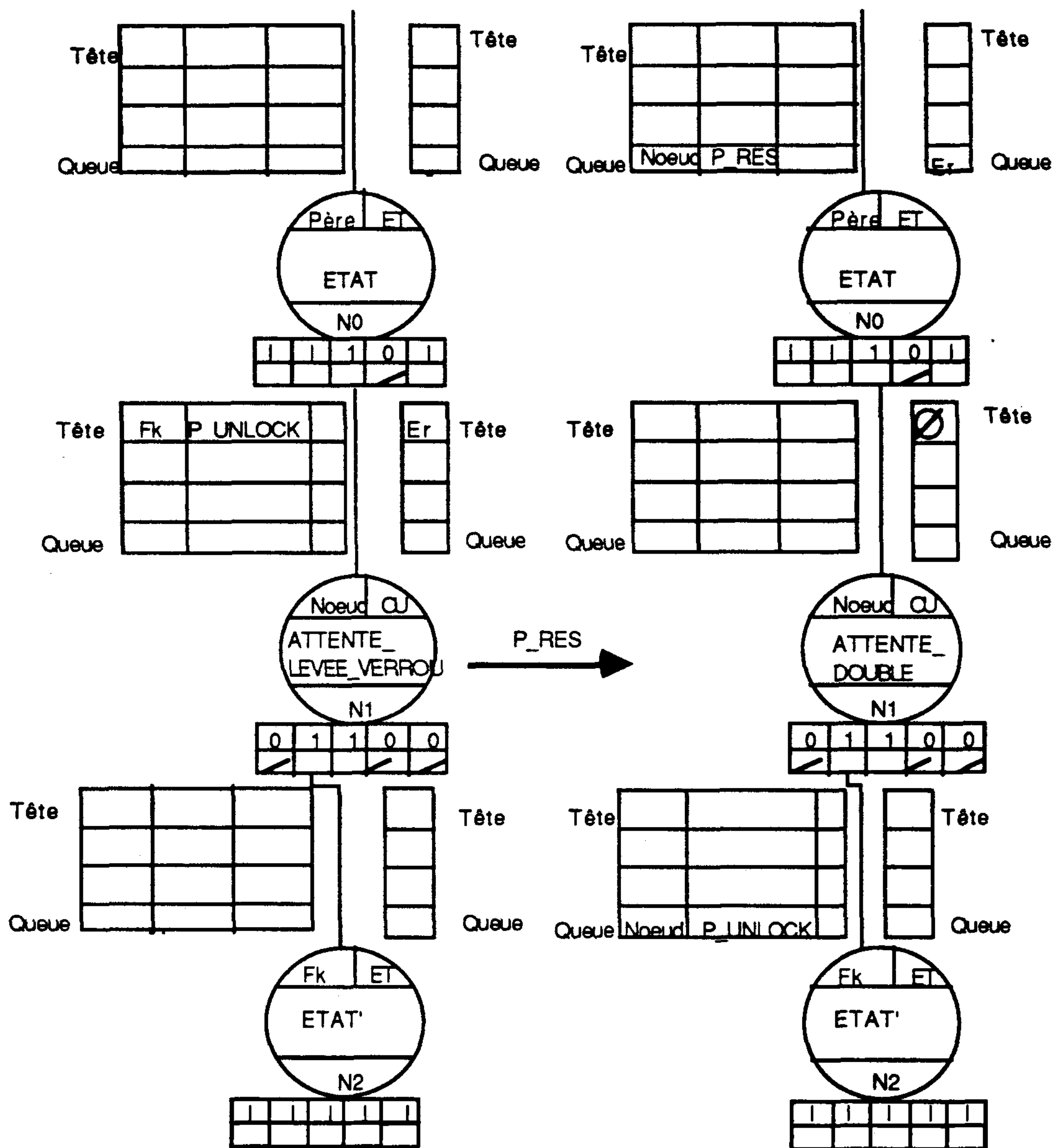


Figure 3.4 : un des nœuds fils est encore actif.

V. LA PRIMITIVE P_RES(F_k Nœud, Attributs, Arguments).

Cette primitive, qui concerne des nœuds existants, traduit le retour d'une solution évaluée par le nœud fils F_k, vers le Nœud. Le gérant du nœud interprète la primitive en fonction d'une part de l'état courant du nœud, et d'autre part de la valeur des attributs (premier et dernier) ainsi que de l'émetteur F_k de la primitive.

Un nœud qui reçoit une telle primitive en a préalablement fait la demande explicite : il a donc traité la solution reçue précédemment et son buffer d'environnements est vide. Il peut toutefois attendre outre une solution, une demande explicite de solutions émanant de son propre nœud père. En conséquence, l'état du nœud peut prendre les deux valeurs suivantes : Attente_Solution et Attente_Double. Examinons ces deux situations.

V.1. L'état du nœud est Attente_Solution.

Il peut alors émettre une primitive de type résultat vers son propre nœud père, primitive dont la valeur des attributs reste à calculer.

Remarque : L'attribut premier est vrai si la solution évaluée par le nœud est la première retournée au père. Les éventuels échecs intermédiaires ont été écartés. L'indice k est donc le plus petit indice tel que la première solution du nœud F_k diffère de l'échec.

V.1.a. La table de liens est vide.

L'attribut dernier est vrai si le nœud n'est plus en mesure d'obtenir une solution de l'un de ses nœuds fils. L'activité de ces derniers est donc nulle, comme en témoigne la table de liens du nœud. L'ensemble des bits d'existence est à zéro. Si tel est le cas, le nœud passe dans l'état résolu. Il peut alors être détruit : c'est pourquoi il dépose dans son buffer de primitives une primitive de destruction, soit P_DEL(figure 4.1).

V.1.b. Un fils du nœud est encore actif.

Dans le cas contraire (l'un au moins des nœuds fils est encore actif), le nœud cherche à obtenir une nouvelle solution de l'un de ses nœuds fils F_i (k<=i<=n). Il détermine alors dans un premier temps le nœud F_i qui lui retournera cette solution, puis le type de primitive à appliquer à ce nœud (ceci en fonction de la valeur des attributs premier et dernier de la primitive de type résultat reçue, et d'autre part l'existence ou de la non existence de ses nœuds fils).

Quatre cas sont à examiner : les deux attributs ont simultanément la même valeur ou ont au contraire deux valeurs distinctes. Etudions chacune de ces éventualités :

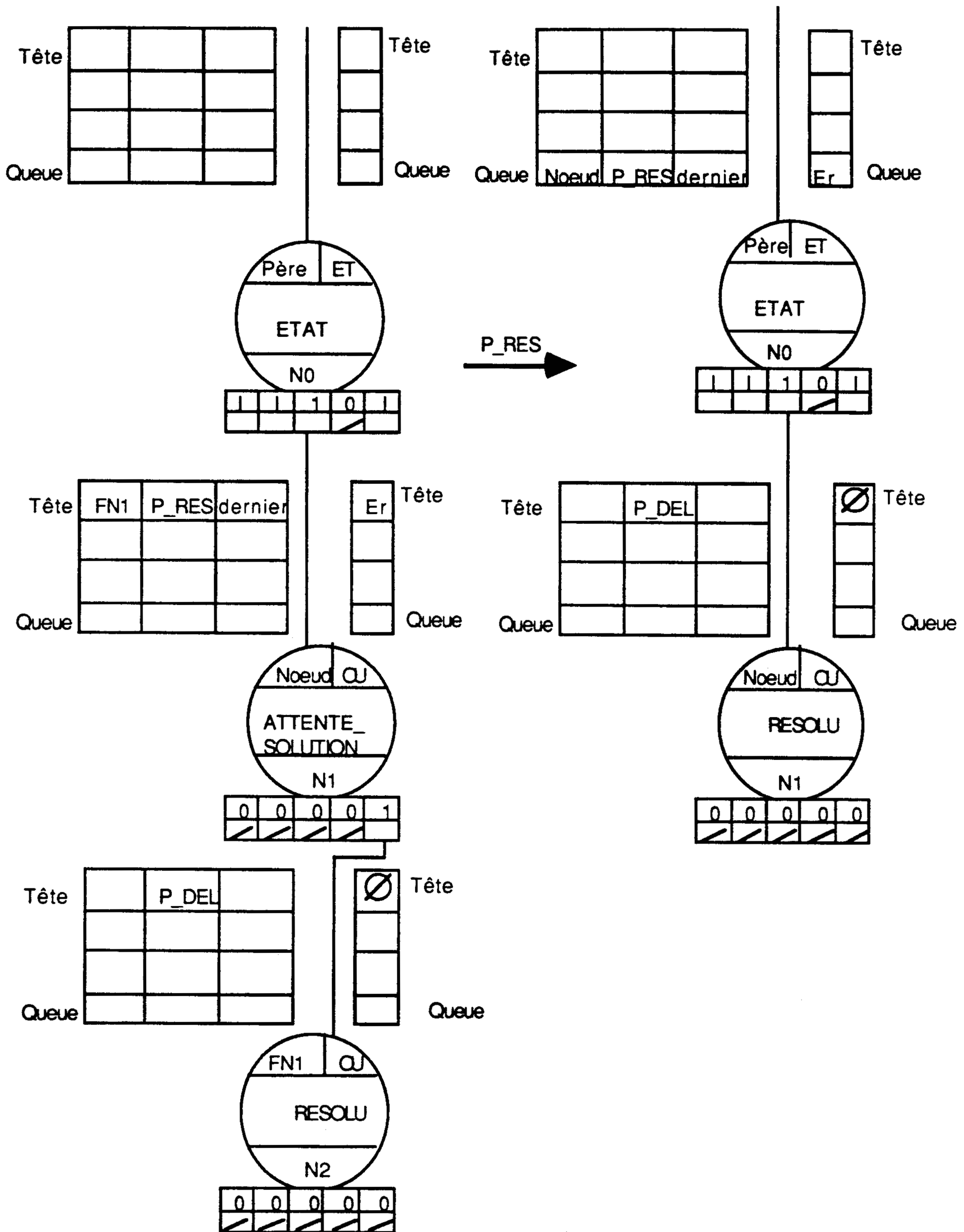


Figure 4.1

Cas 1. Aucun attribut n'est vrai. Le nœud F_k, qui a fourni plus d'une solution, est toujours actif. Le nœud émet donc une demande de

solutions, qu'il dépose dans le buffer de primitives de ce dernier (Cf figure 4.2).

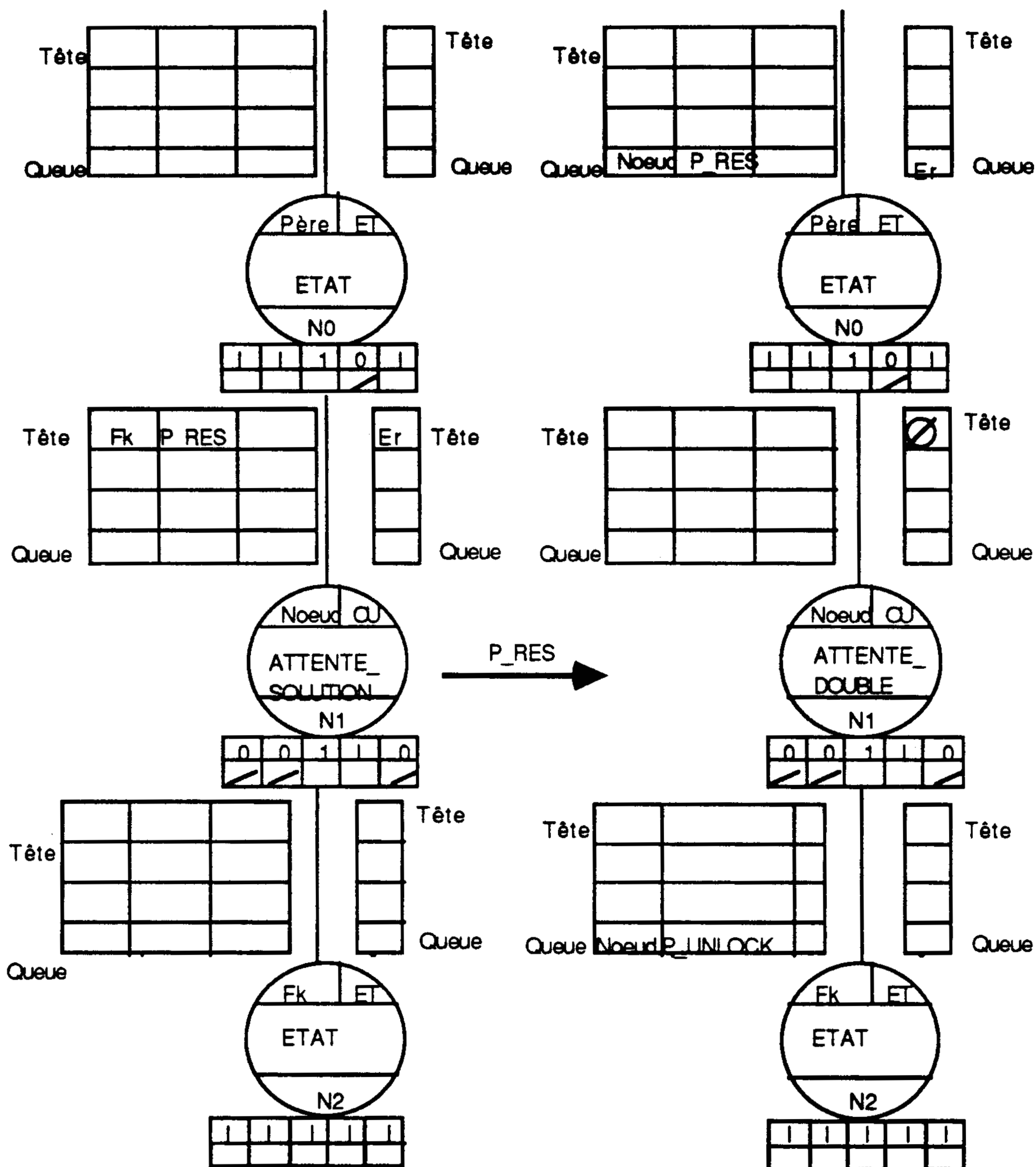


Figure 4.2

Cas 2. L'attribut premier est seul vrai . Le nœud F_k est donc encore actif, comme en témoigne le bit d'existence correspondant et élabore une autre solution. Le nœud dépose donc une demande de solutions dans le buffer de primitives du nœud F_k. Comme la solution reçue est la première calculée par le nœud F_k, le nœud active en mode restreint le nœud fils F_{k+1} s'il existe , c'est-à-dire lorsque k diffère de n (figure

4.3).

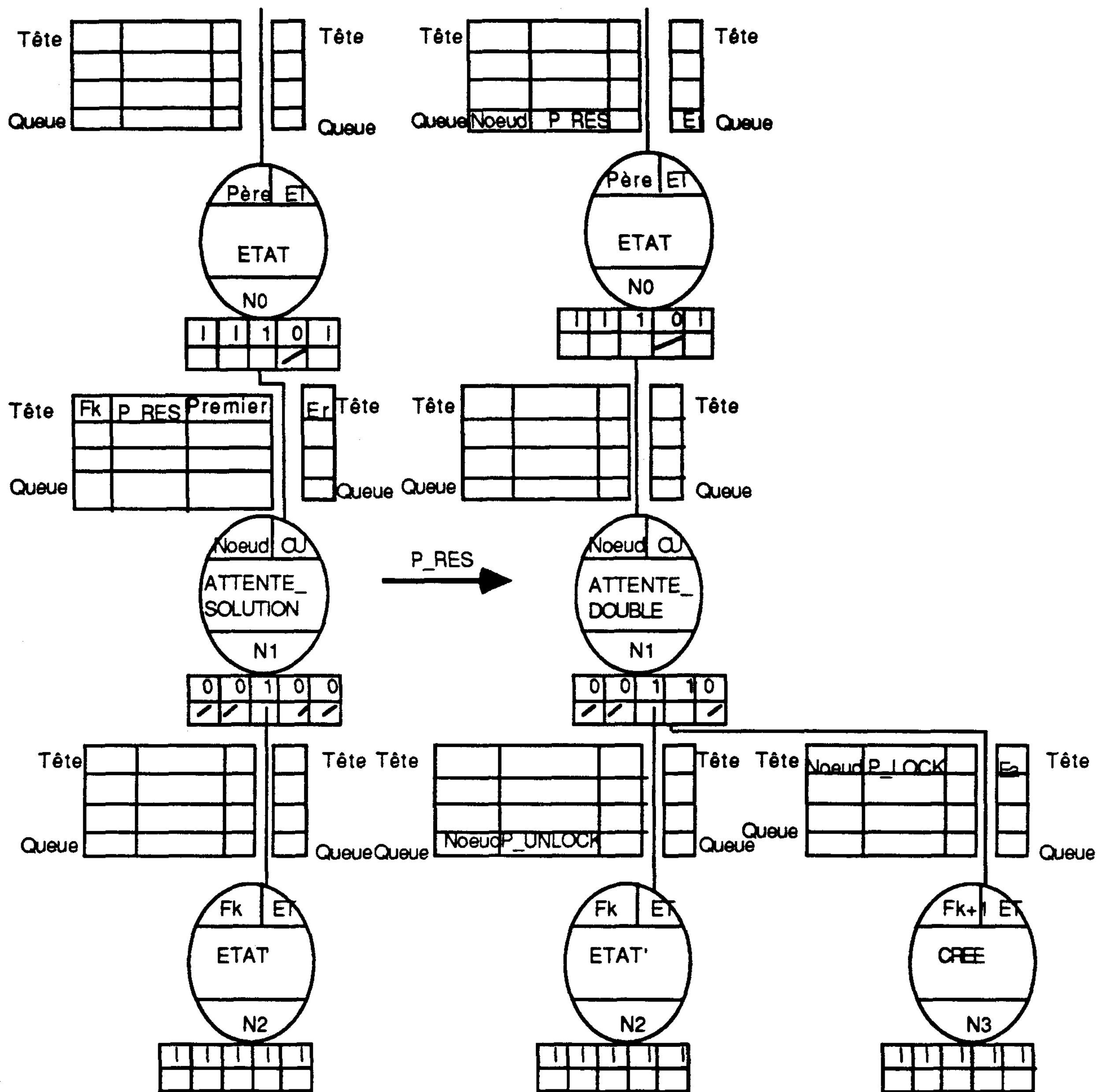


Figure 4.3

Cas 3 L'attribut dernier est vrai . Dans ce cas, le nœud F_k est devenu inactif. Aussi, le nœud met-il à jour sa table de liens. Le bit activé prend la valeur zéro, et le lien vers le nœud F_k est détruit.

Si l'attribut dernier est seul vrai, le nœud a préalablement retourné au moins une solution, entraînant ainsi, l'activation en mode restreint du nœud fils suivant éventuel F_{k+1} (pour k<n). Il convient donc de déposer dans le buffer de primitives du nœud F_{k+1} une demande de solutions (Cf figure 4.4).

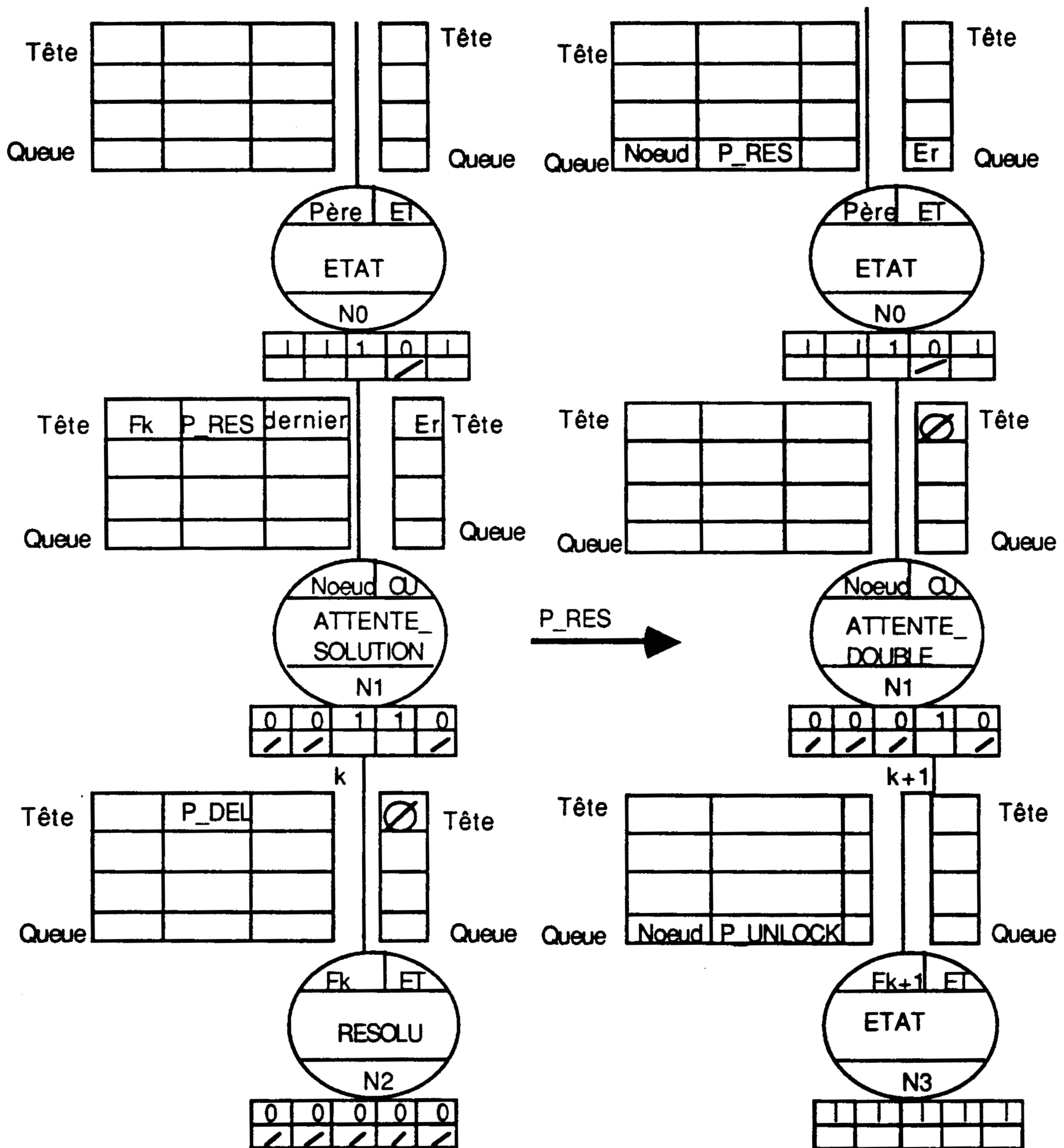


Figure 4.4

Cas 4. les deux attributs sont vrais : le nœud F_k a évalué une solution unique. Le nœud F_{k+1} (pour k<n) est donc toujours inactif. Aussi, le

nœud active-t-il le nœud F_{k+1} (Cf figure 4.5).

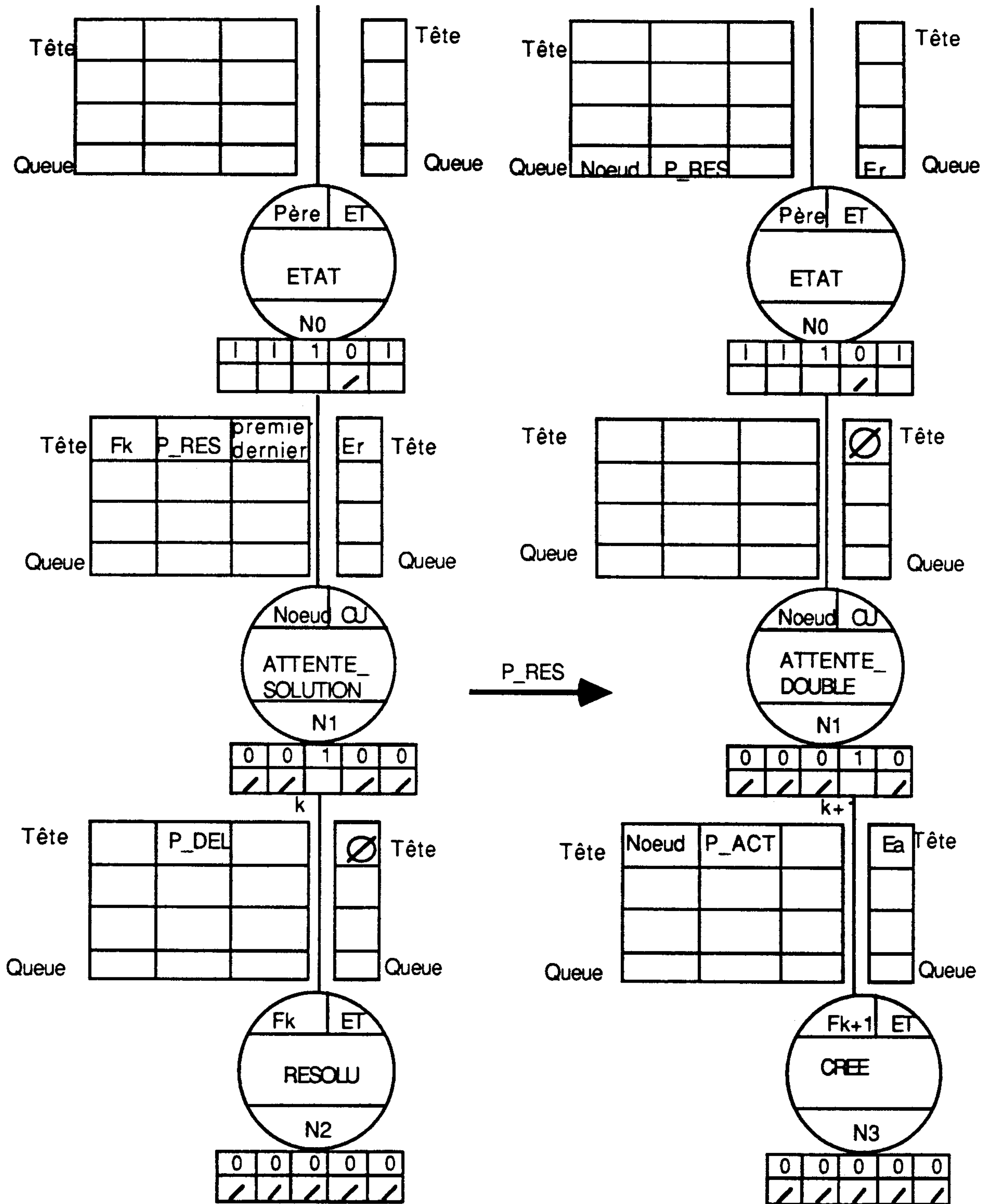


Figure 4.5

V.2.Le nœud est en Attente_Double.

Il ne peut donc retourner la solution reçue par le biais de la primitive P_RES, à son père. Aussi, l'environnement-solution est-il mémorisé dans le buffer d'environnements. Tant qu'il détient dans son buffer d'environnement cette solution, le nœud ne peut demander explicitement une nouvelle solution à l'un de ses fils. Tout au plus, le cas échéant, peut-il activer en mode restreint l'un de ses fils. Reprenons les situations précédemment évoquées et voyons en quoi elles sont modifiées :

V.2.a. L'activité de tous les nœuds fils est nulle.

L'ensemble des bits d'existence est à zéro. La solution détenue dans le buffer d'environnements est la dernière que le nœud peut évaluer : il passe en l'état Attente_Levée_Verrou (voir figure 4.6).

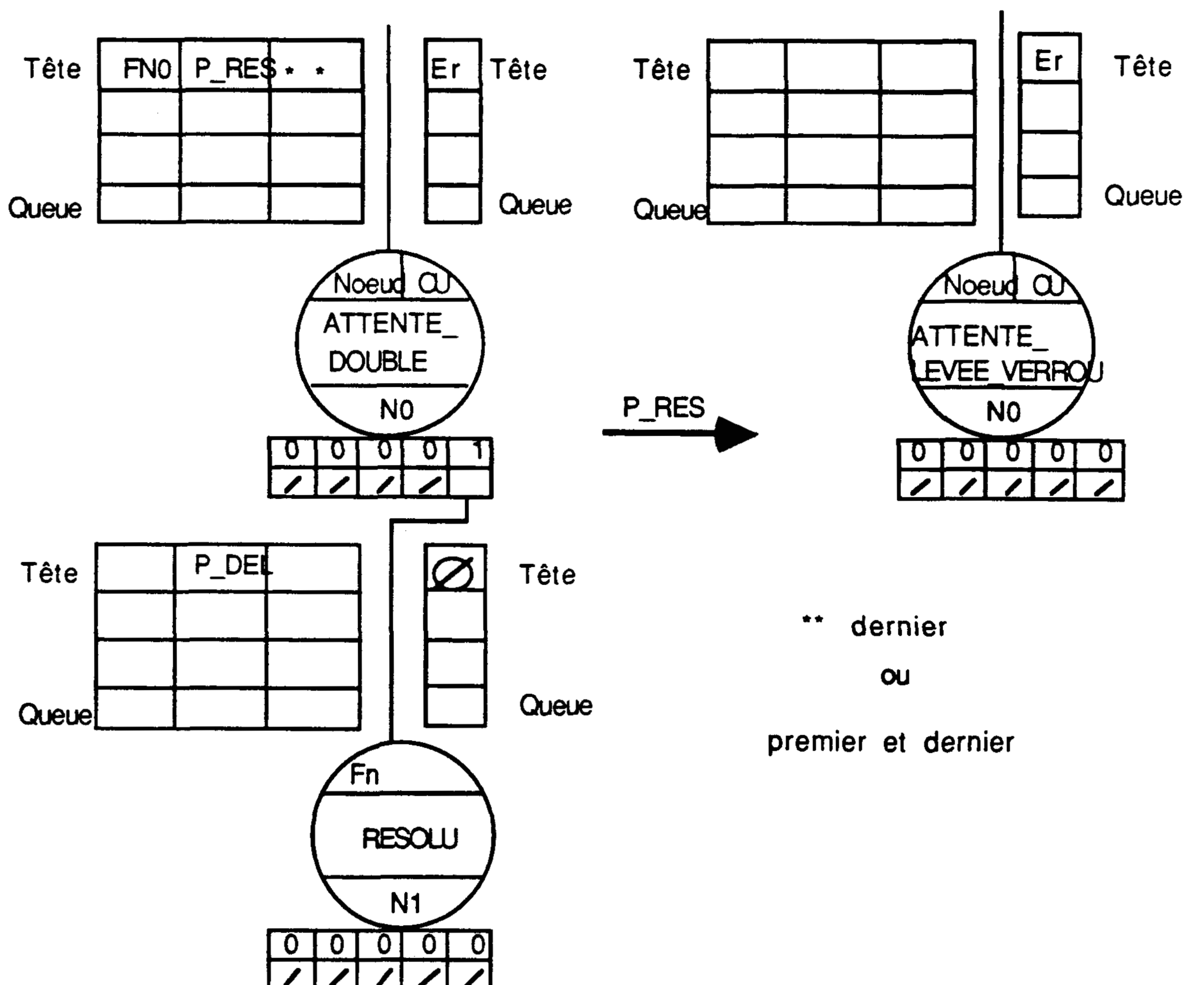


Figure 4.6

V.2.b. Un des fils du nœud est encore actif.

Cas 1. L'attribut premier P_RES est seul vrai. La solution obtenue est donc la première retournée par le nœud fils actif courant. Le dialogue est momentanément suspendu entre le nœud fils actif courant et le nœud. Toutefois, le nœud active en mode restreint le nœud fils suivant éventuel (Cf figure 4.7)

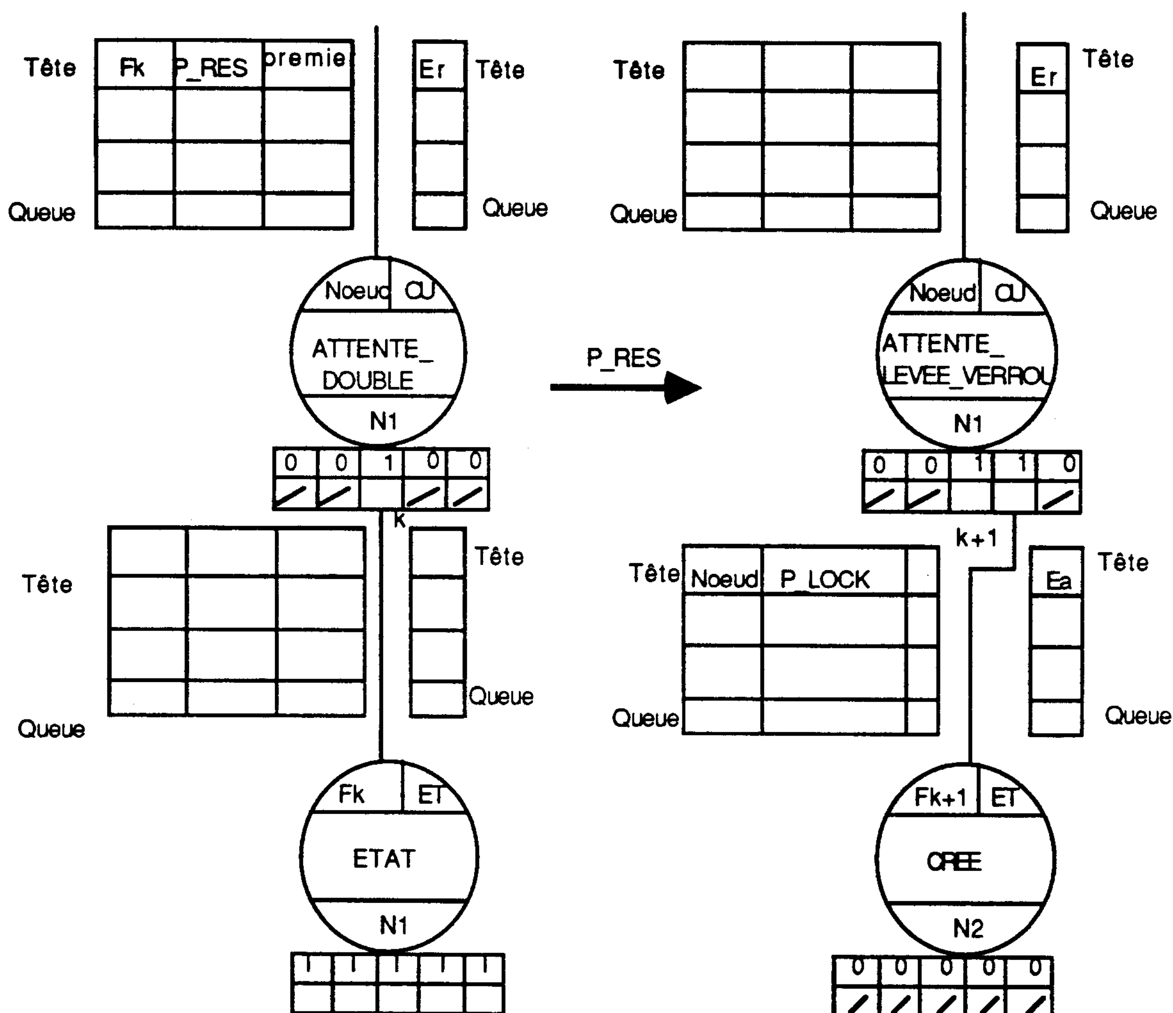


Figure 4.7

Cas 2. Les deux attributs, en l'occurrence premier et dernier sont vrais. Le nœud F_k a donc évalué une solution, par ailleurs unique. Le nœud met à jour sa table de liens en conséquence : le bit d'existence, d'indice k est mis à zéro, le lien fils détruit. Le nœud active en mode restreint le nœud fils F_{k+1} , s'il existe (figure 4.8).

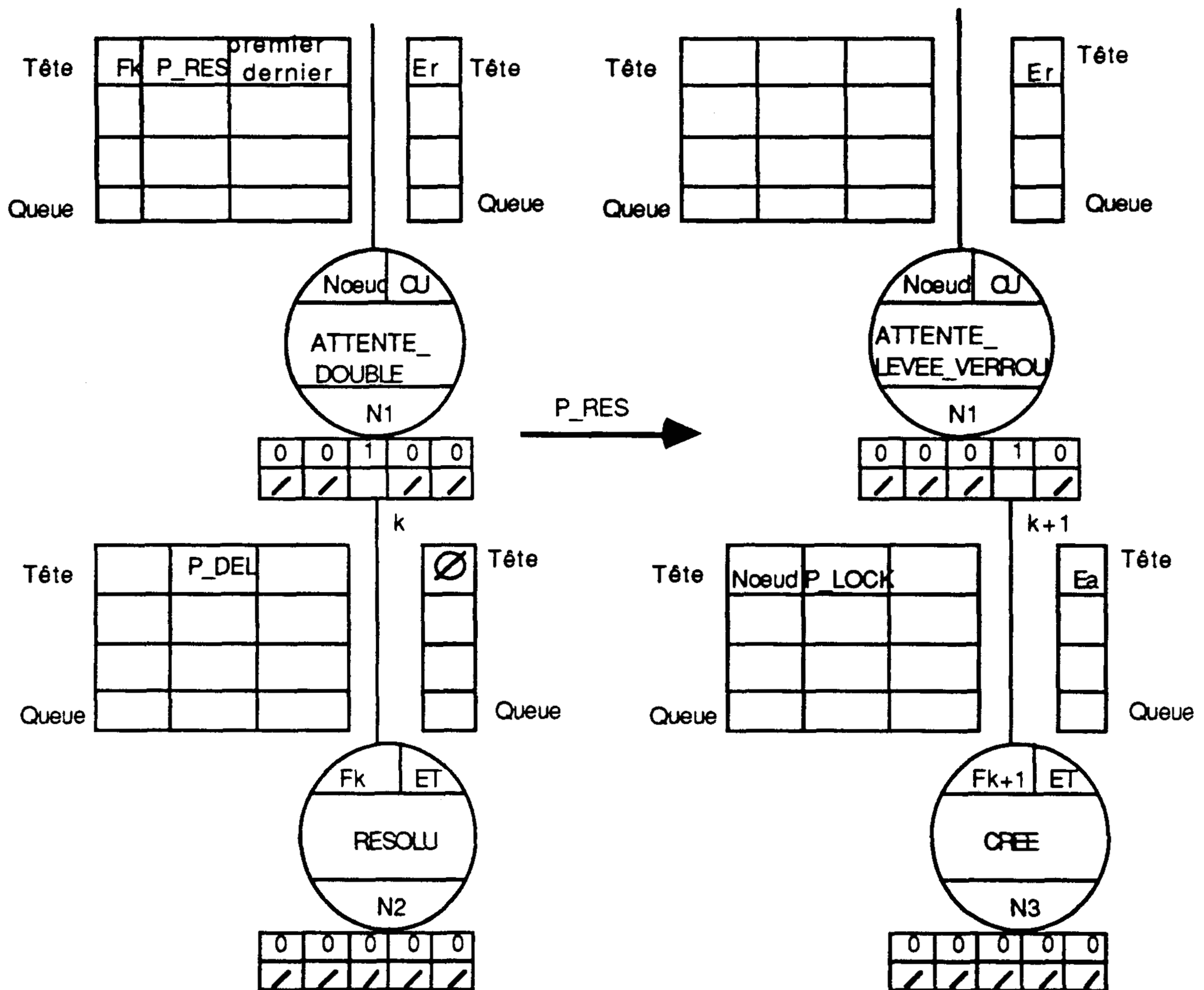


Figure 4.8

Cas 3. L'attribut dernier est seul vrai. Le nœud fils F_k a donc préalablement retourné au moins une solution. Le nœud n'effectue aucune action. Seul son état change : d'Attente_Double, il passe à Attente_Levée_Verrou (figure 4.9).

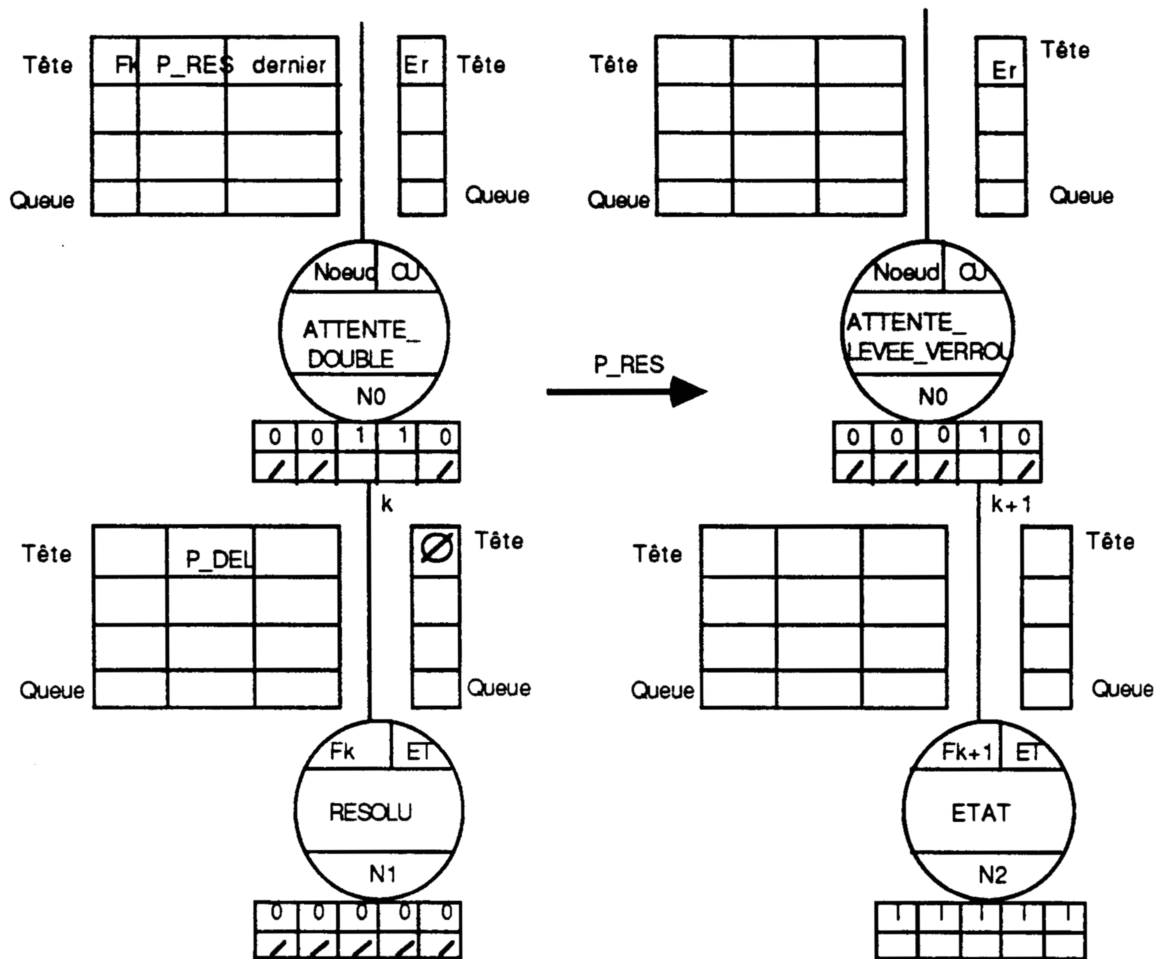


Figure 4.9

Cas 4. Aucun des attributs n'est vrai. Le nœud fils F_k est toujours actif. Toutefois, le dialogue est momentanément suspendu entre le nœud et le nœud fils F_k. Il reprend au moment où le buffer d'environnements du nœud est vidé. L'état du nœud est seul modifié : sa valeur devient Attente_Levée_Verrou (figure 4.10).

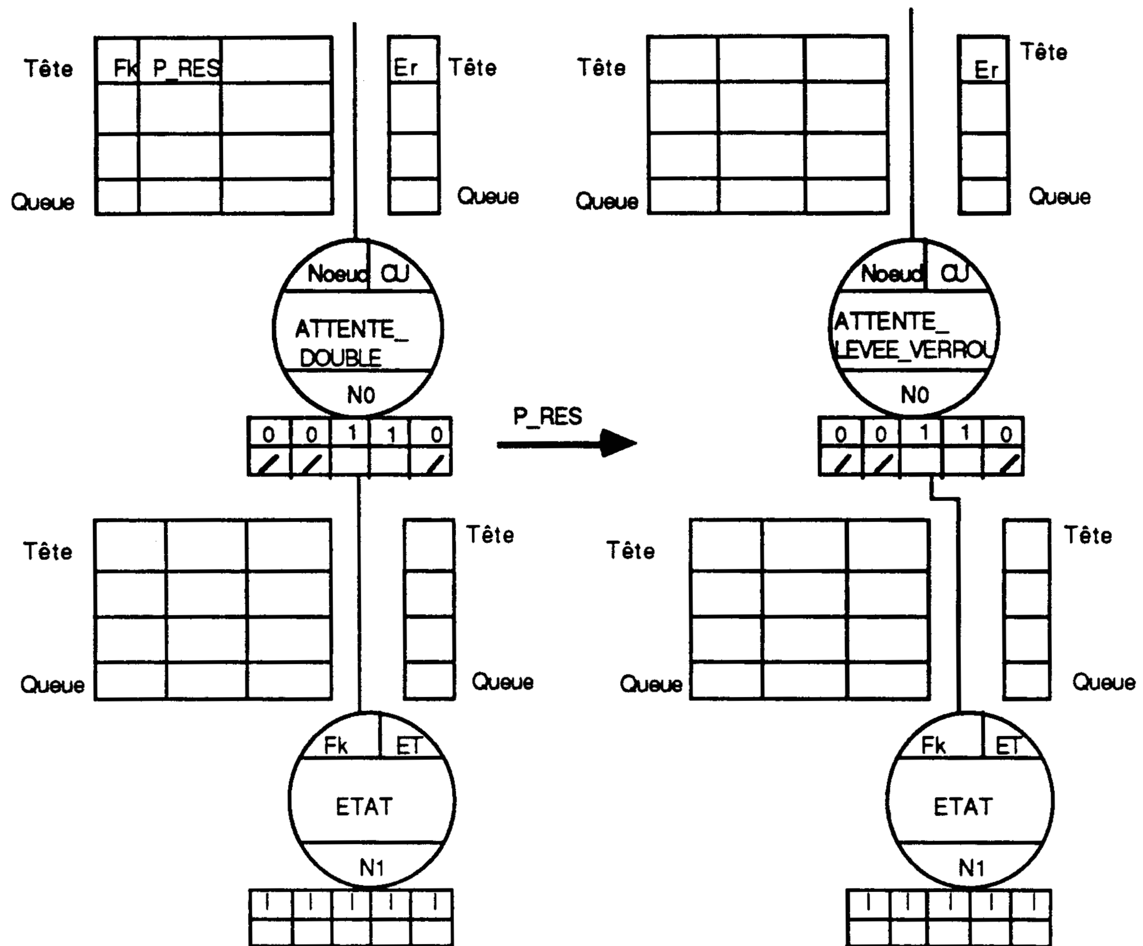


Figure 4.10

VI. LA PRIMITIVE P_ECHEC(F_k,Nœud)

Le gérant du nœud auquel la primitive est appliquée, interprète cette primitive en fonction d'une part de l'émetteur de la primitive(le nœud F_k), et d'autre part de l'état du nœud.

Un échec n'est transmis que lorsqu'il est définitif. Cette constatation amène les deux conséquences suivantes :

-*Le nœud F_k qui a évalué l'échec ne peut transmettre d'autre solution*. Le nœud réactualise alors sa table de liens. Le bit d'existence, d'indice k est mis à zéro, le lien d'indice k est détruit.

- *Le nœud considère donc l'échec obtenu comme une solution "intermédiaire"*. Le nœud tente alors d'obtenir une solution d'un autre de ses fils. Si aucun fils ne peut lui transmettre d'autre solution, l'échec est définitif. A l'inverse, l'échec est ignoré. L'examen de la table de liens permet de répondre à cette question.

VI.1.Le nœud fils responsable est le dernier fils du nœud.

L'échec est alors définitif. En effet, tous les autres fils ont déjà été activés et détruits. De l'état courant du nœud dépend la transmission de l'échec.

VI.1.a. Le nœud est en Attente Solution.

Si le nœud n'attend qu'une solution (son état est Attente_Solution), le nœud construit une primitive d'échec, P_ECHEC(Nœud, Nœud_père) qu'il dépose dans le buffer de primitives de son nœud père (figure 5.1), ainsi qu'une primitive de destruction qu'il place dans son propre buffer.

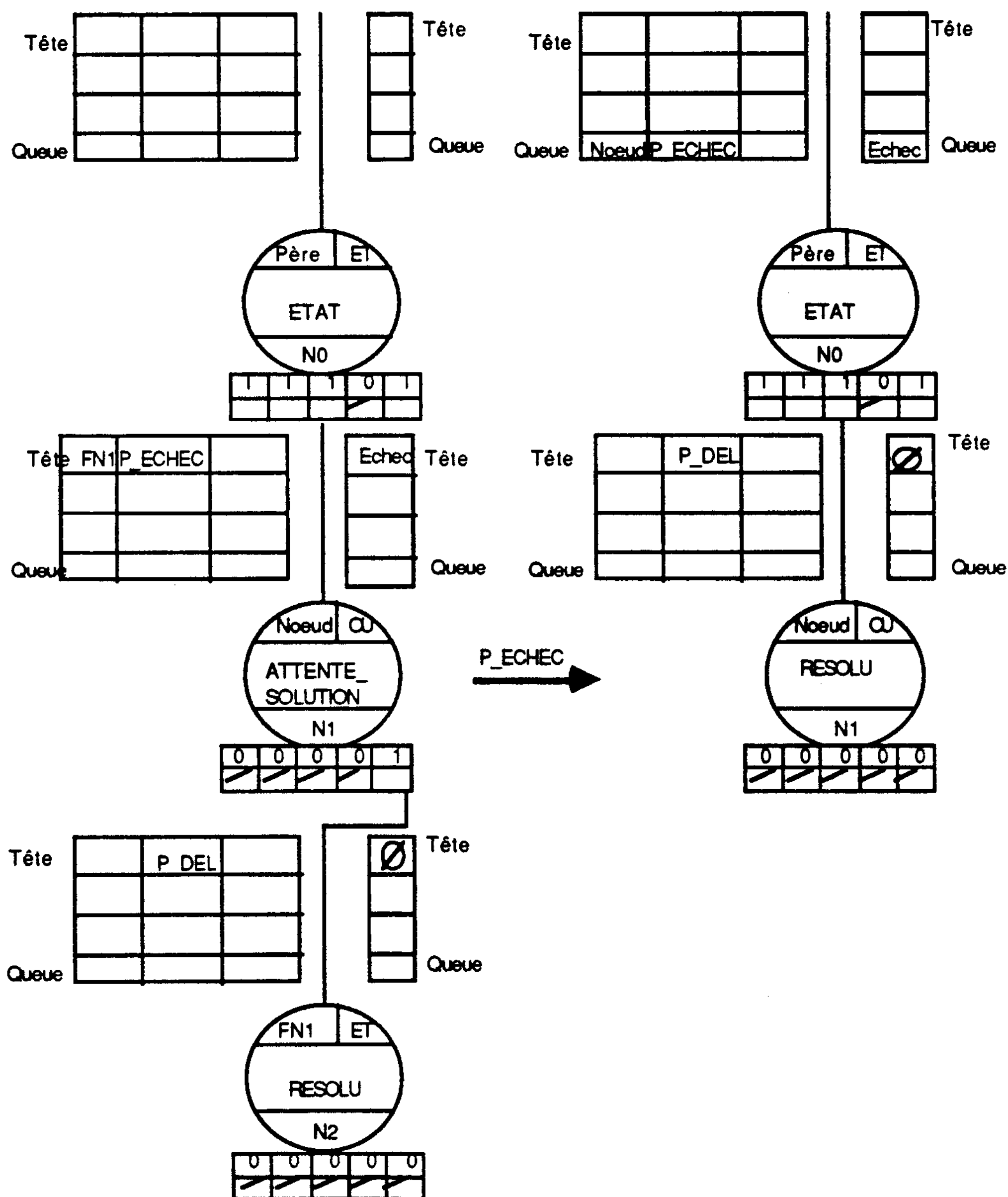


Figure 5.1

VI.1.b. Le nœud est en Attente Double.

Si le nœud attend conjointement une solution et une demande de solutions, le nœud mémorise l'échec dans son buffer d'environnements : l'état du nœud devient alors en Attente_Levée_Verrou (figure 5.2).

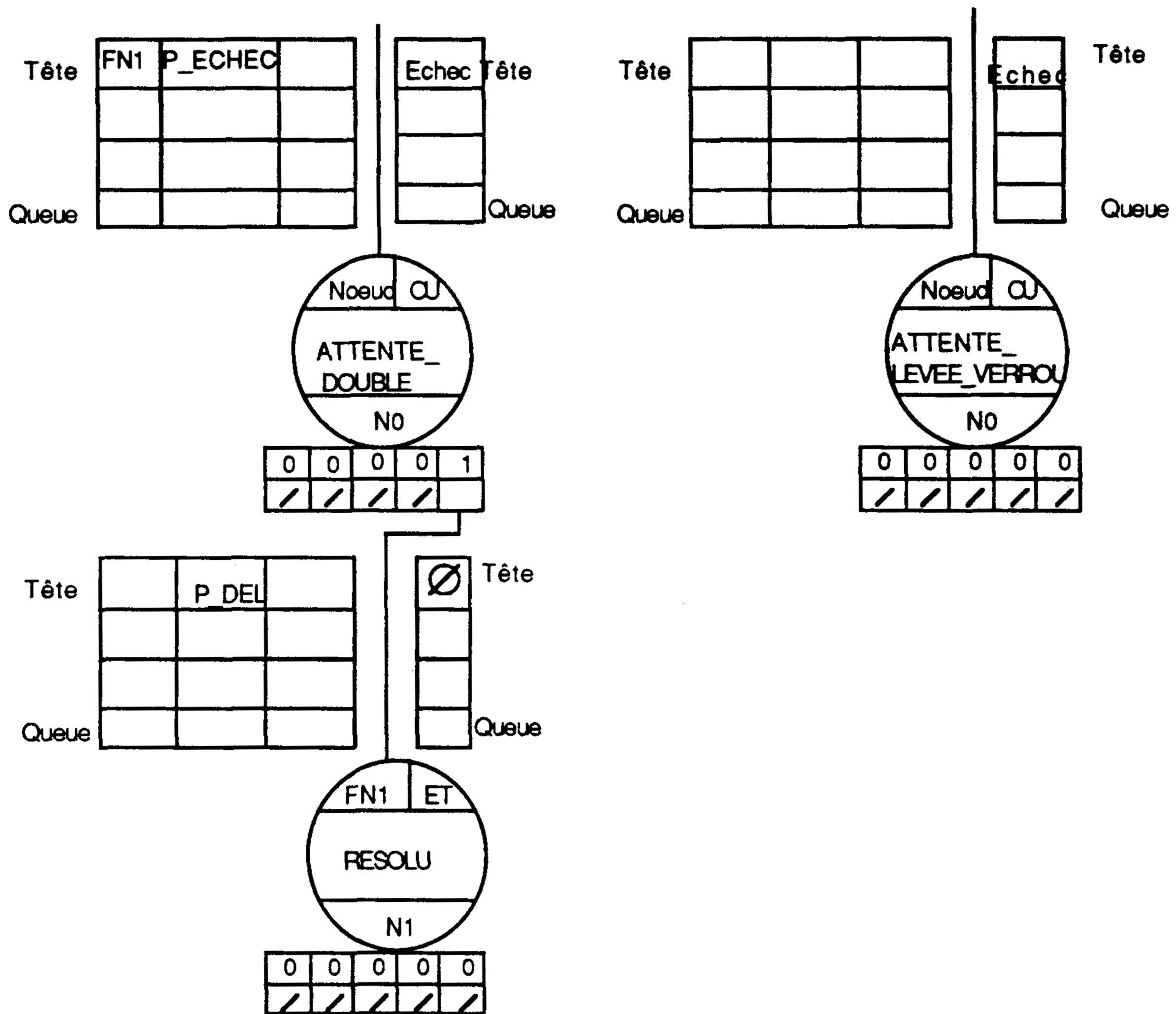


Figure 5.2

VI.2. Un nœud fils est susceptible de transmettre une autre solution.

Le nœud teste alors l'activité du nœud fils F_{k+1}.

Cas 1. Le bit d'existence d'indice k+1 est à zéro. Le nœud fils F_{k+1} est encore inactif : il est donc activé par le nœud (figure 5.3).

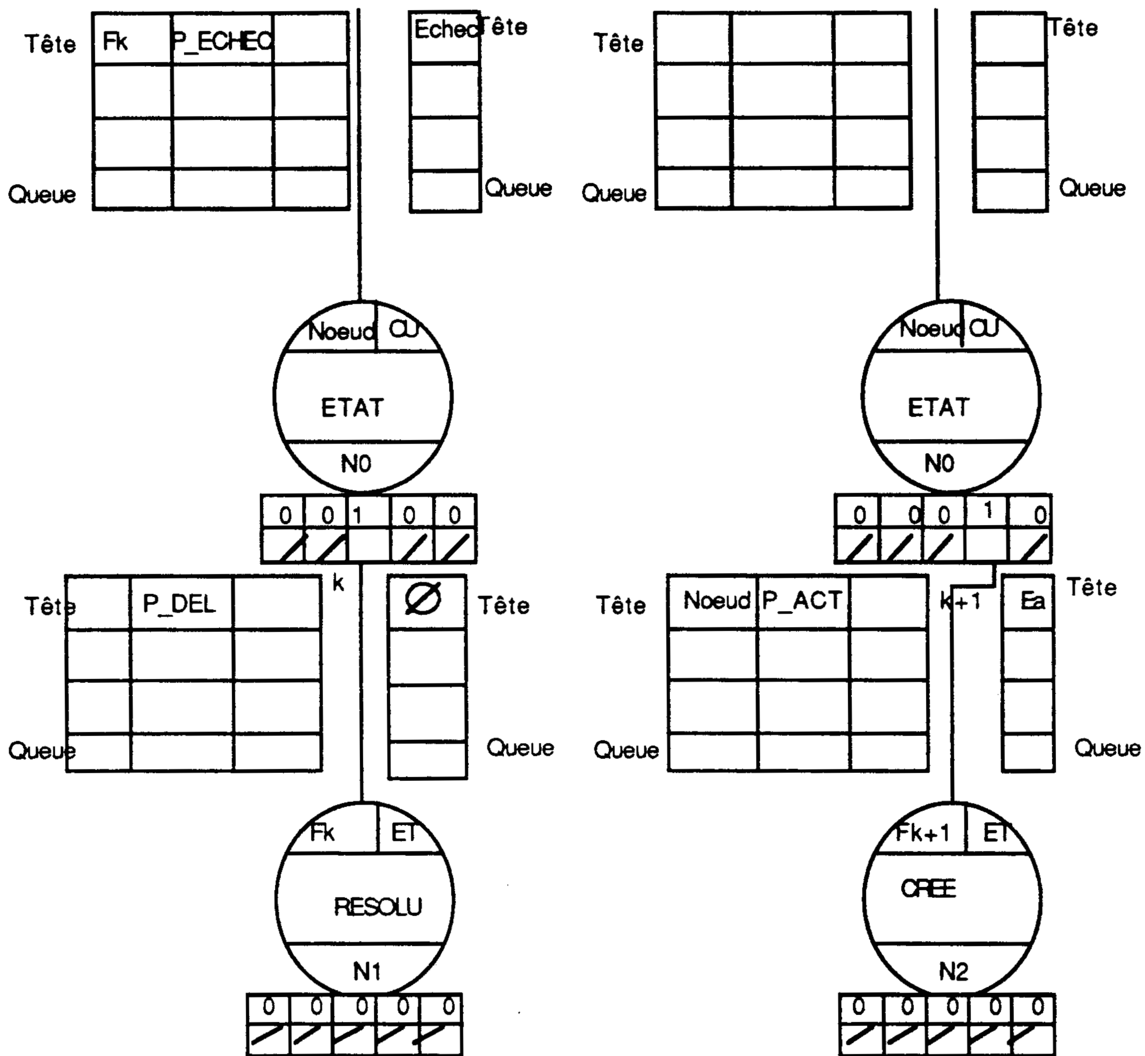


Figure 5.3

Cas 2. Le bit d'existence d'indice k+1 est à un. Le nœud F_{k+1} a été préalablement activé en mode restreint. Aussi, une demande de solution suffit (figure 5.4). Il est à noter que l'état du nœud est alors indifférent: cet état est soit Attente_Solution, soit Attente_Double. Le nœud père n'est en effet pas concerné par cet échec.

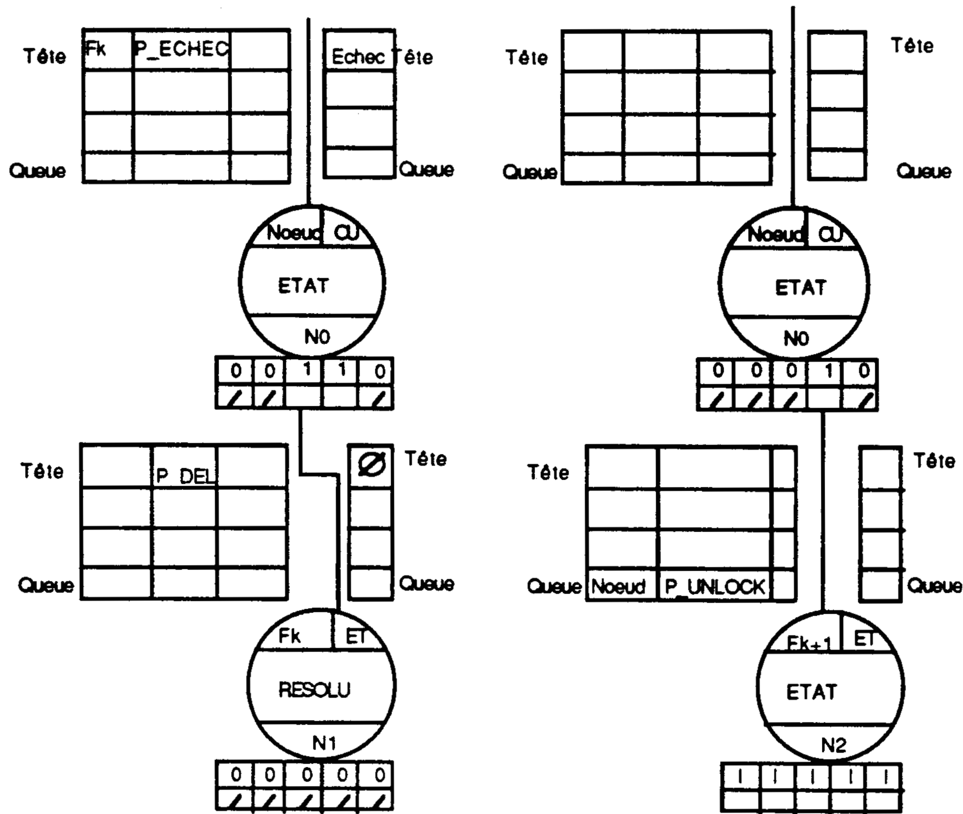


Figure 5.4.

VII. LA PRIMITIVE P_DEL.

Elle est créée par le nœud, dont l'état est alors résolu. Elle entraîne la destruction du nœud.

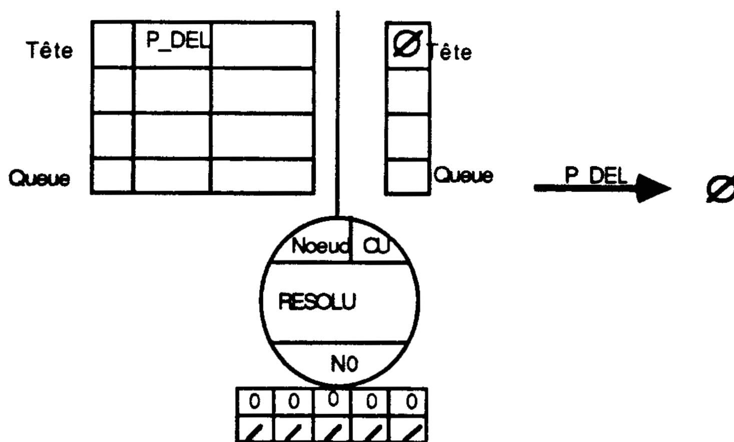


Figure 6

VIII. CONCLUSION.

Le formalisme introduit dans ce chapitre permet de dénombrer les informations nécessaires dans le cadre d'une implantation de ce modèle. Ces informations sont, non seulement relatives aux primitives, aux états des nœuds mais encore au nombre de primitives à tout instant dans le buffer de primitives d'un nœud.

En effet, à tout instant t, un nœud OU dialogue avec au plus deux nœuds, qui sont d'une part son père, d'autre part un de ses fils. Les autres de ses fils sont soit inexistants, soit activés en mode restreint (leurs verrous de solutions sont posés et empêchent toute remontée de solution vers le nœud).

On remarque que dès lors que deux nœuds pères et fils sont activés, et le nœud fils retourne plus d'une solution, ces deux nœuds travaillent de façon asynchrone. Le point de synchronisation est la demande de solution qu'émet le père, afin d'obtenir une nouvelle solution. Cet asynchronisme explique notamment l'état Attente_Double du nœud OU, état dans lequel le nœud peut recevoir indifféremment une primitive de demande de solution et une primitive résultats.

LE GERANT D'UN NŒUD ET

LE GERANT D'UN NŒUD ET

I. INTRODUCTION.

Le présent chapitre est consacré à l'étude du gérant d'un nœud ET. La démarche adoptée est similaire à celle mise en œuvre dans le chapitre précédente: les interprétations des primitives du modèle, en l'occurrence les primitives d'activation, de demande de solution, de résultat et de destruction, seront successivement présentées. Les conditions initiales, les émissions de primitives, les changements d'états du nœud seront précisés.

II. LA PRIMITIVE P_ACT(Nœud_père, Nœud, Arguments).

Le nœud père exécute la primitive P_ACT(Nœud_père, Nœud, Arguments). Il crée le nœud dont l'état initial aura la valeur Créé. Le lien fils du nœud père et le lien père du nœud sont associés. Le bit d'existence correspondant est mis à un. La primitive P_ACT est déposée dans le buffer de primitives du nœud.

La phase d'unification entre l'environnement transmis par la primitive et celui détenu par le nœud commence. Le résultat de cette unification influencera le comportement du nœud

II.1. L'unification échoue.

Le nœud construit une primitive d'échec qu'il dépose ensuite dans le buffer de primitives de son nœud père, ainsi qu'une primitive de destruction qu'il dépose dans son propre buffer de primitives. L'état du nœud devient alors résolu (cf figure 1.1).

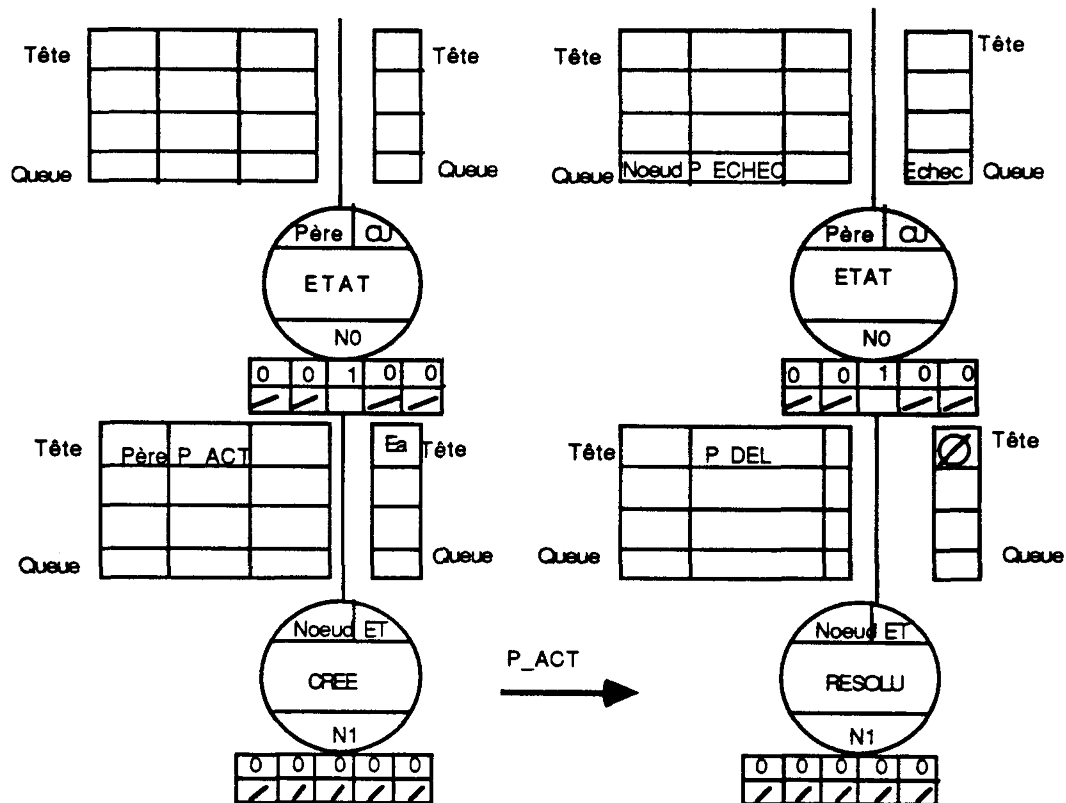


Figure 1.1

II.2. L'unification est un succès et le nœud n'a pas de fils (N1=0).

Le nœud a donc calculé une solution, solution unique par ailleurs. Il crée donc une primitive de type résultat, dont les attributs (premier et dernier) sont tous deux positionnés. Le nœud dépose cette primitive dans le buffer de primitives du nœud père, ainsi qu'une primitive de destruction dans son propre buffer de primitives (cf figure 1.2). Il passe alors à l'état résolu.

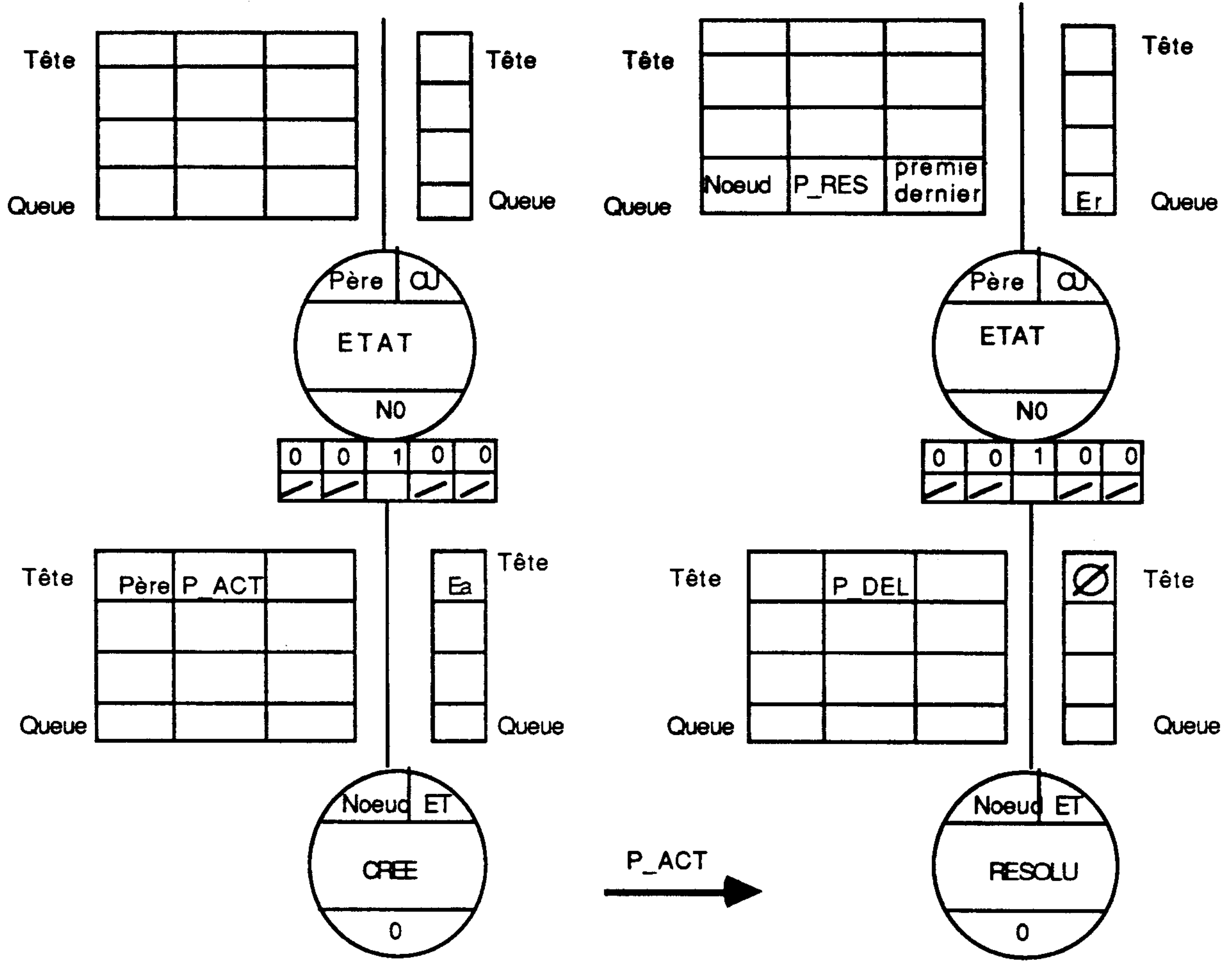


Figure 1.2

II.3. L'unification est un succès et le nœud possède au moins un fils.

Le nœud active alors son premier fils, dont il attend un environnement solution (cf figure 1.3). Son état devient Attente_Solution.

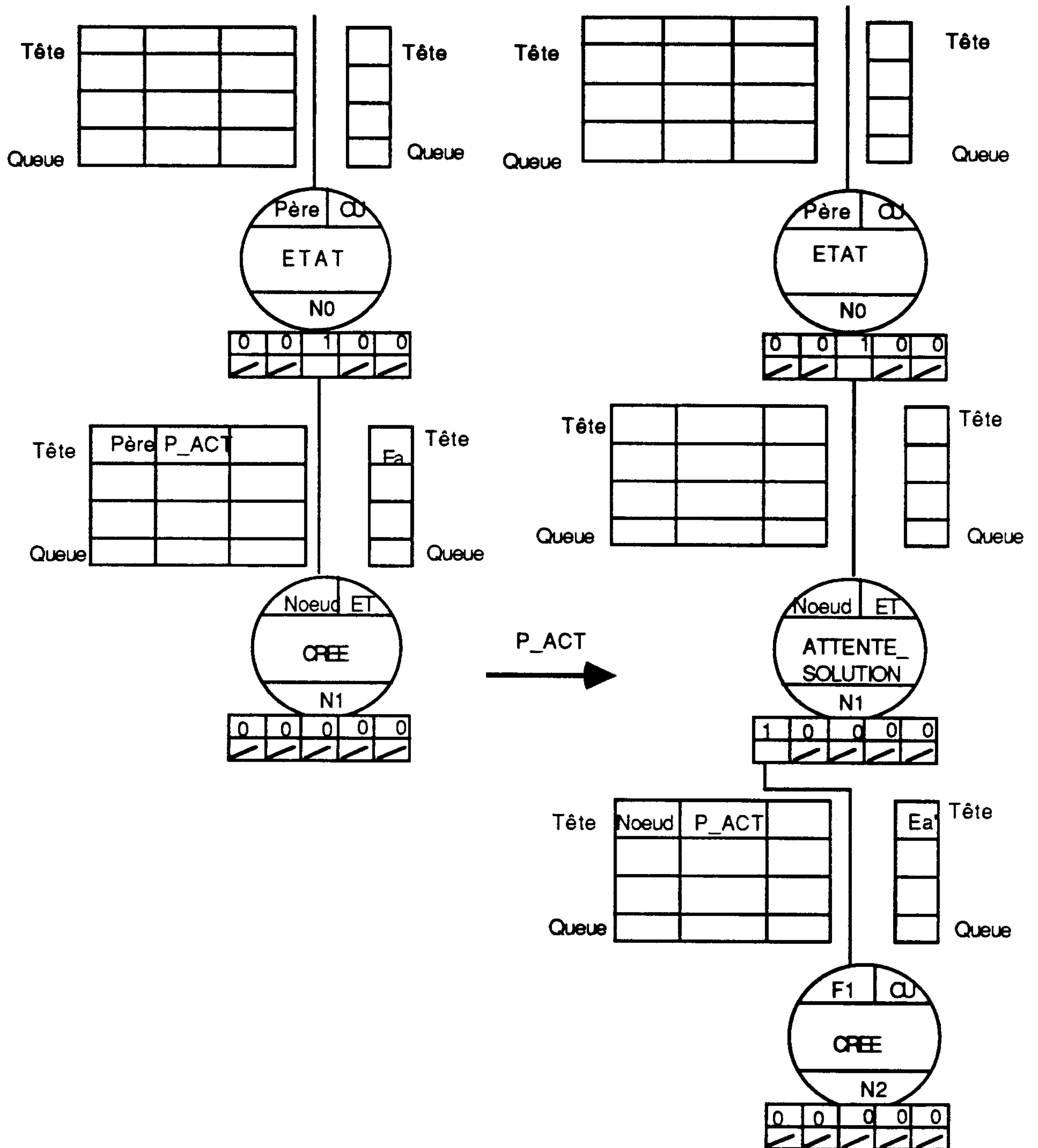


Figure 1.37

III. LA PRIMITIVE P_LOCK(Nœud_père, Nœud, Arguments).

L'interprétation de cette primitive est similaire à celle de la primitive P_ACT. Le nœud est donc créé, le lien père du nœud et le lien fils du nœud père sont associés. La primitive P_LOCK est déposée dans le buffer de primitives du nœud.

L'unification entre l'environnement transmis par la primitive et celui détenu par le nœud est alors effectuée. Comme précédemment, le résultat de cette unification est déterminant pour le comportement du nœud. Remarquons toutefois que le nœud ayant été activé en mode restreint, le nœud mémorise le cas échéant, dans son buffer d'environnements l'environnement résultat de l'unification.

Etudions chacune des situations précédemment évoquées :

III.1. L'unification échoue.

L'échec est mémorisé dans le buffer d'environnements jusqu'à réception d'une demande de solutions émanant du nœud père. L'état du nœud devient Attente_Levée_Verrou (cf figure 2.1).

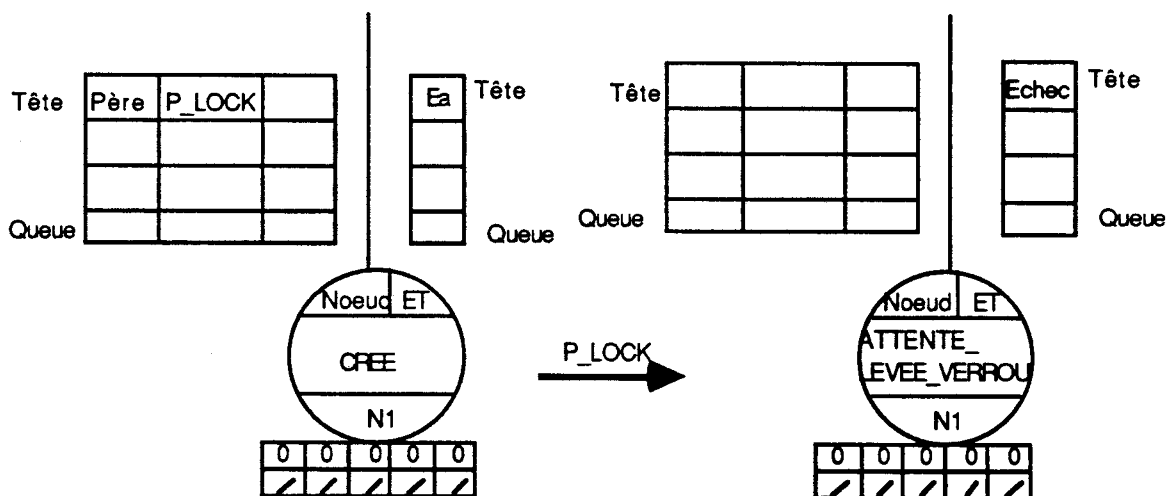


Figure 2.1

III.2. L'unification est un succès et le nœud n'a pas de fils(N1=0).

L'environnement solution est mémorisé dans le buffer d'environnements jusqu'à réception d'une demande de solution émanant

du nœud père. Comme dans la situation précédente, l'état du nœud devient Attente_Levée_Verrou (cf figure 2.2).

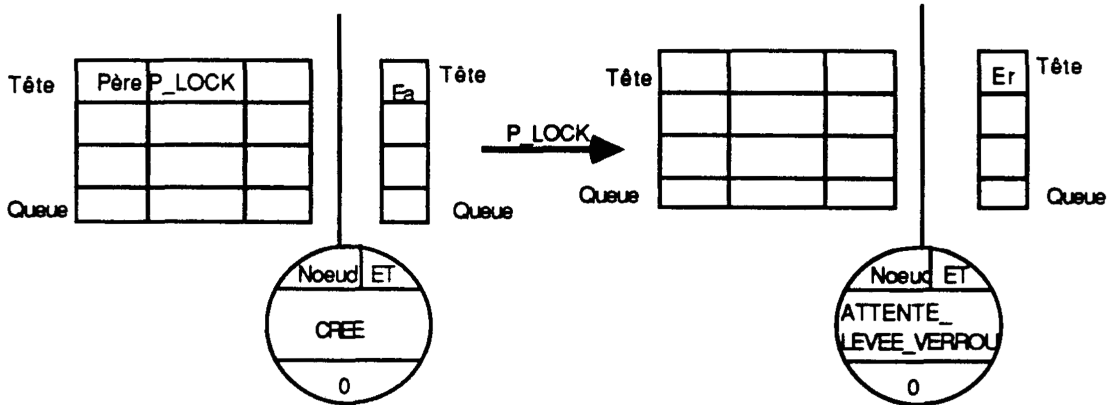


Figure 2.2

III.3. L'unification est un succès et le nœud possède au moins un fils (N1 >=1).

Le nœud active alors son premier fils, dont il attend un environnement solution. Il attend de plus une demande de solution émanant du nœud père : aussi, son état est-il en Attente_Double.

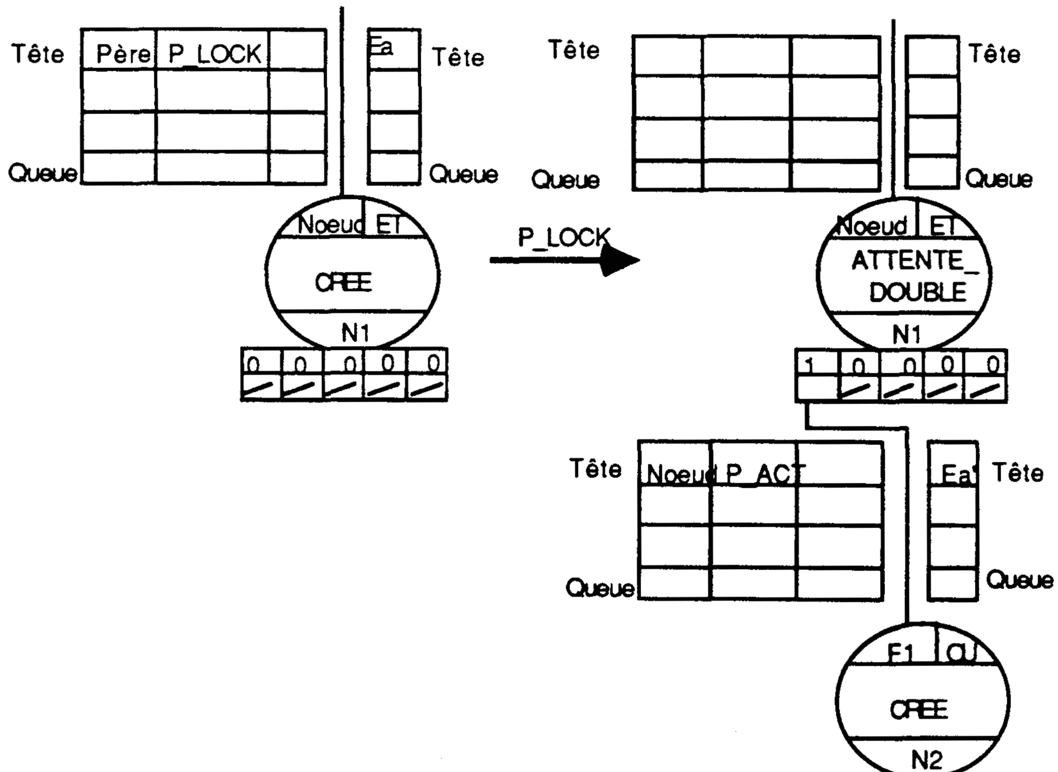


Figure 2.3

IV. LA PRIMITIVE P_UNLOCK(Nœud).

L'interprétation de cette primitive est réalisée en fonction du contenu du buffer d'environnements et de l'état courant du nœud.

Le buffer d'environnements peut contenir (ou non) un environnement solution, qui est soit un échec, soit une solution. Examinons ces trois situations.

IV.1. Un échec est mémorisé dans le buffer d' environnements

Cet échec est définitif, car les échecs intermédiaires sont ignorés. Le nœud n'est donc plus en mesure de calculer une autre solution. Le nœud émet alors deux primitives : l'une de type résultat, qu'il dépose dans le buffer de primitives de son père, la seconde de type destruction qu'il place dans son propre buffer de primitives. L'état du nœud devient alors résolu (cf figure 3.1).

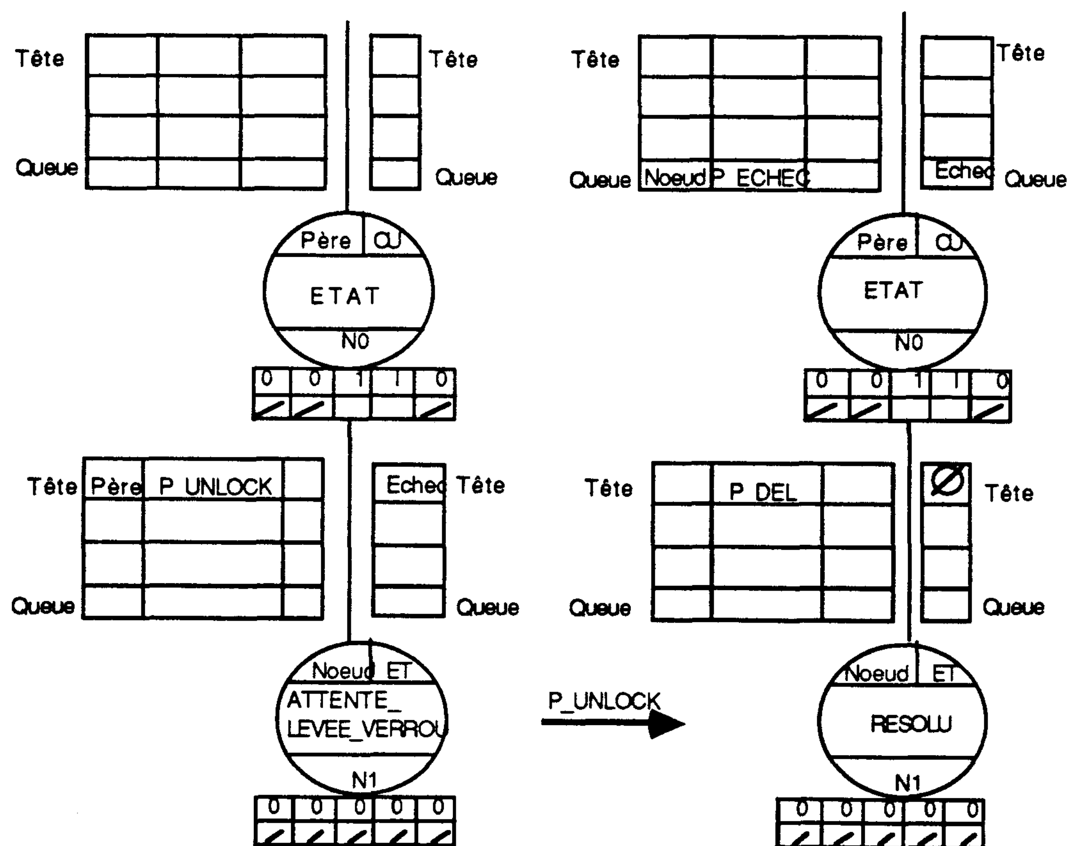


Figure 3.1

IV.2. L'environnement mémorisé n'est pas un échec.

Le nœud crée alors une primitive de type résultat, à l'intention du nœud père, dont la valeur des attributs premier et dernier reste à calculer.

Remarque :L'attribut premier est vrai si l'environnement-solution retourné est le premier contenu dans le buffer d'environnements.

- *L'attribut dernier est vrai* : l'activité de tous les nœuds fils est nulle, comme en témoigne la table de liens (tous les bits d'existence sont à zéro, tous les liens fils sont détruits). Le nœud dépose alors dans son propre buffer de primitives une primitive de destruction. Son état passe alors à résolu (cf figure 3.2).

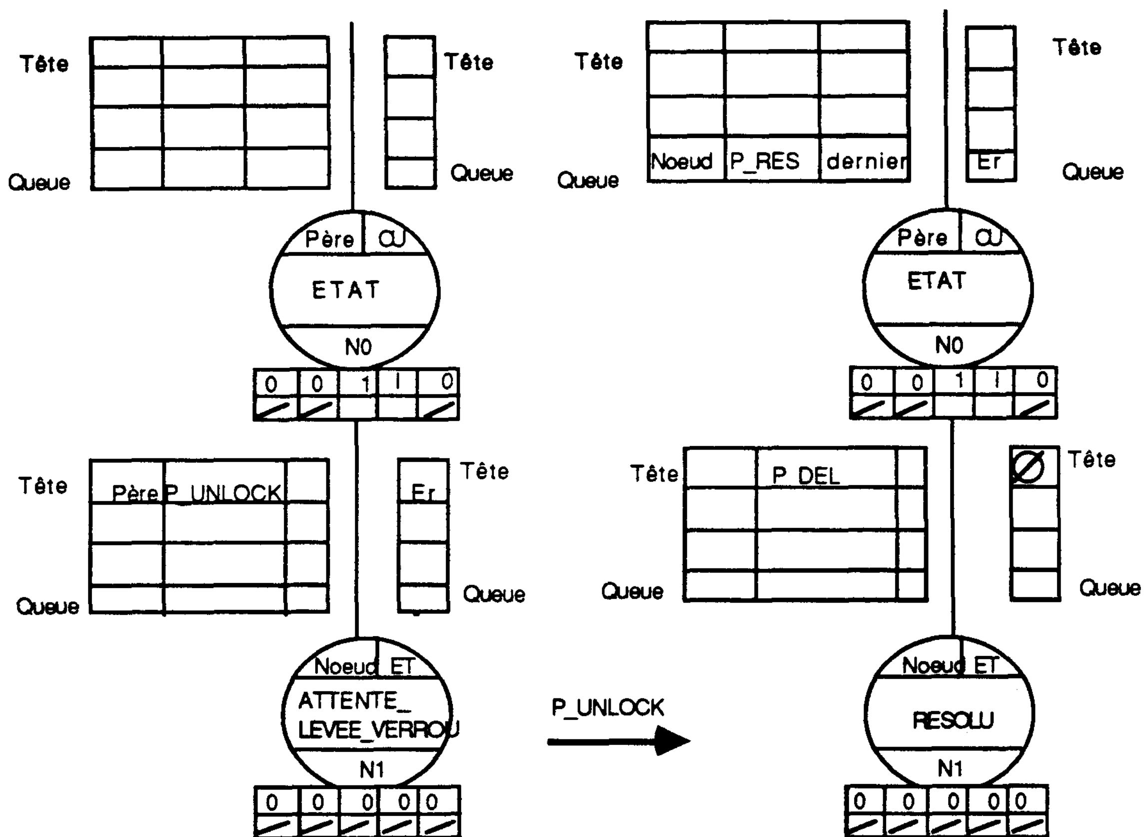


Figure 3.2

- *L'attribut dernier n'est pas vrai* : l'un au moins des nœuds fils est encore actif (le bit d'existence correspondant est à un). Le nœud recherche le fils d'indice le plus élevé dont le bit d'existence est à un. Ce

nœud fils est soit le dernier fils du nœud ou ne l'est pas. Nous allons examiner ces deux situations :

Cas 1. Le nœud fils est le dernier fils du nœud. Le nœud dépose dans le buffer de primitives de ce fils une demande de solutions. En effet, un nœud ne peut retourner consécutivement deux environnements-solutions de sa propre initiative. Les deux envois sont séparés par la réception d'une primitive demande de solution. Le dernier nœud fils a déjà retourné un environnement solution, en l'occurrence celui contenu dans le buffer d'environnements. Une demande de solution s'avère donc nécessaire (cf figure 3.3).

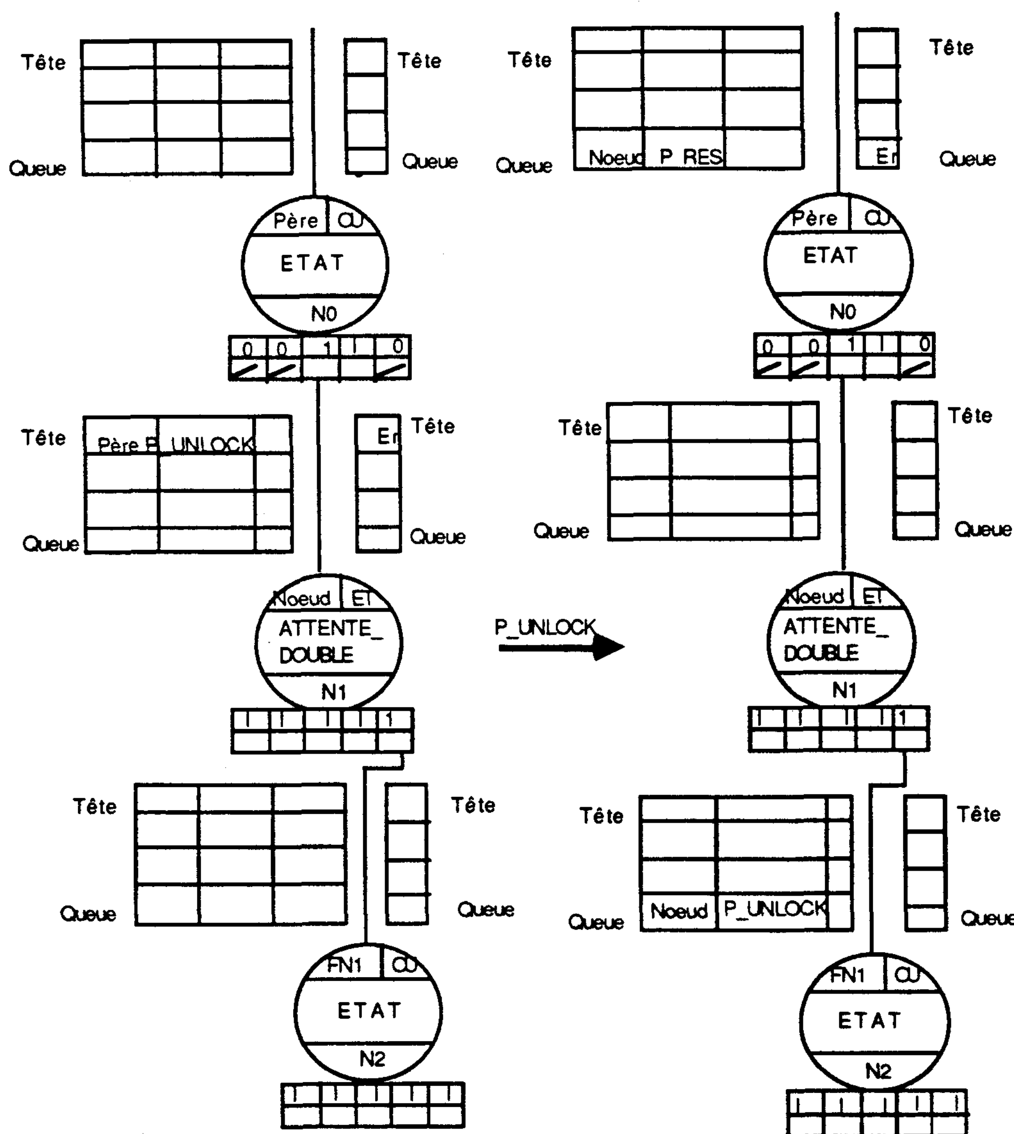


Figure 3.3

Cas 2. Le nœud fils encore actif n'est pas le dernier fils. Le nœud n'effectue aucune action en dehors de l'émission de la primitive résultat, car le pipeline est convenablement amorcé (cf figure 3.4). L'état du nœud reste Attente_Double.

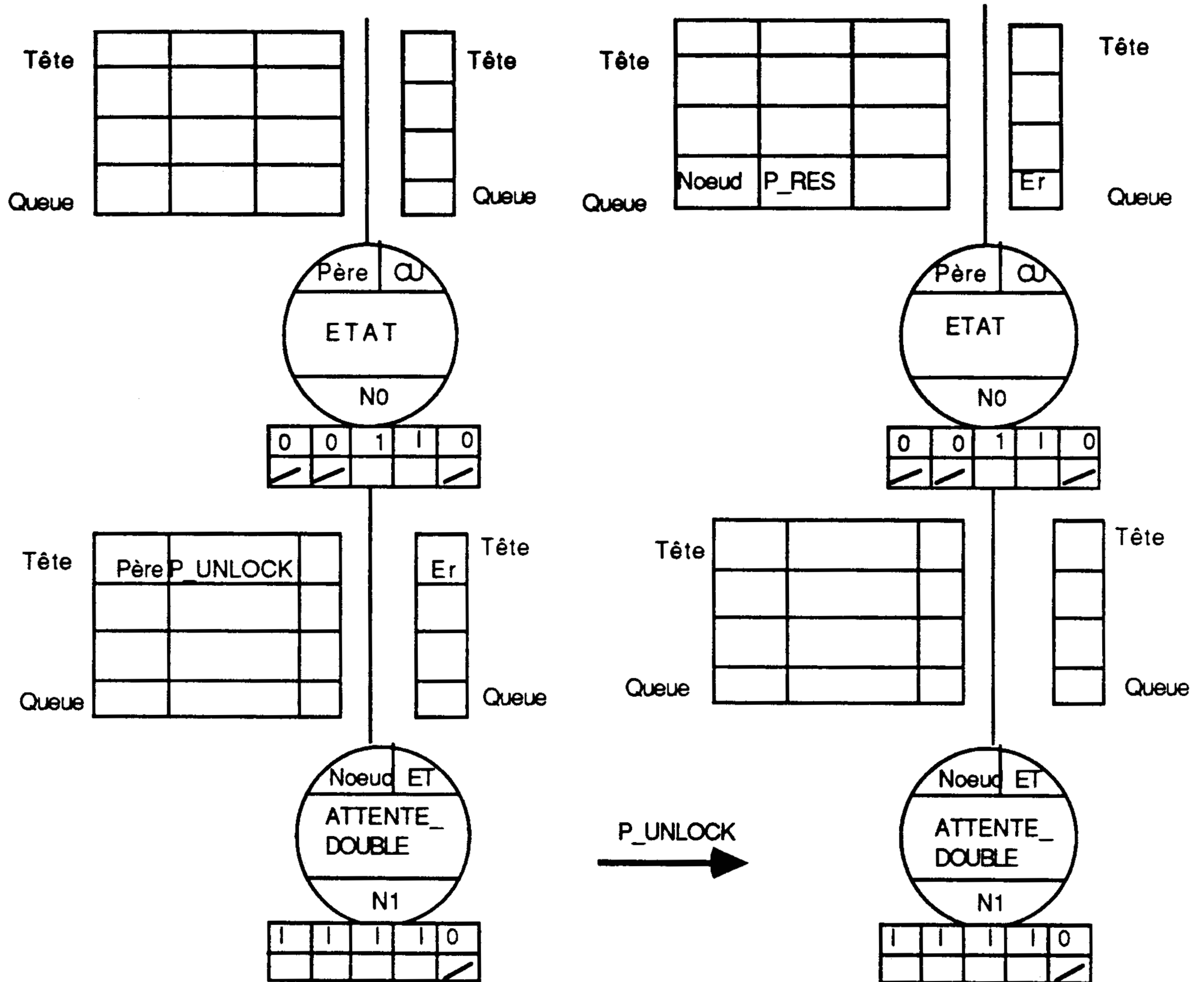


Figure 3.4

IV.3. Le buffer d'environnements est vide.

Seul, l'état du nœud est modifié. D'Attente_Double, il passe à Attente_ Solution (cf figure 3.5).

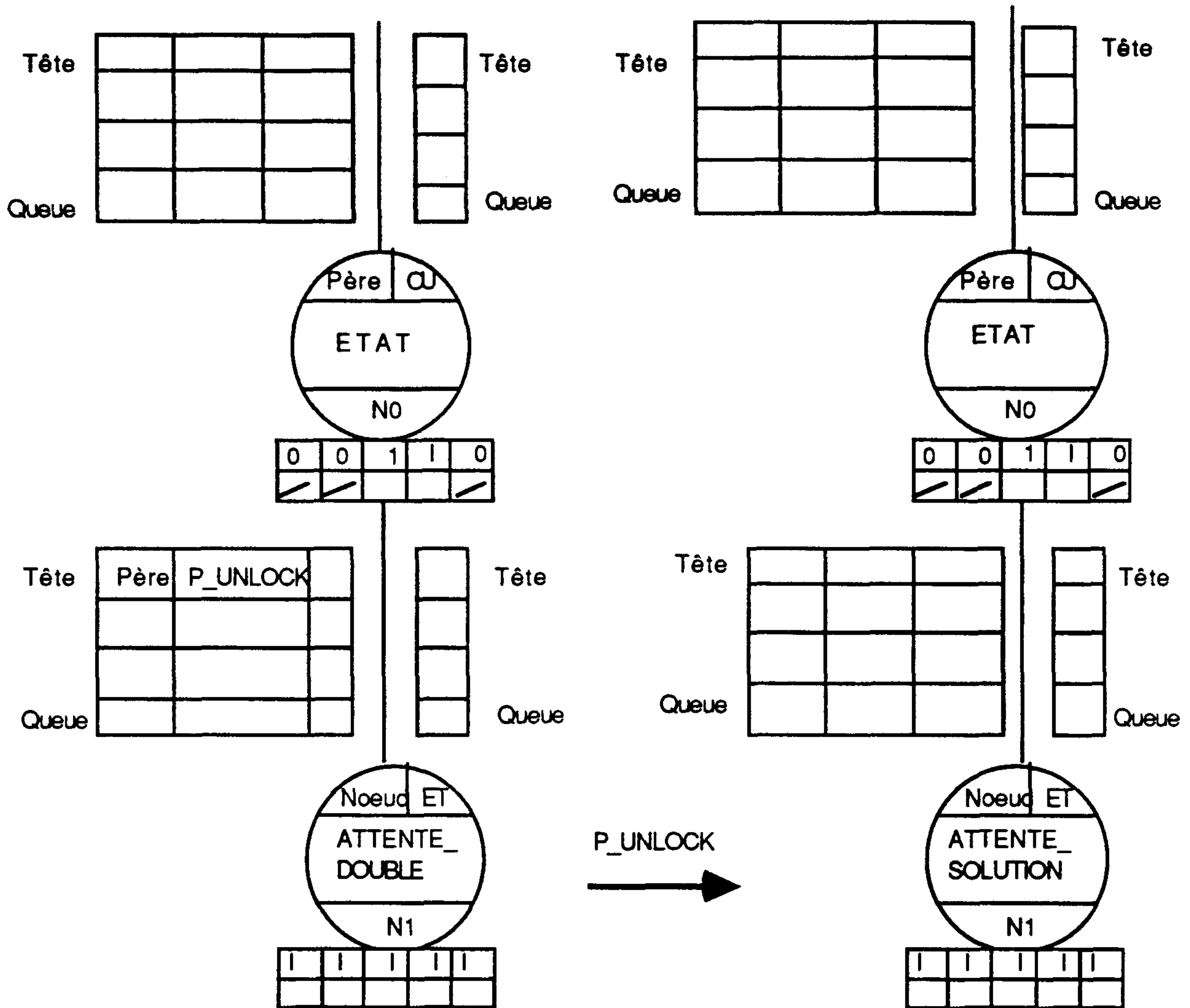


Figure 3.5

V. LA PRIMITIVE P_RES(Fk, Nœud, Attributs, Arguments).

Cette primitive traduit le retour d'un environnement solution, évalué par le nœud F_k vers le nœud. Un environnement solution est dit partiel d'ordre k s'il est calculé par le nœud fils F_k , complet s'il est construit par le dernier fils. Cette primitive est liée à l'avancée dans le pipeline d'activation ET, dont il faut assurer la continuité. En conséquence, l'interprétation de cette primitive peut modifier le comportement non

seulement des nœuds immédiatement successeurs du nœud F_k, mais encore celui du nœud prédécesseur de F_k. Ces modifications sont fonction d'une part de l'émetteur F_k, de la valeur des attributs premier et dernier de la primitive P_RES interprétée, et d'autre part de l'état courant du nœud.

Remarques :

1) Si l'attribut dernier de la primitive P_RES interprétée est vrai, l'arbre dont le nœud F_k, émetteur de la primitive, est la racine a été détruit. Le nœud F_k ne peut retourner d'autres solutions. Aussi, la table de liens est-elle mise à jour : le bit d'existence d'indice k est mis à zéro, le lien fils est détruit.

2) Deux nœuds fils successifs dans un pipeline ne peuvent être activés en mode "normal". En d'autres termes, ces deux nœuds fils ne peuvent retourner *simultanément* d'environnements-solution. Aussi, le nœud fils d'indice k+1 est activé en mode normal, et celui d'indice k en mode restreint.

V.1. L'émetteur de la primitive est le dernier fils du nœud (F_{N1}).

Le nœud détient alors un environnement solution complet qu'il retourne ou non à son père en fonction de son état courant.

V.1.a. L'attribut dernier de la primitive est vrai.

Le nœud F_{N1} est donc détruit. Le comportement du nœud est alors déterminé en fonction de son état, qui est soit Attente_Solution, soit Attente_Double.

Cas 1. L'état du nœud est Attente_Solution (le nœud n'attendait qu'une solution de l'un de ses fils). Il peut alors créer une primitive de type résultat, qu'il place dans le buffer de primitives du nœud père. Le nœud doit en outre maintenir le pipeline convenablement amorcé. A cette fin, il vérifie la présence du nœud fils d'indice N1-1, en testant le bit d'existence N1-1. Ce dernier est ou non égal à zéro.

* Le bit d'existence d'indice N1-1 est à zéro. Le nœud fils correspondant n'est donc plus actif. L'état du nœud est alors modifié en fonction du contenu de la table de liens.

Si tous les bits d'existence sont à zéro, l'attribut dernier de la primitive P_RES émise prend la valeur vrai. Le nœud est prêt à être détruit. Il dépose dans son buffer de primitives une primitive de destruction. son état devient résolu(cf figure 4.1).

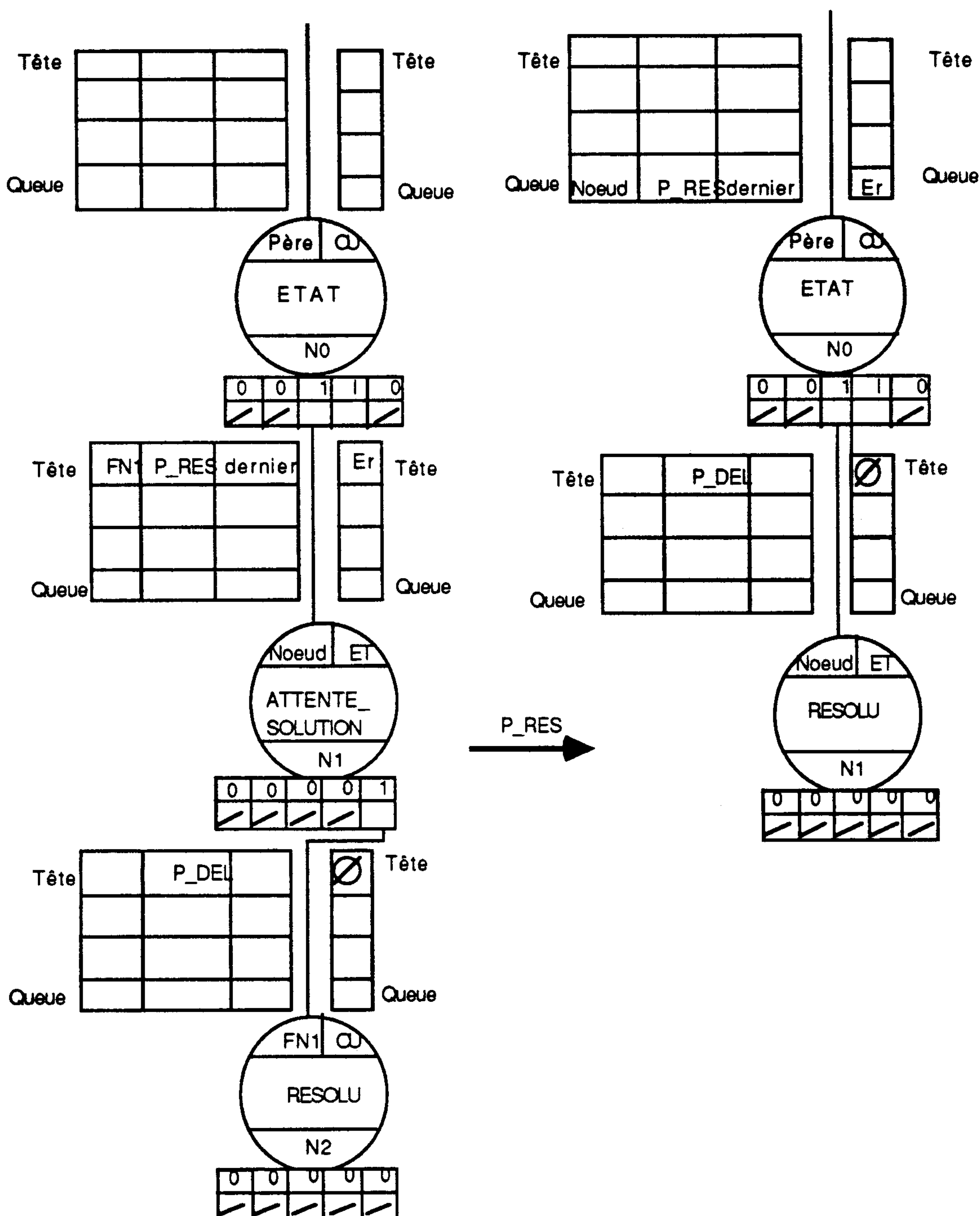


Figure 4.1.

Si l'un au moins des bits d'existence est à un, l'état du nœud devient Attente_Double (cf figure 4.2). En effet, un des nœuds fils du nœud est encore actif. De plus, le nœud émettant une primitive de type résultat, doit attendre une nouvelle demande de solutions.

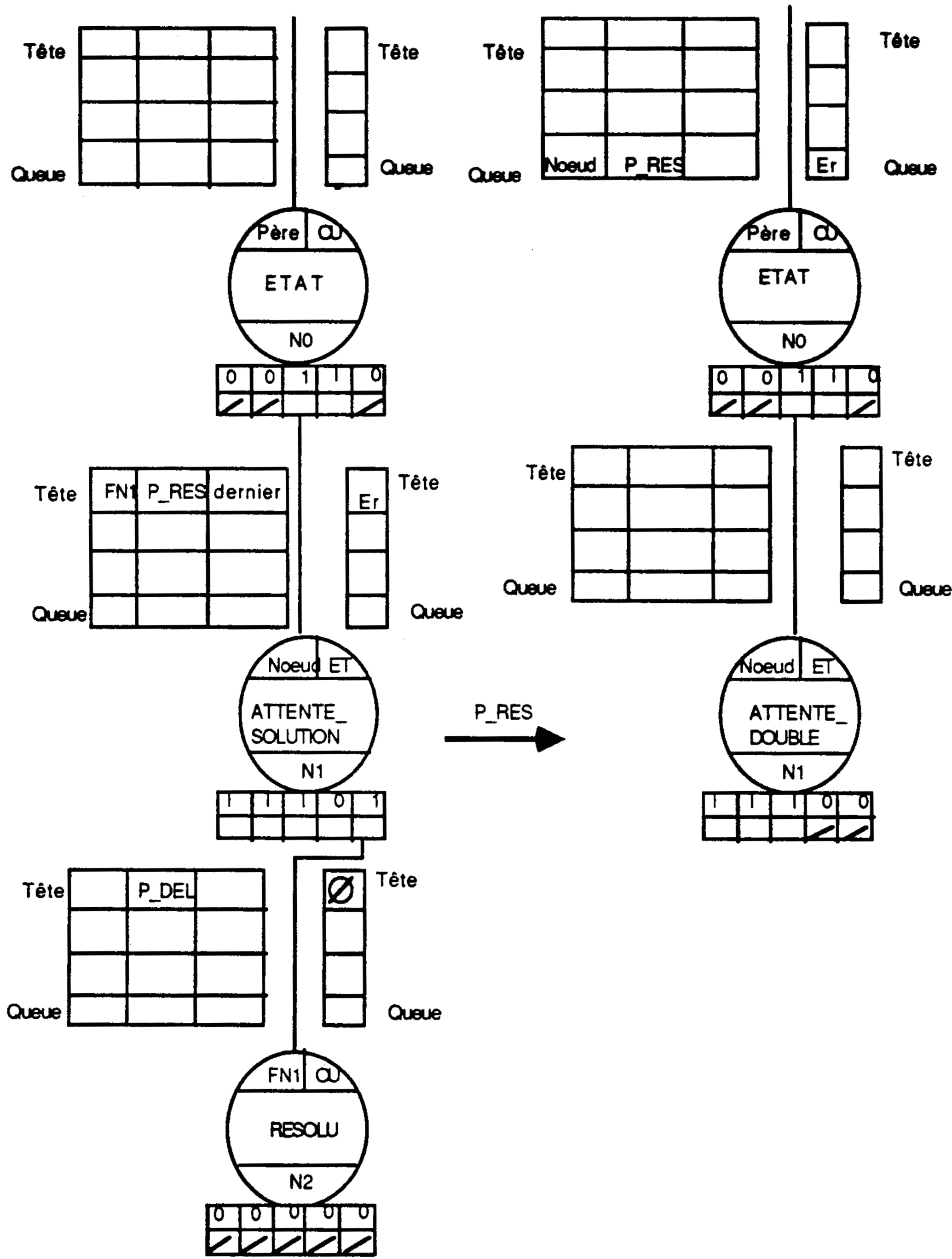


Figure 4.2.

* Le bit d'existence d'indice N1-1 est à un. Le nœud fils correspondant est donc actif et attend une demande de solution. Aussi, le nœud émet une primitive demande de solution à l'intention du nœud d'indice N1-1 (cf figure 4.3).

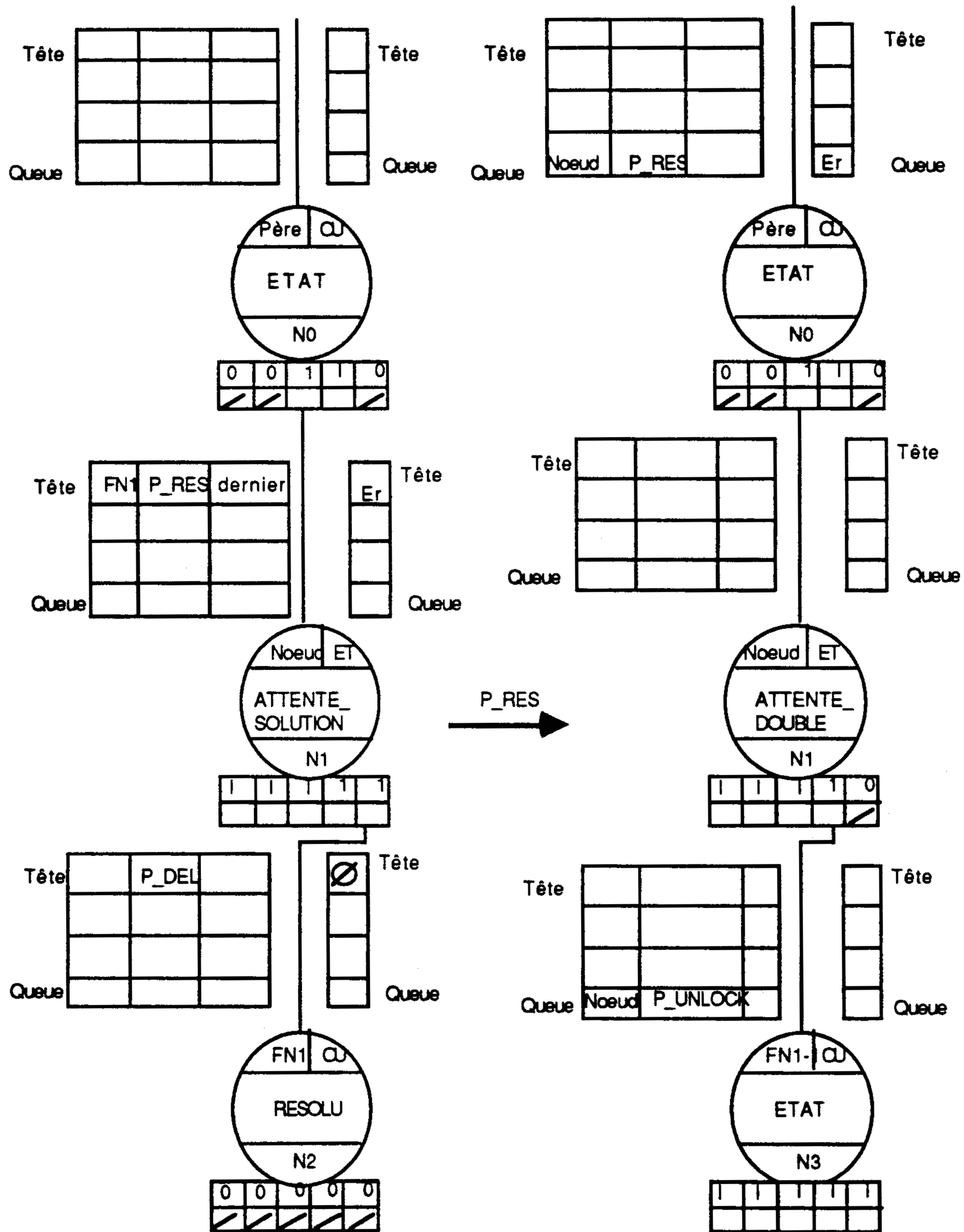


Figure 4.3

Cas 2. L'état du nœud est Attente_Double (le nœud attend conjointement une demande de solutions et une solution). Cette solution est mémorisée dans le buffer d'environnements, jusqu'à réception d'une demande de solutions. Le bon fonctionnement du pipeline ne doit pas être perturbé. Aussi, le nœud teste le bit d'existence d'indice N1-1.

* Si le bit d'existence d'indice N1-1 est à zéro, le nœud F_{N1-1} est inactif. Le nœud n'effectue aucune action en dehors de la mémorisation de l'environnement solution(cf figure 4.4)

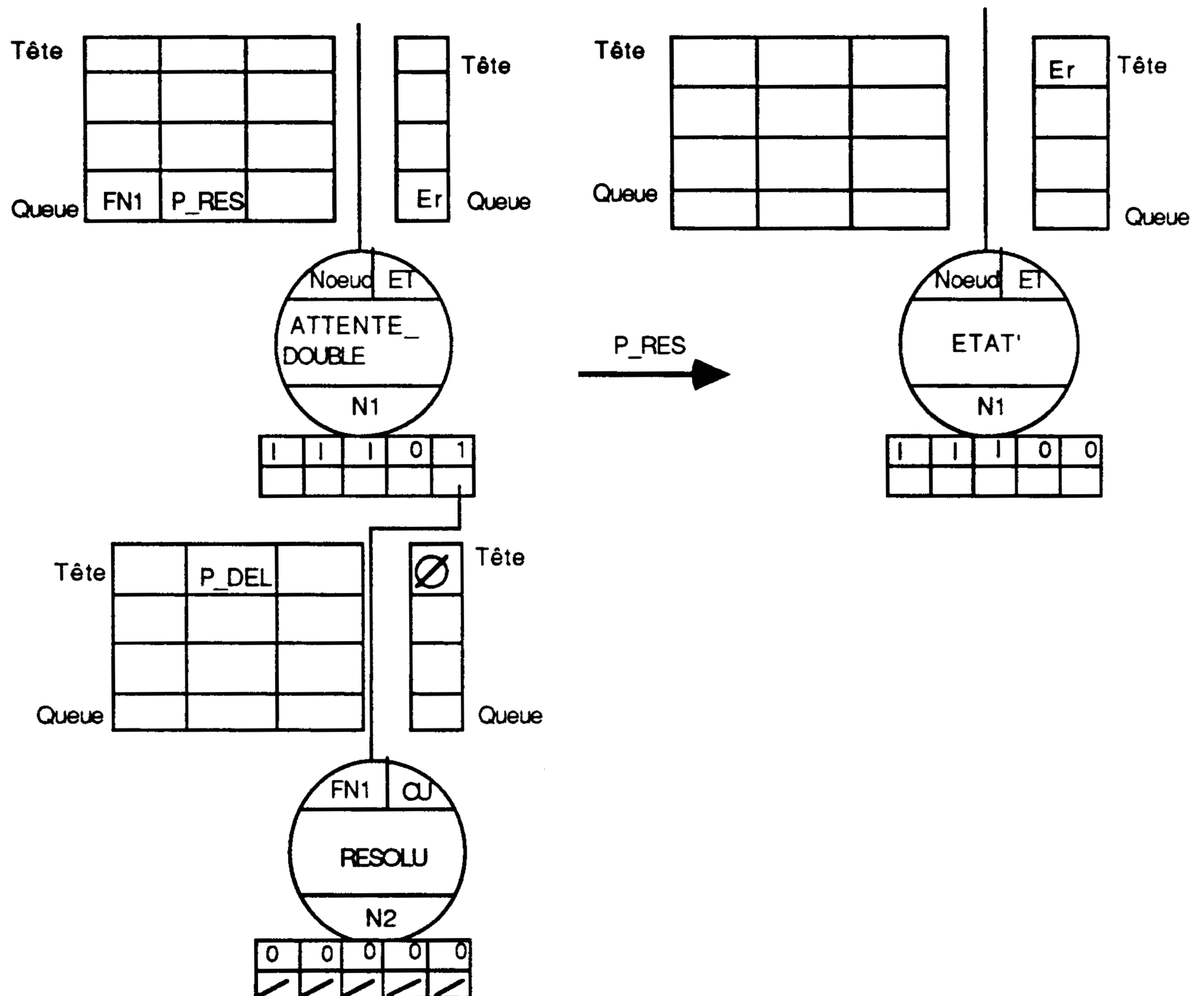


Figure 4.4.

Le bit d'existence N1-1 est à un. Le nœud dépose dans le buffer de primitives du nœud fils F_{N1-1} une demande de solutions. L'état du nœud devient en Attente_Double (cf figure 4.5).

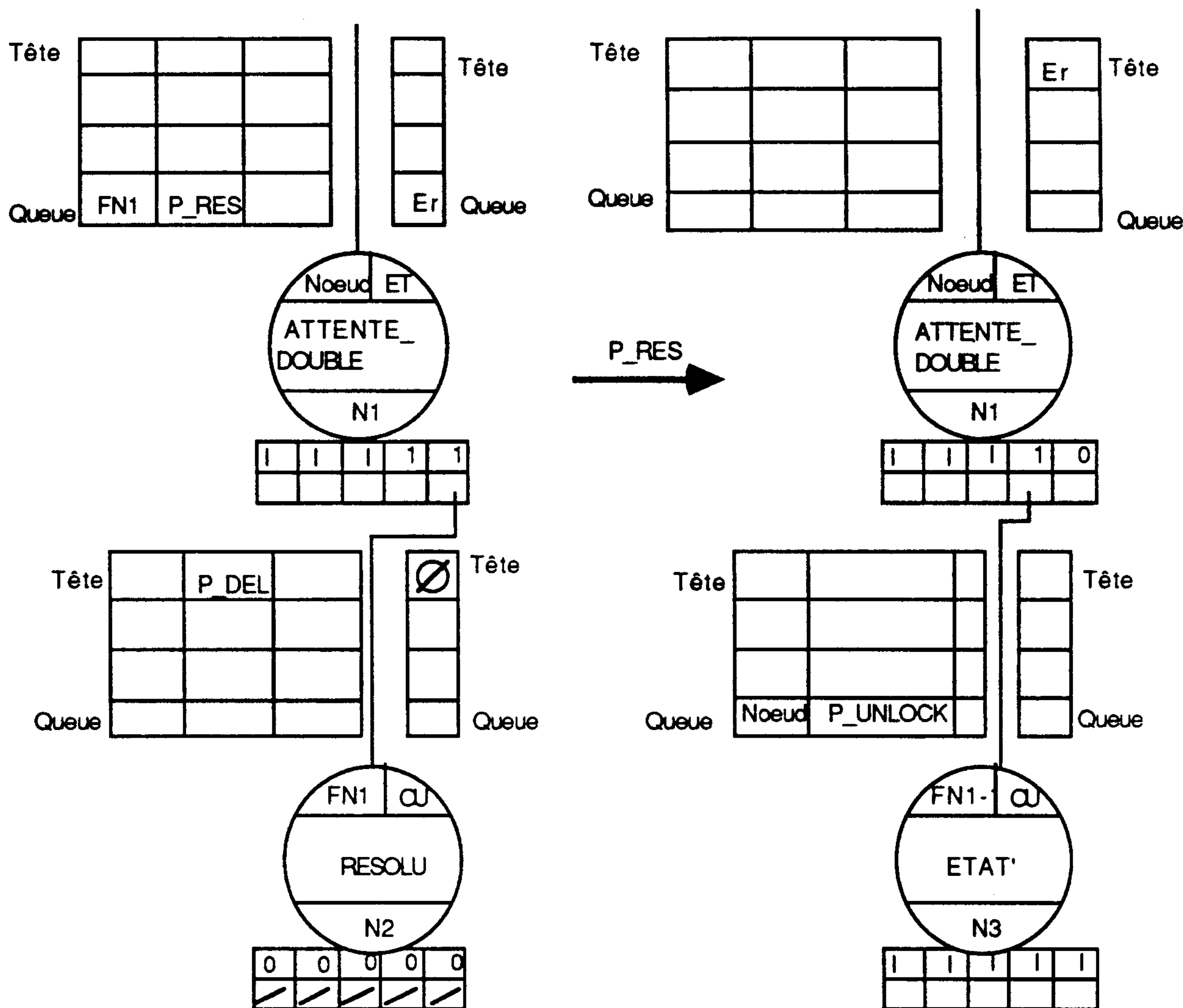


Figure 4.5.

V.1.b. L'attribut dernier n'est pas positionné.

Le nœud F_{N1} est toujours actif (le bit d'existence d'indice N1-1 est à un). Tout dépend alors de l'état courant du nœud.

Cas 1. Le nœud est en attente double. L'environnement reçu est mémorisé et tout dialogue entre le nœud et son père est suspendu jusqu'à réception par le nœud d'une demande de solution émanant du nœud père (cf figure 4.6).

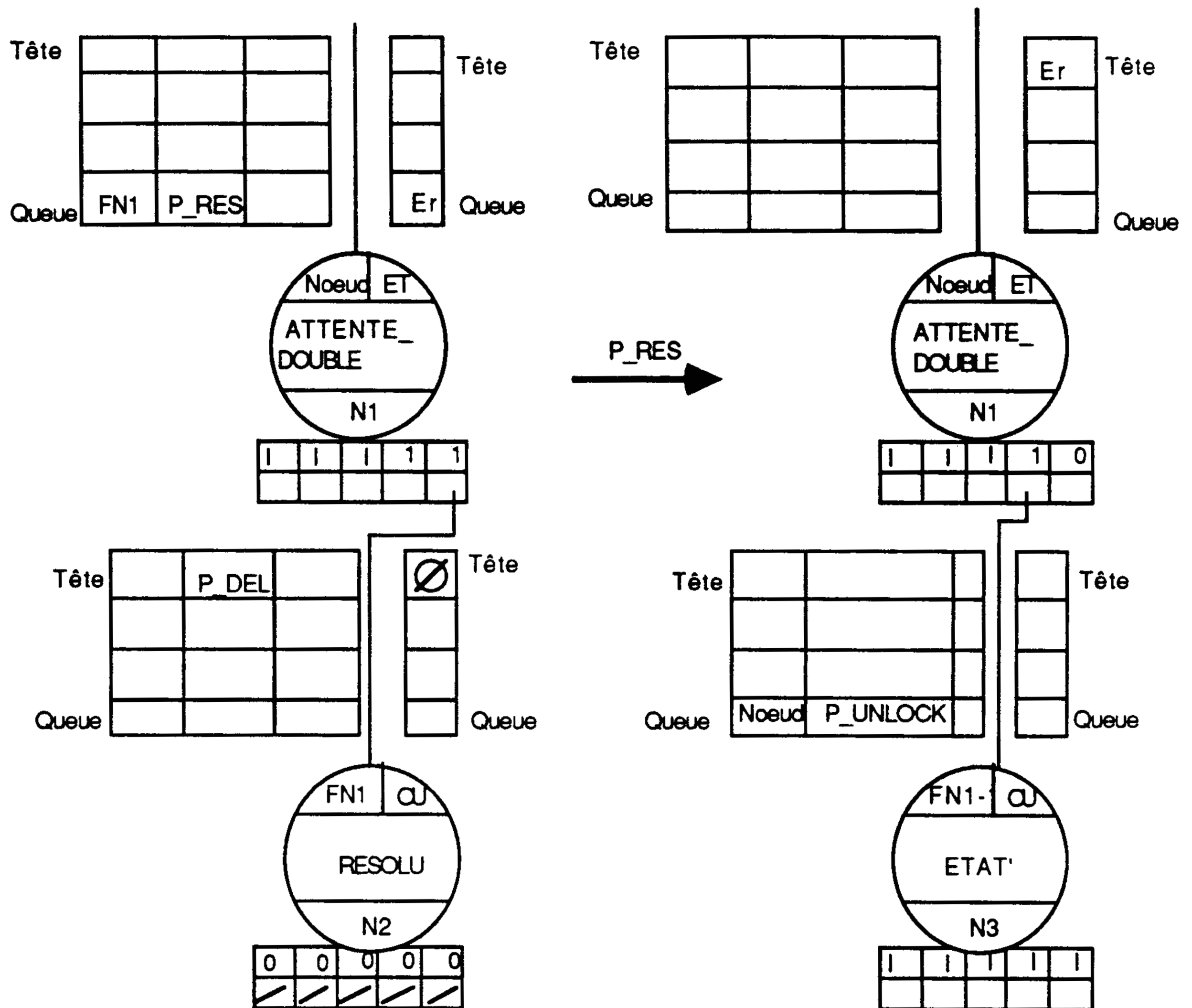


Figure 4.5.

V.1.b. L'attribut dernier n'est pas positionné.

Le nœud F_{N1} est toujours actif (le bit d'existence d'indice N1-1 est à un). Tout dépend alors de l'état courant du nœud.

Cas 1. Le nœud est en attente double. L'environnement reçu est mémorisé et tout dialogue entre le nœud et son père est suspendu jusqu'à réception par le nœud d'une demande de solution émanant du nœud père (cf figure 4.6).

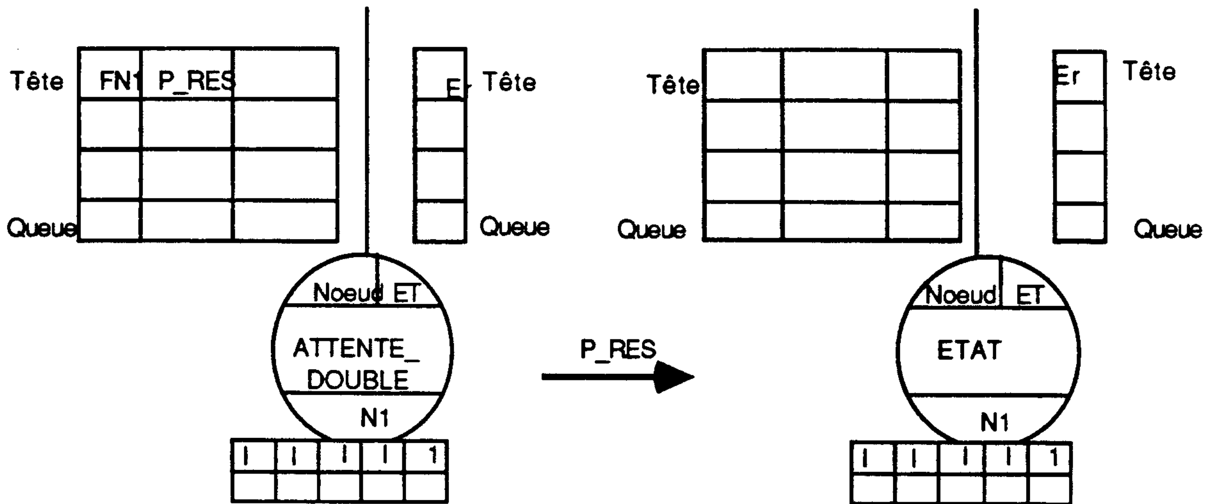


Figure 4.6

Cas 2. L'état du nœud est Attente_Solution. Le nœud n'attendant qu'une primitive de type résultat, crée deux primitives, en l'occurrence une primitive de type résultat et une demande de solutions qu'il place dans les buffers de primitives respectifs de son nœud père et de son dernier fils (cf figure 4.7).

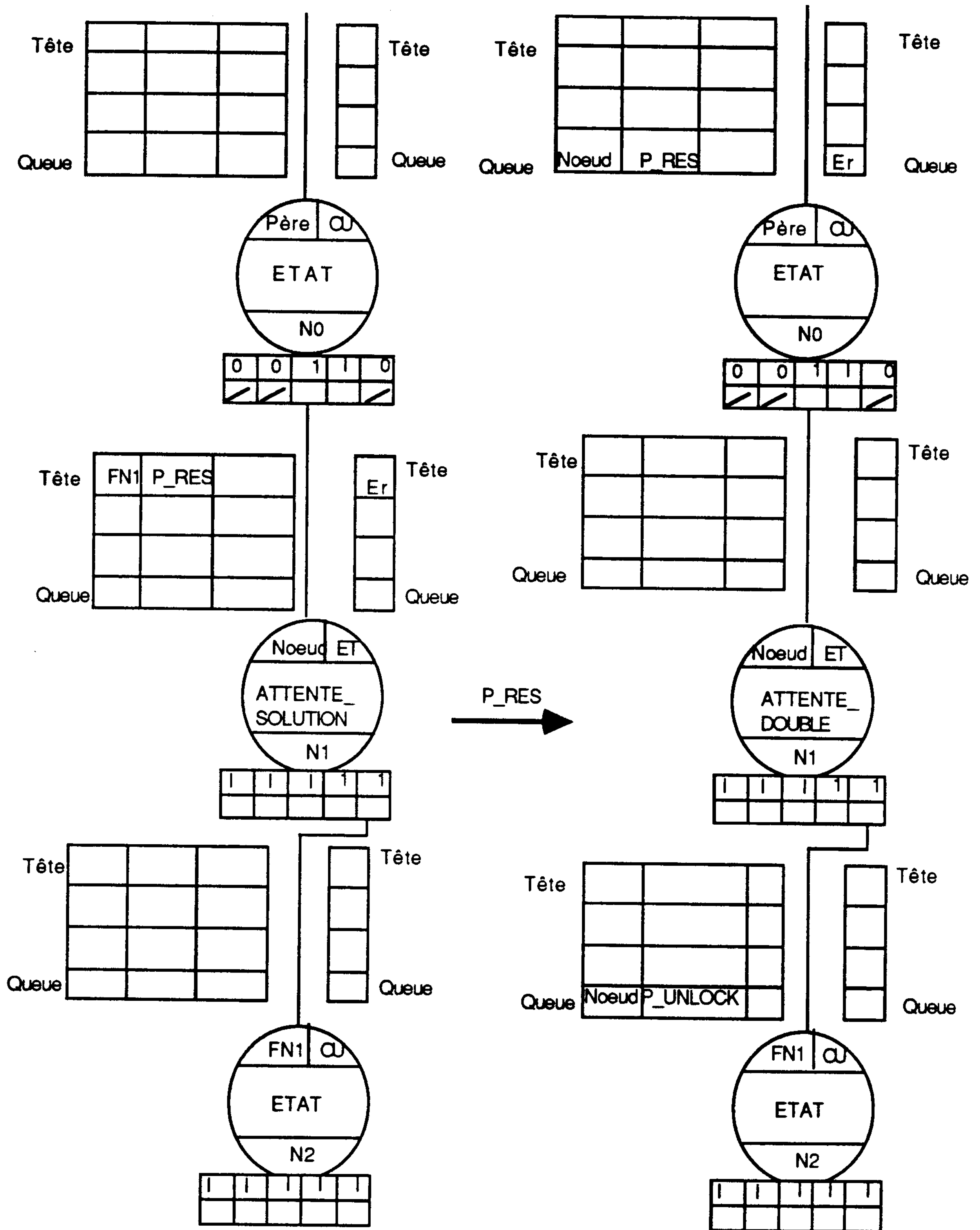


Figure 4.7

BU LILLE

V.2. L'émetteur de la primitive n'est pas le dernier fils du nœud

Le nœud détient alors un environnement solution partiel d'ordre k. Cet environnement-solution est (ou non) le dernier évalué par le nœud F_k. Nous allons donc examiner ces deux possibilités, et constater leurs incidences quant à la gestion du pipeline.

Remarque : Dans cette partie, l'état du nœud est indifféremment Attente_Solution, Attente_Double.

V.2.a. L'attribut dernier de la primitive est vrai.

Le nœud F_k peut être détruit. Le nœud met à jour la table de liens : le bit d'existence d'indice k est mis à zéro, le lien détruit. Le nœud doit de plus maintenir une activité continue du pipeline. Il effectue donc les actions suivantes :

1) Il active le nœud fils successeur du nœud fils F_k, en l'occurrence le nœud fils F_{k+1}. Le type d'activation est déterminé d'une façon similaire à celle décrite dans le paragraphe précédent.

2) Il teste l'activité du nœud prédécesseur du nœud F_k, dont l'indice est k-1. Ce dernier est actif (le bit d'existence d'indice k-1 est à un), ou non.

Dans le premier cas, le nœud émet une demande de solutions destinée au nœud fils F_{k+1}(car deux nœuds fils consécutifs dans un pipeline ne peuvent être activés tous deux en mode normal : or, le nœud F_k était activé en mode normal). Dans le second cas, le nœud ne réalise aucune action relative au nœud fils F_{k-1}.

Le nœud peut donc faire face aux quatre situations suivantes.

Cas 1. Le bit d'indice k+2 est nul. Le nœud fils F_{k+2} est donc inexistant. Le nœud émet deux primitives, qui sont une demande de solution et une activation. Celles-ci sont respectivement déposées dans les buffers de primitives des noeuds F_{k-1} et F_{k+1} (figure 4.8).

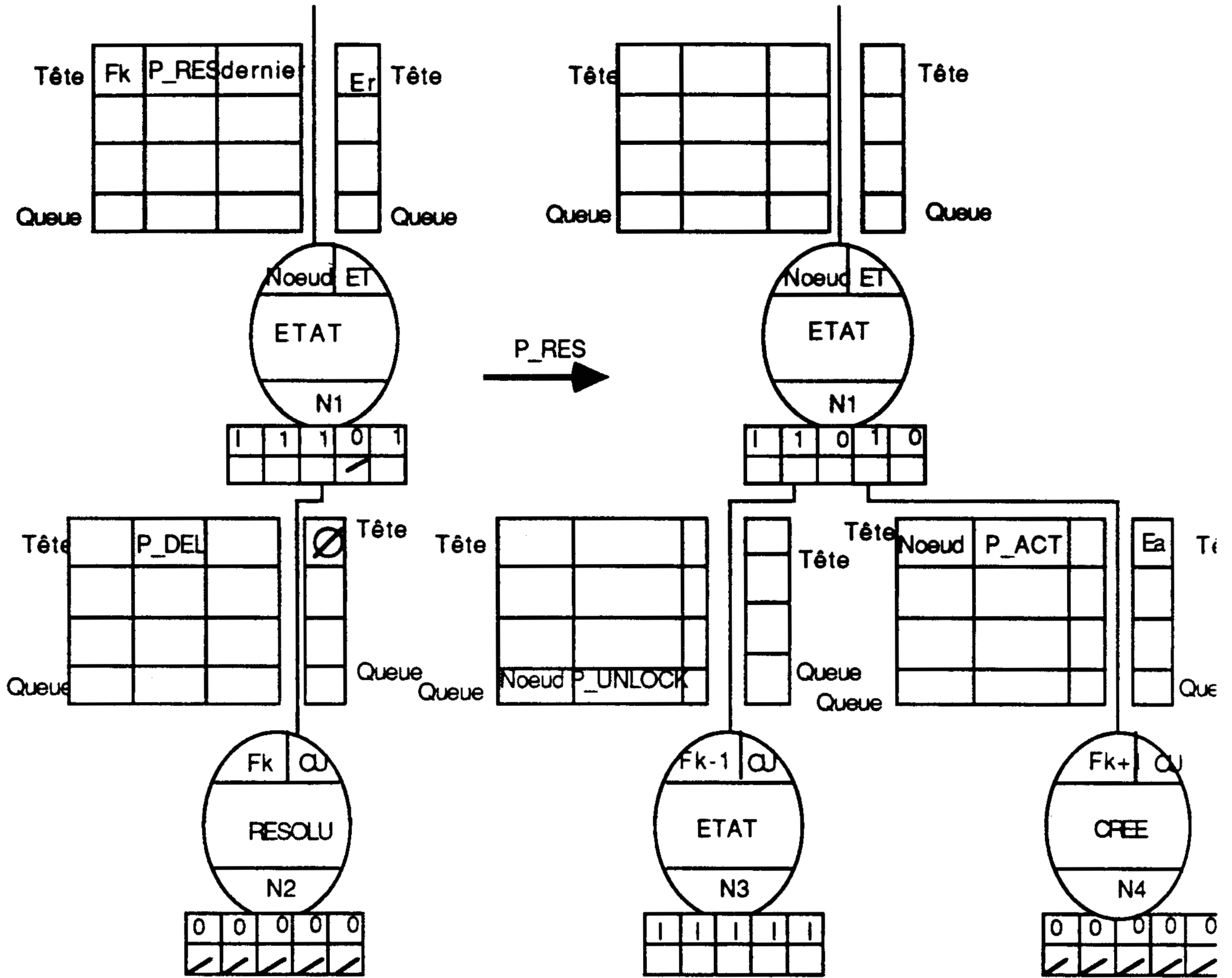


Figure 4.8.

Cas 2. Les deux nœuds F_{k-1} et F_{k+2} sont actifs. Seule diffère, par rapport au cas précédent, le type d'activation du nœud F_{k+1}. L'activation s'effectue en mode restreint (figure 4.9).

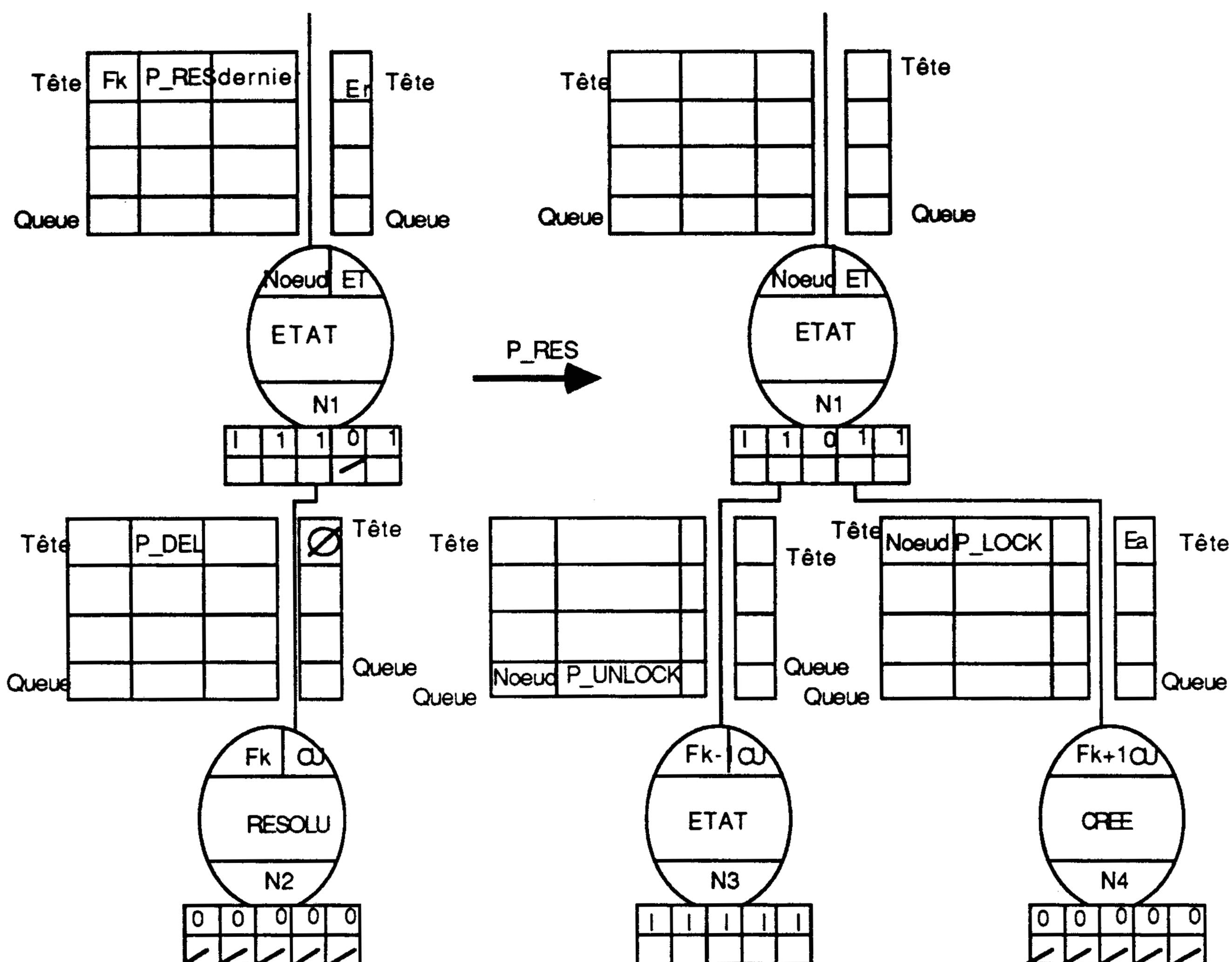


Figure 4.9

Cas 3. Les deux nœuds fils F_{k-1} et F_{k+2} sont inactifs. Le nœud active en mode normal le nœud fils F_{k+1} (cf figure 4.10) de manière à compléter l'environnement solution obtenu. Aucune autre action n'est à réaliser.

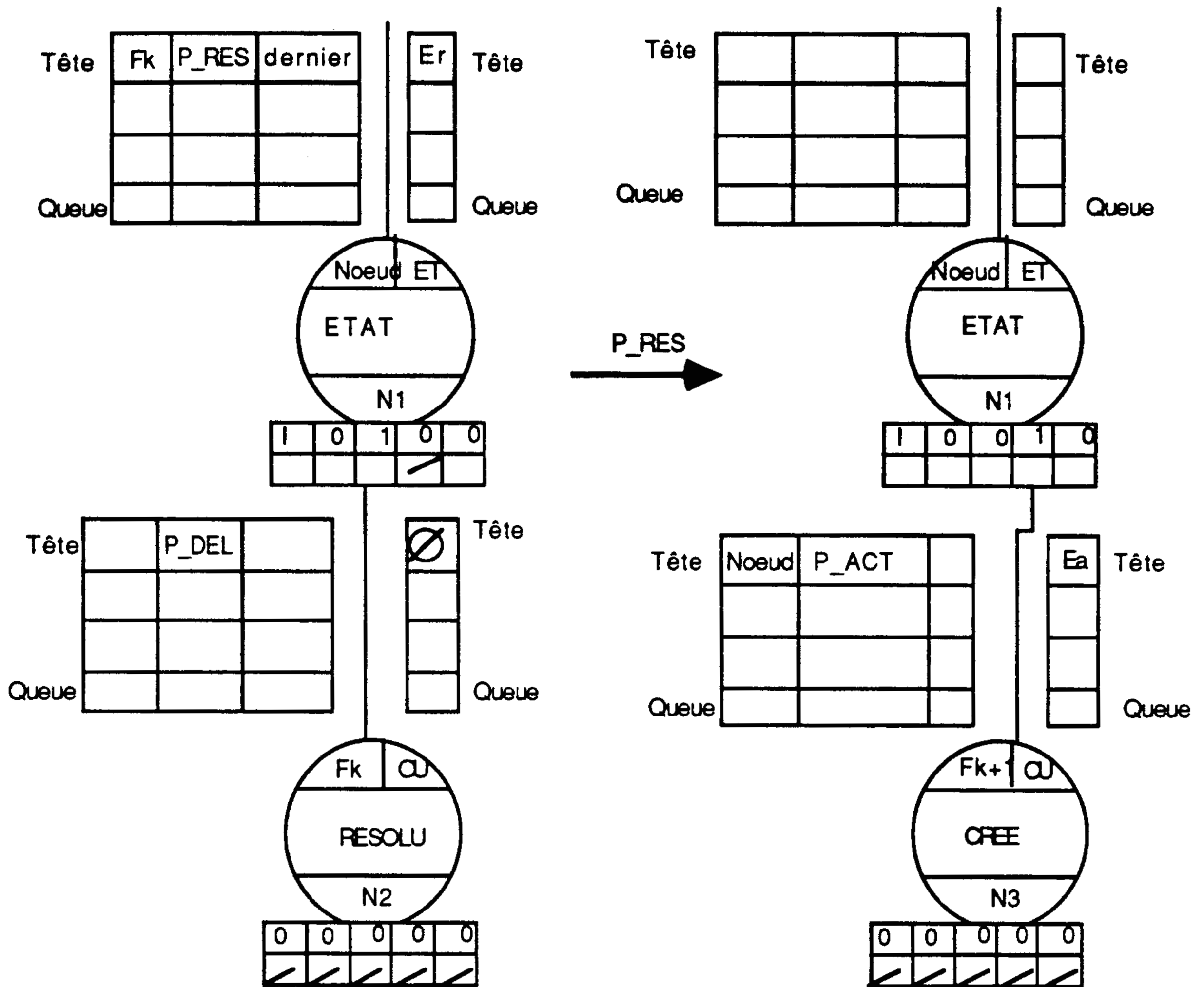


Figure 4.10

Cas 4. Le nœud fils F_{k-1} est inactif, le nœud fils F_{k+2} est actif. Ce cas est similaire au précédent. Seul, le type d'activation est modifié. L'activation du nœud fils F_{k+1} est une activation en mode restreint (figure 4.11).

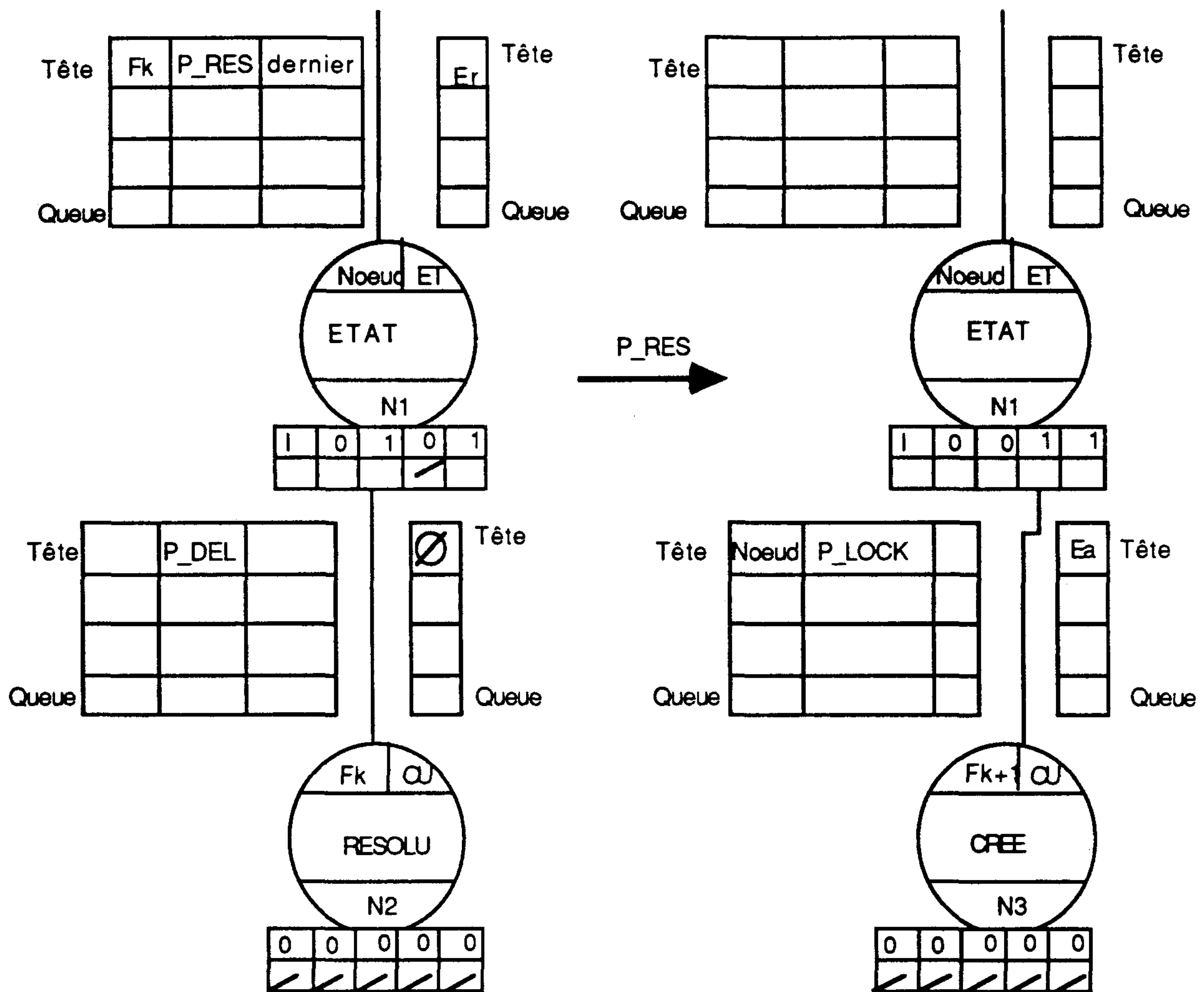


Figure 4.11

V.2.b. L'attribut dernier de la primitive n'est pas vrai.

Afin de compléter l'environnement détenu, le nœud active le nœud fils successeur, soit F_{k+1}. Le type d'activation est relatif à l'activité du nœud F_{k+2}.

Cas 1. Le nœud F_{k+2} est soit inactif, soit inexistant. Le nœud F_{k+1} est activé en mode normal (cf figure 4.12), c'est-à-dire par une primitive P_ACT.

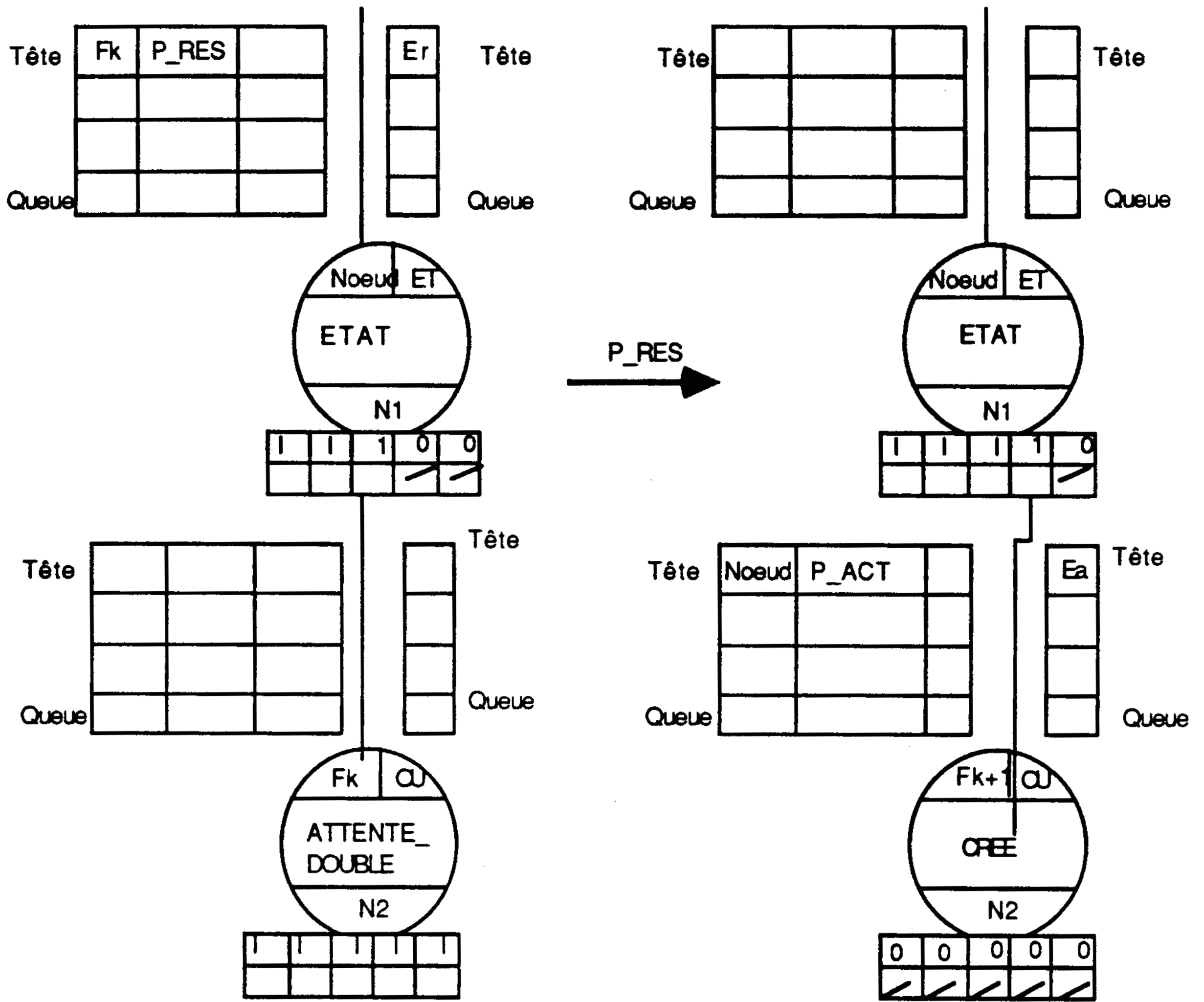


Figure 4.12

Cas 2. Le nœud F_{k+2} est actif. Deux fils consécutifs dans un pipeline ne peuvent être activés en mode normal. Le nœud fils dont l'indice est le plus petit des deux est activé en mode restreint. C'est le cas du nœud F_{k+1} (cf figure 4.13).

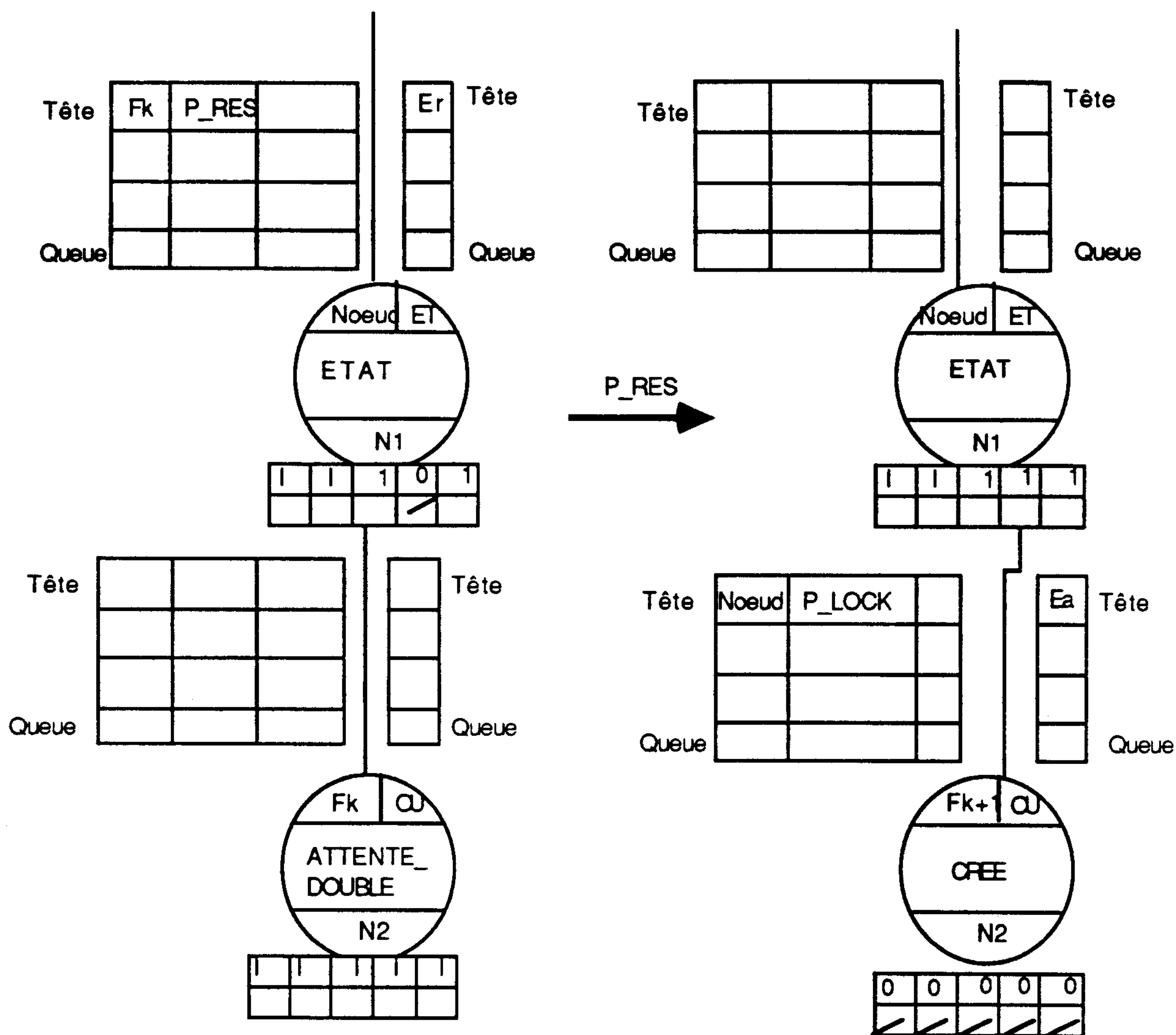


Figure 4.13

VI. LA PRIMITIVE P_ECHEC(Fk,Nœud).

Cette primitive transmet l'échec évalué par le nœud F_k vers le nœud. Le nœud met à jour sa table de liens : le bit d'existence d'indice k est mis à zéro, le lien détruit. De plus, le nœud ne mémorise cet échec que s'il est définitif. Aussi, tente-t-il d'obtenir un autre environnement-solution.

Il consulte donc sa table de liens afin de déterminer si l'un de ses fils est toujours actif.

VI.1. L'échec obtenu est définitif.

Lorsque l'échec obtenu est définitif, aucun nœud fils n'est actif (tous les bits d'existence sont à zéro). Le comportement du nœud varie en fonction de l'état courant du nœud.

Cas 1. Le nœud est en Attente_Double. L'échec est mémorisé dans le buffer d'environnements et l'état du nœud devient en Attente_Levée_Verrou (cf figure 5.1).

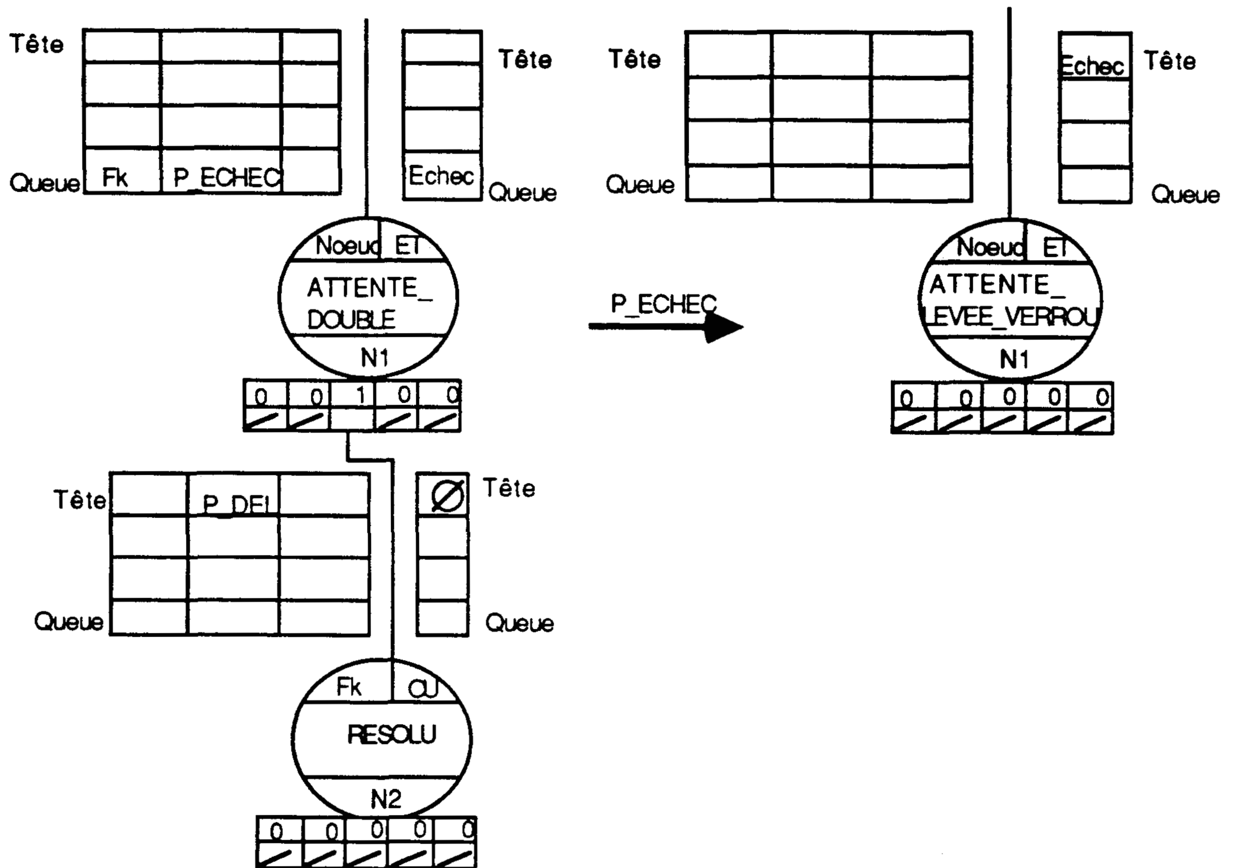


Figure 5.1

Cas 2. Le nœud est en Attente_Solution. Le nœud émet alors deux primitives. Ces deux primitives sont respectivement une primitive d'échec à l'intention de son propre père, et une primitive de destruction qui lui est destinée. Son état devient alors résolu (cf figure 5.2).

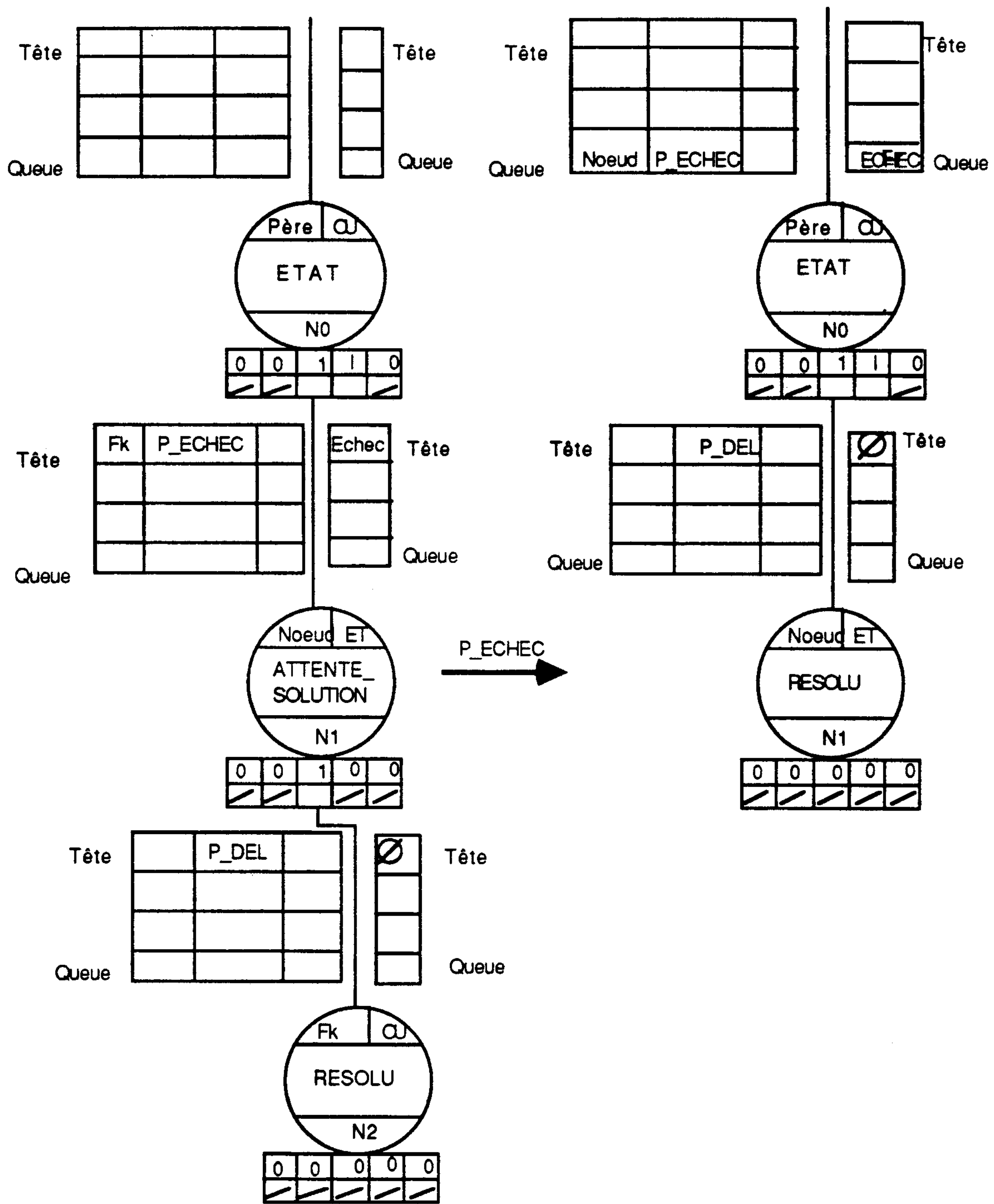


Figure 5.2

VI.2. L'échec obtenu n'est pas définitif.

L'un au moins des prédécesseurs du nœud F_k est encore actif. Le nœud cherche à déterminer le nœud fils F_i répondant aux critères suivants :

- F_i est actif
- i est inférieur à k
- il n'existe aucun fils actif entre les fils F_i et F_k (il n'existe pas d'indice j, tel que j soit compris entre i et k et le bit d'existence d'indice j soit à un).

Deux cas se présentent :

Cas 1. Le nœud fils F_i déterminé est le prédécesseur immédiat de F_k ($i = k-1$) : il est alors activé en mode restreint. Le nœud émet donc une demande de solutions à l'intention du nœud F_i (cf figure 5.3).

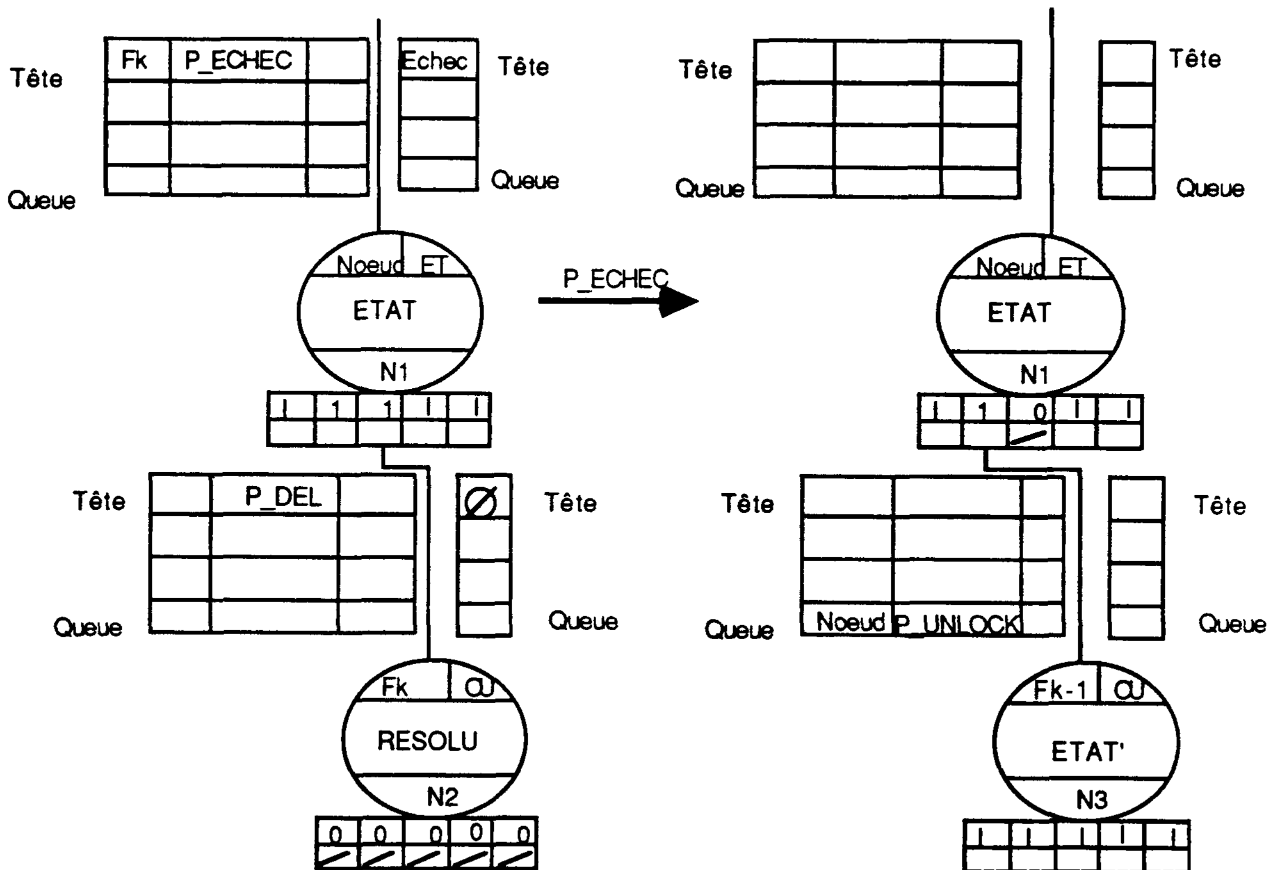


Figure 5.3

Cas 2. Le nœud F_i n'est pas le prédécesseur immédiat de F_k : il est alors activé en mode normal. Le nœud n'effectue alors aucune action supplémentaire (cf figure 5.4).

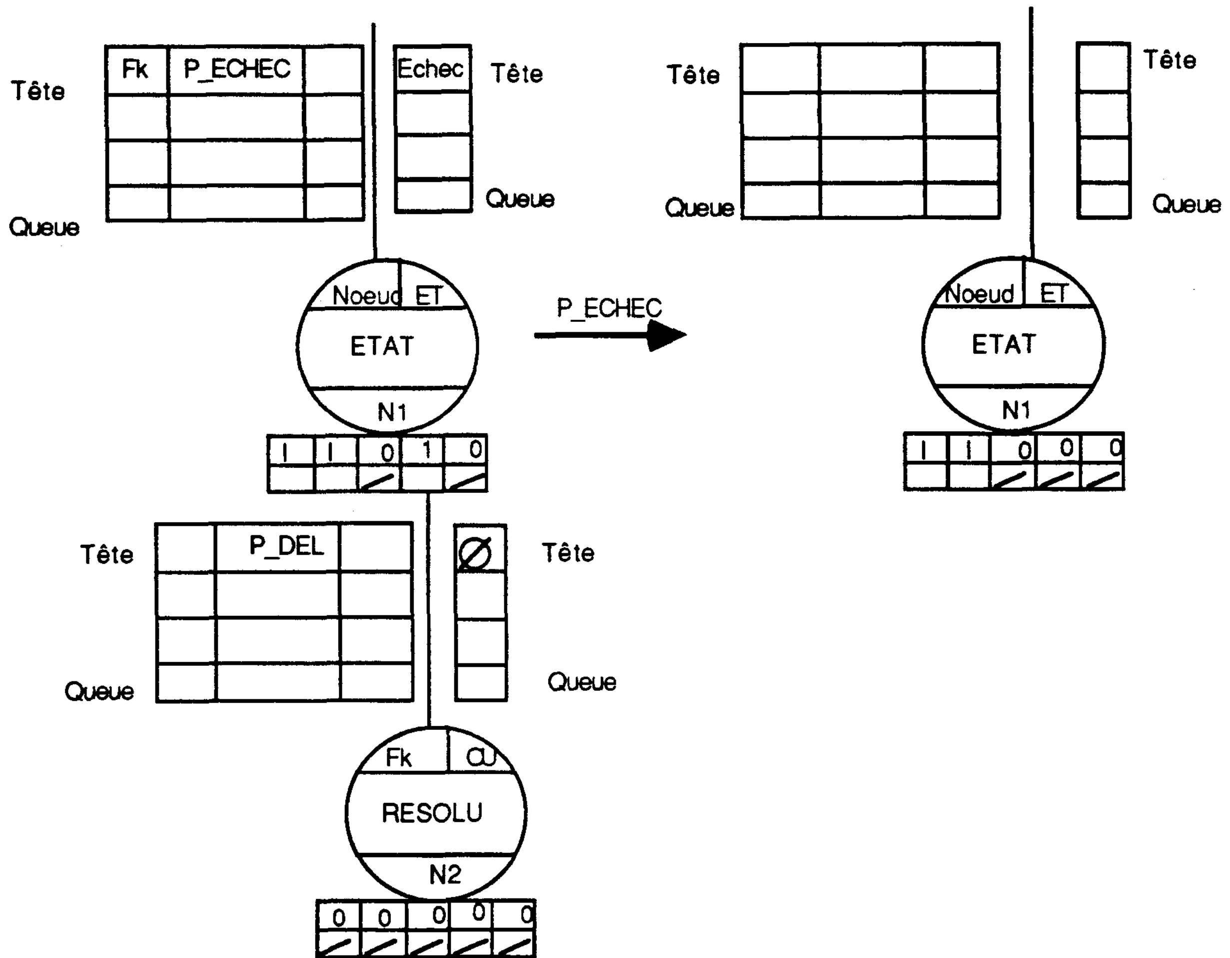


Figure 5.4

VII. LA PRIMITIVE P_DEL.

La primitive P_DEL est créée par le nœud lui-même lorsque l'état de ce dernier est résolu (l'arbre dont il est la racine est détruit). Elle entraîne la destruction du nœud (cf figure 6).

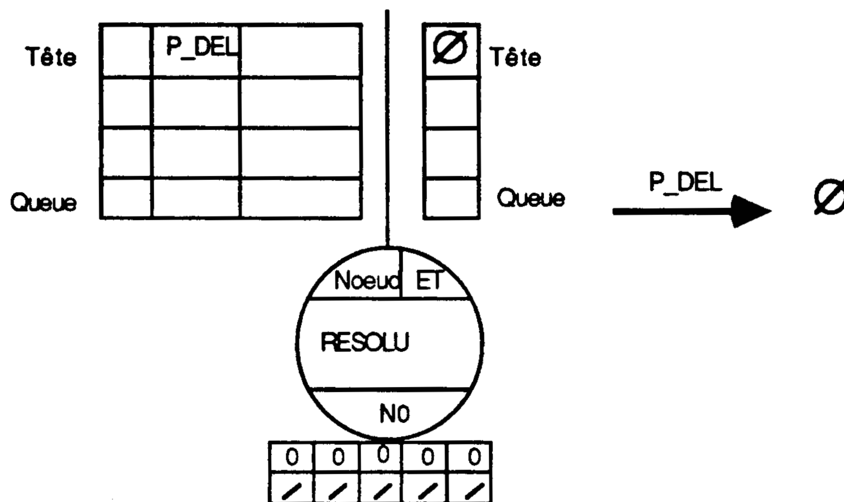


Figure 6

VIII. CONCLUSION.

Le gérant d'un nœud ET apparaît donc plus complexe que celui d'un nœud OU. Ceci est du à la gestion du pipeline par un nœud ET qui entraîne le dialogue simultané entre le nœud ET et plusieurs de ses fils (dialogue nécessaire à l'alimentation continu du pipeline).

Cette gestion d'un pipeline par un nœud ET a d'ailleurs une conséquence sur le nombre maximum de primitives contenues dans le buffer de primitives du nœud. Ce nombre vaut (n/2+1) où n est le nombre de fils du nœud. En effet, deux fils consécutifs d'un nœud ET ne peuvent simultanément retourner une solution à ce nœud ET (l'un au moins de ses fils est activé en mode restreint). La primitive supplémentaire est émise par le nœud père.

De manière générale, la description détaillée des gérants des noeuds ET et des nœuds OU suscitent les commentaires suivants:

- l'ordre de consommation, par un nœud B, des primitives émises par un nœud A, respecte l'ordre d'émission de ces primitives par le nœud A.

Ceci s'explique par le mode de contrôle adopté, qui est le contrôle dirigé par la nécessité.

- la seule initiative réelle d'un nœud est sa destruction. Le nœud a préalablement d'une part vérifié l'inactivité de l'ensemble de ses fils, d'autre part mis un terme au dialogue avec son père en émettant vers ce dernier, soit une primitive résultat dont l'attribut dernier est vrai, soit une primitive d'échec.

LES PREDICATS A EFFET DE BORD

LES PREDICATS A EFFET DE BORD.

I. INTRODUCTION.

Dans les chapitres précédents, nous avons présenté un modèle d'évaluation parallèle de Prolog, dont l'objectif principal est le respect de la sémantique opérationnelle de Prolog. Des programmes Prolog "classiques" doivent donc pouvoir être exécutés selon ce modèle d'évaluation sans avoir subi de modifications. Ce modèle d'évaluation ne serait pas complet s'il n'intégrait certains prédicats, dits à effets de bords. Ces derniers permettent l'introduction de mécanismes de contrôle, la réalisation des entrées/sorties...

Ce chapitre aborde la mise en œuvre de quatre de ces prédicats dans le modèle, en l'occurrence les prédicats FAIL, CUT et ceux d'entrée-sortie. Certains de ces prédicats étant susceptibles de modifier l'aspect de l'arbre, nous montrerons comment les manipuler sans perturber le bon fonctionnement du modèle.

II. LE PREDICAT FAIL.

II.1. Rôle du prédicat FAIL.

Le prédicat FAIL est un prédicat prédéfini qui n'est jamais satisfait. Il entraîne donc un échec systématique sans toutefois affecter le retour arrière. Il apparaît toujours en position terminale.

II.2. Implantation en terme de primitives.

Le prédicat FAIL est représenté par un noeud OU, qui n'a pas de fils et qui est toujours le dernier fils d'un noeud ET (figure 1).

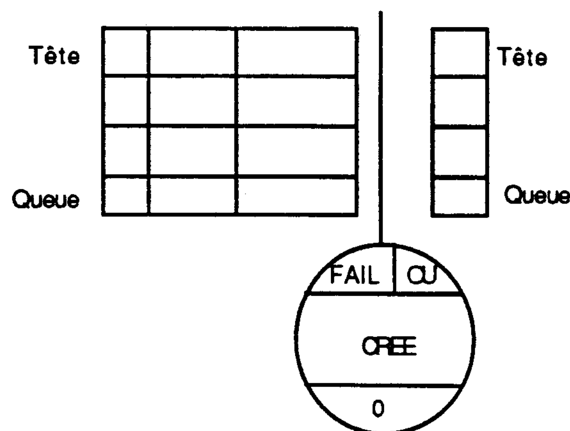


Figure 1 : Le noeud FAIL.

Le noeud père dépose une primitive P_ACT dans le buffer de primitives du noeud FAIL. Ce dernier n'a pas de noeud fils. Aussi, il émet deux primitives : une primitive d'échec à l'intention de son noeud père et une primitive de destruction à sa propre intention. Il dépose ces primitives respectivement dans le buffer de primitives de son noeud père et le sien. Le noeud père traite cette primitive d'échec comme nous l'avons exposé dans les chapitres précédents (figure 2).

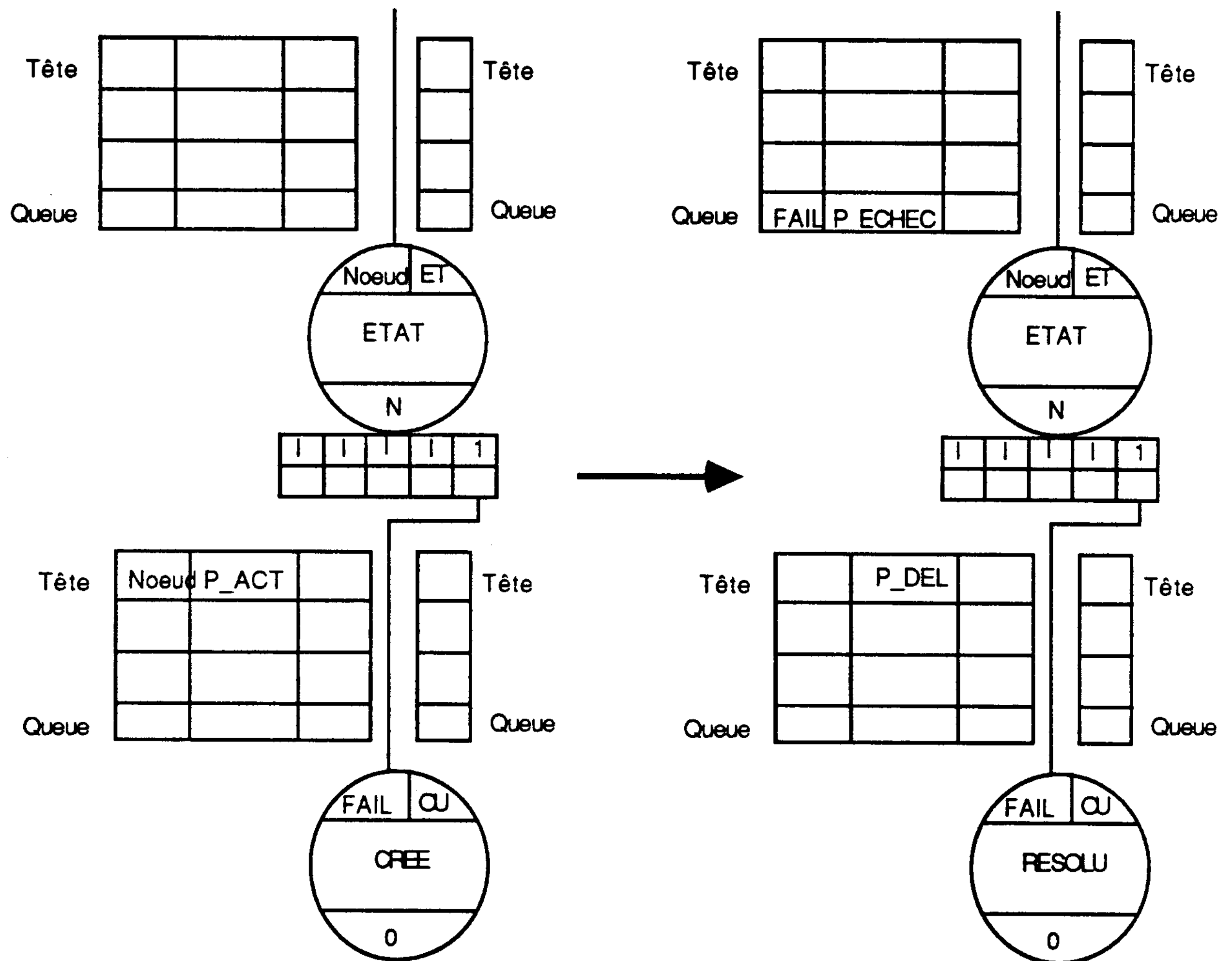


Figure 2 : Comportement du noeud FAIL.

III. LE PREDICAT CUT.

III.1. Rôle du prédicat CUT.

Le CUT, ou coupe-choix, contrôle l'expansion de l'arbre par suppression de points de choix. Condillac [Cond 86] en donne la définition suivante : " l'exécution d'un CUT contenu dans le corps d'une clause, entraîne la suppression de tous les choix en attente pour tous les buts à

partir de celui qui a activé la règle contenant le CUT, et jusqu'à celui qui précède le CUT dans le corps de cette règle ".

Boizumault [Boiz 89] définit cette notion en terme de noeuds : " le CUT supprime tous les noeuds OU, construits jusque-là, figurant dans le sous-arbre de racine son but père, y compris ce dernier s'il est lui-même un noeud OU. Le CUT rend donc déterministe le début de preuve courante de son but père".

III.2. Le CUT face au parallélisme du modèle: solution retenue.

Les définitions précédentes font apparaître qu'une fois le CUT "franchi", un retour arrière sur les prédicats qui précèdent le CUT dans le corps de la clause est impossible. De même, toute évaluation des alternatives du paquet non encore envisagées est interdite.

Considérons l'exemple suivant (figure 3) :

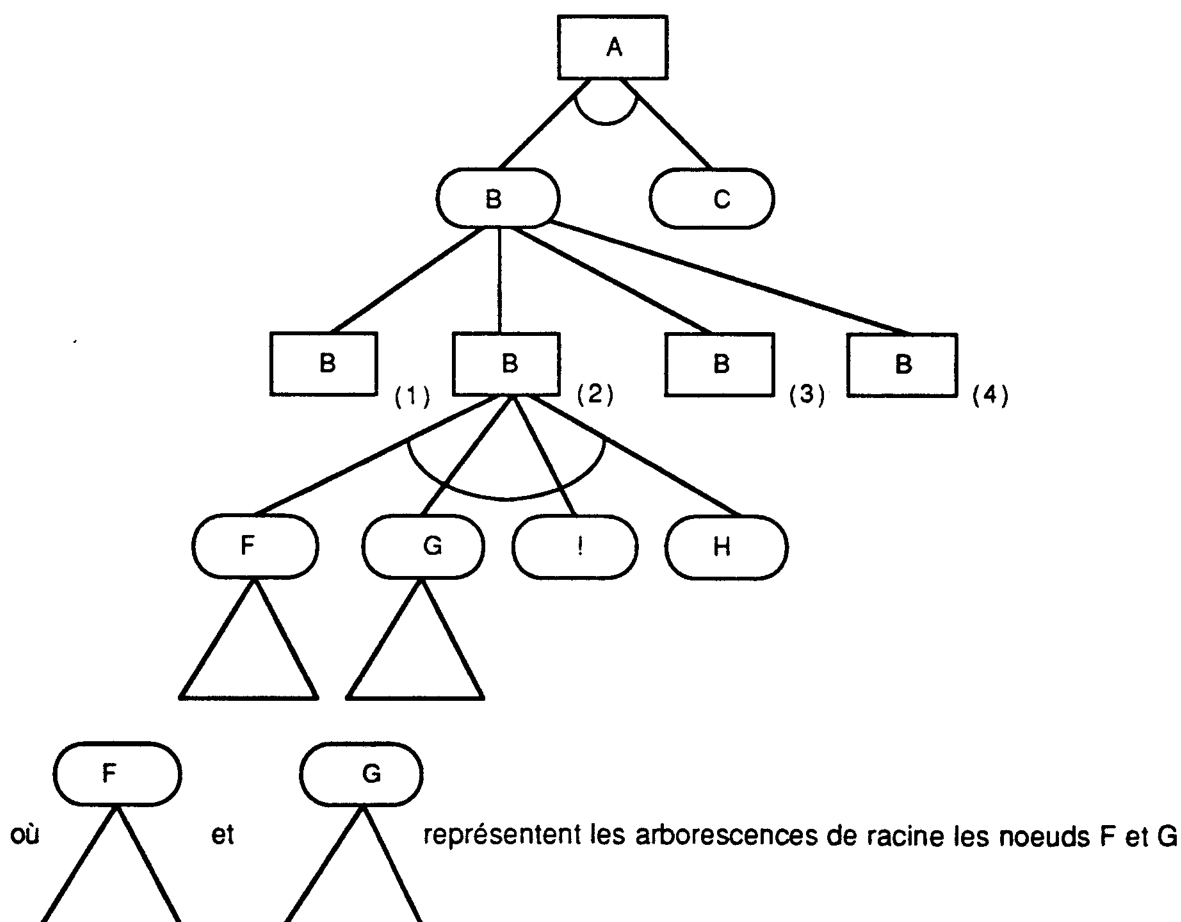


Figure 3 : Effets du CUT.

Le prédicat B est défini par quatre alternatives, dont le corps de l'une d'entre elles (en l'occurrence la seconde) contient un CUT. Si le CUT est satisfait, il faut :

- détruire les arborescences ayant comme racine les noeuds F et G;
- continuer la résolution normale de H ;
- empêcher l'activation des noeuds B(3) et B(4).

La gestion du CUT entraîne la mise en place d'un mécanisme supplémentaire, qui consiste en la destruction d'une arborescence. Dans un premier temps, nous décrirons le principe de cette destruction. Puis, nous introduirons les primitives nouvelles, nécessaires à la gestion du CUT. Nous terminerons en décrivant les modifications apportées aux gérants des noeuds.

.III.2.a. La destruction d'une arborescence : principe général.

Le problème consiste à détruire une arborescence dynamique : en effet, des noeuds sont créés et détruits sur échange de primitives. Le principe de cette destruction s'appuie sur les trois points suivants :

*) La racine de l'arborescence a l'initiative de la destruction : elle communique à ses descendants l'ordre de se détruire. Chacun de ses fils est lui-même racine d'une arborescence (cela tient à la définition récursive d'une arborescence) qu'il faut aussi détruire : en conséquence, ces noeuds répercutent sur leurs propres fils l'ordre de destruction, qui est ainsi propagé jusqu'aux feuilles de l'arbre.

*) Un noeud qui a reçu un ordre de destruction n'a plus besoin de poursuivre son évaluation : ses calculs sont devenus inutiles. Il doit suspendre toute activité. Or cette dernière est consécutive à l'interprétation des primitives qu'il reçoit (va recevoir) ou détient déjà dans son buffer de primitives. Aussi, ignore-t-il toute primitive de résultat reçue après la prise en compte de l'ordre de destruction.

*) Un nœud qui reçoit et transmet à ses fils l'ordre de destruction ne peut se détruire immédiatement. En effet, pendant l'intervalle de temps séparant l'émission de l'ordre de destruction par le nœud et la réception de cet ordre par ses fils, ces derniers ont continué à élaborer des solutions. Certains d'entre eux (ceux dont le verrou de solutions est levé) ont pu même émettre des primitives de type résultat à l'intention du nœud. Une destruction prématurée du nœud priverait ces primitives de destinataire.

Aussi, la destruction du nœud n'est effective qu'au moment où le

nœud est assuré de la bonne réception de l'ordre de destruction par ses nœuds fils. A cette fin, chaque nœud fils concerné émet un accusé de réception de l'ordre. Ces accusés de réceptions sont les seules primitives prises en compte par un nœud prêt à se suicider. En effet, un nœud ne dialogue qu'avec son père et ses fils. Or l'ordre de destruction est la dernière primitive émise par un nœud père vers un de ses fils. De plus, l'accusé de réception est la dernière primitive émise par un nœud vers son père. Si un nœud A émet consécutivement deux primitives vers un nœud B, ce dernier reçoit et par conséquent interprète les deux primitives selon leur ordre d'émission. L'accusé de réception met donc un terme au dialogue fils-père.

Tel quel, le mécanisme reste insuffisant, car il peut d'une part entraîner la perte d'accusés de réception, et d'autre part empêcher la destruction de nœuds. Considérons l'exemple suivant. Un nœud A intime à l'un de ses fils l'ordre de se détruire. Or, ce fils a déjà pris l'initiative de son auto-destruction, suite à l'émission, de sa part, d'une primitive, dite primitive "fin de travail" (c'est-à-dire soit une primitive échec, soit une primitive résultat, dont l'attribut dernier est vrai), que le nœud A n'a pas encore interprété cette primitive. Le nœud A attend alors un accusé de réception de la part du nœud fils, qui n'existant plus, ne peut l'envoyer. Le nœud A ne pourra, faute de réception de cet accusé, se détruire.

Une première solution consiste à ne plus laisser aux nœuds de leur destruction, mais de la confier aux nœuds pères. L'interprétation d'une primitive "fin de travail" par un nœud père entraînera donc l'émission systématique d'une autorisation de destruction vers le nœud fils concerné. Ce n'est que sur réception d'une telle autorisation que le nœud fils se détruira effectivement.

Une seconde solution associe à l'émission d'une primitive de destruction, un mécanisme de validation du destinataire. Ce mécanisme implique l'apparition d'un nouveau type de nœud, noté nœud puits. Ce nœud prend en charge la réception des primitives de destruction, dont les destinataires ont déjà disparu et l'émission de l'accusé de réception correspondant.

III.2.b. Les primitives supplémentaires nécessaires à la gestion du CUT.

Remarque : Le prédicat CUT est représenté par un nœud OU, qui n'a pas de fils (figure 4).

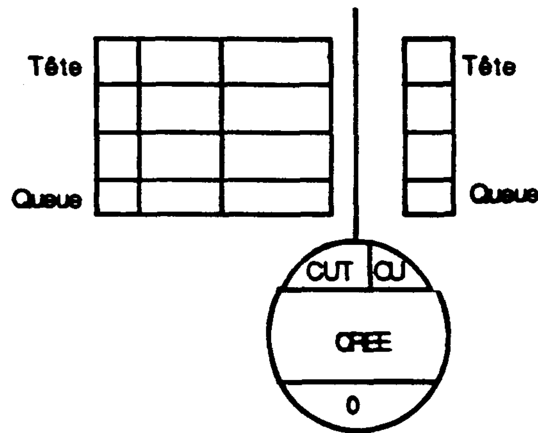


Figure 4.

La mise en oeuvre du CUT implique la gestion des trois primitives suivantes P_CUT, P_KILL, ET P_ACK_KILL.

*) La primitive P_CUT s' écrit P_CUT(Id_Emetteur, Id_destinataire). Un noeud fils, désigné par Id_Emetteur, prévient son père, Id_Destinataire, de la présence d'un noeud CUT. Le nœud Id_Emetteur est soit le nœud CUT lui-même, soit le père du nœud CUT. Le nœud destinataire est en conséquence soit le père du nœud CUT, soit le grand-père du nœud CUT. La primitive P_CUT est une primitive résultat particulière.

*) Par la primitive P_KILL(Id_Emetteur, Id_destinataire), le noeud Id_Destinataire reçoit de son père, Id_Emetteur, l'ordre de se détruire.

*) La primitive P_ACK_KILL(Id_Emetteur, Id_Destinataire) constitue l'accusé de réception de l'ordre de destruction précédemment reçu. Il est émis par un noeud fils, Id_Emetteur, à l'intention d'un noeud père, Id_Destinataire.

III.2.c. Les états supplémentaires nécessaires à la gestion du CUT.

L'introduction des primitives précédentes implique la gestion des états suivants : Suicide, Attente_Ack_Kill, Attente_Ack_Kill_Double, Attente_Cut_Solution, Attente_Cut_Double, Attente_Cut_Levée_Verrou.

- **Suicide** est l'état du nœud qui vient de recevoir l'ordre de se tuer. Dans cet état, un nœud n'interprète aucune primitive en dehors des accusés de réception de l'ordre de destruction (en l'occurrence la primitive P_ACK_KILL).

- **Attente_Ack_Kill** et **Attente_Ack_Kill_Double** sont des états associés au nœud ET, père du nœud CUT. Ce nœud ET attend les accusés de réception des nœuds prédécesseurs du nœud CUT avant de reprendre

le traitement initial. Le premier état est consécutif à une activation "normale" du nœud ET (l'activation a été effectuée à l'aide d'une primitive P_ACT), le second état à une activation en mode restreint (la primitive d'activation utilisée est une primitive P_LOCK). Dans l'un ou l'autre de ces états, les primitives de type résultat ou de type échec sont ignorées.

- **Attente_Cut_Solution** est un état associé au seul nœud OU, grand-père du nœud CUT. Le nœud a l'autorisation de retourner à son propre nœud père la solution qu'il vient de recevoir.

- **Attente_Cut_Double** est un état associé soit au nœud OU, grand-père du nœud CUT, soit au nœud ET père du CUT. Dans cet état, le nœud attend les solutions évaluées par le nœud fils en cours d'évaluation.Le nœud mémorise cette solution jusqu'à obtenir une demande de solution émanant de son nœud père.

- **Attente_Cut_Levée_Verrou** est un état associé soit au nœud OU, grand_père du nœud CUT, soit au nœud ET père du CUT. Lorsqu'ils disposent d'une solution, les nœuds concernés attendent la demande explicite de solutions émanant de leur père respectif, qui leur permettra de retourner cette solution.

Gérer le CUT implique non seulement l'introduction de nouvelles primitives, mais encore la modification de plusieurs des primitives décrites dans les chapitres précédents. Nous présenterons d'abord les primitives P_KILL et P_ACK_KILL, que les gérants des noeuds ET et OU interprètent de la même façon. Nous décrirons ensuite l'interprétation de la primitive P_CUT spécifique à chaque gérant. Nous étudierons enfin les modifications apportées à l'interprétation des primitives P_UNLOCK, P_RES, P_ECHEC, mises en oeuvre dans les chapitres précédents, en nous limitant dans un premier temps à la gestion du CUT par un nœud ET activé en mode "normal". Nous terminerons en décrivant les modifications apportées aux gérant d'un nœud ET, modifications rendues nécessaires par la prise en compte d'un nœud CUT par un nœud ET activé en mode restreint.

III.2.d. La primitive P_KILL(Nœud Père, Noeud).

Le noeud, de type ET ou OU, reçoit de son père l'ordre de se détruire. L'état du noeud est indifférent. Il doit alors :

*) Propager l'information à ses descendants
*) Avertir son père de la prise en compte de l'ordre
*) Attendre les accusés de réception de ses descendants afin de pouvoir réaliser sa propre destruction.

A cette fin, il dépose dans le buffer de primitives de son père un accusé de réception. Il parcourt sa table de liens afin de déterminer les noeuds fils encore présents à qu'il propage l'ordre de destruction. Il passe alors en l'état Suicide, dans lequel il n'interprète aucune primitive, sauf les accusés de réception de ses descendants. Toute primitive déjà reçue, donc mémorisée dans le buffer de primitives, ou non sera ignorée (figure 5).

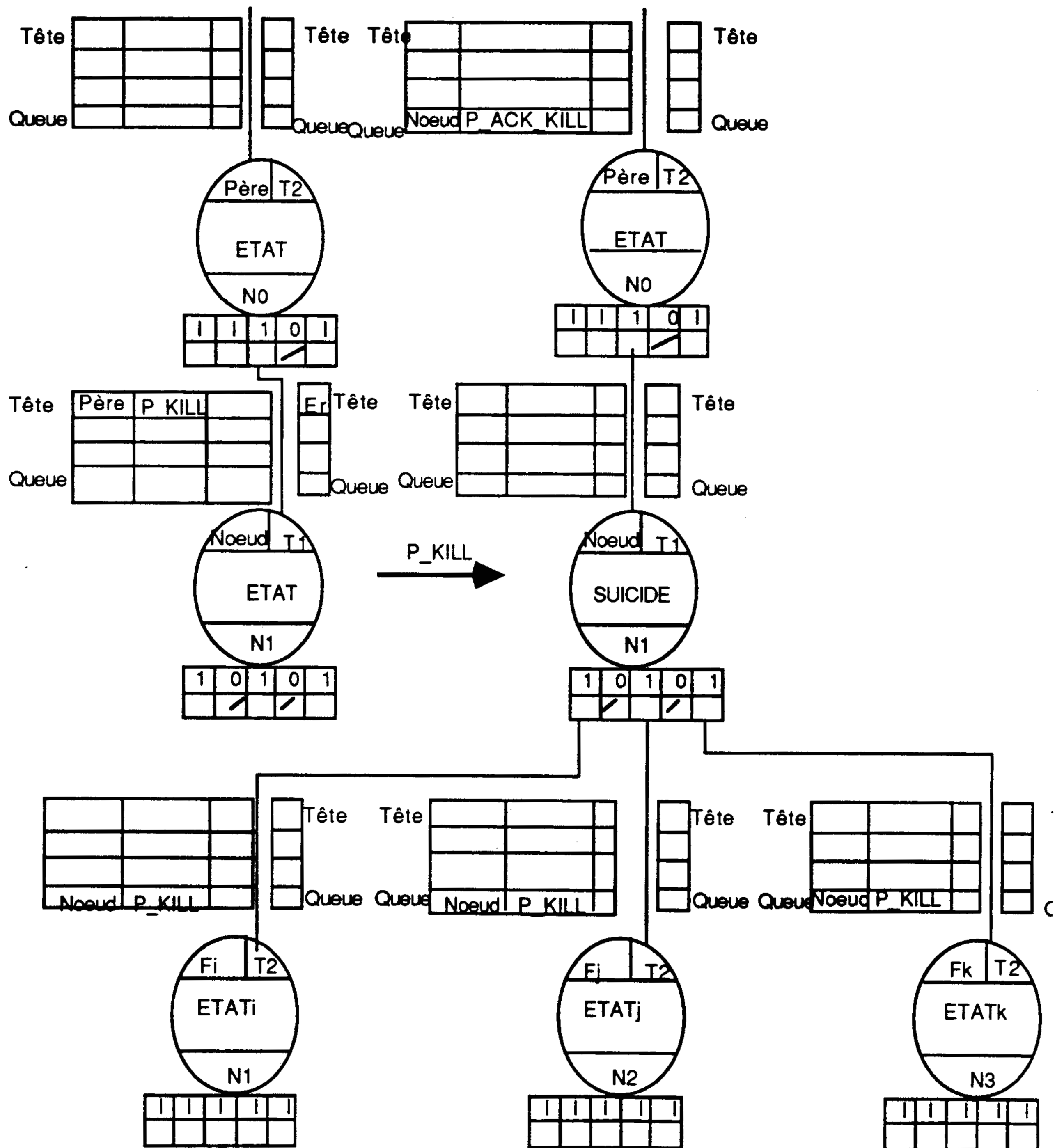


Figure 5.

III.2.e. La primitive P_ACK_KILL(F_k, Noeud).

Un nœud qui reçoit une telle primitive est :

*) soit un noeud qui est dans l'état Suicide : il attend de ses fils les accusés de réception qui lui permettront de rendre effective sa destruction.

*) soit le père du noeud du CUT : il attend l'accusé de réception des prédécesseurs du CUT, avant de reprendre le traitement initial.

Nous allons successivement étudier ces deux situations.

III.2.e.1. L'état du nœud est Suicide.

Le noeud (qu'il soit de type ET ou OU) détruit le lien qui l'unissait au noeud F_k, et remet à zéro le bit d'existence correspondant. Il consulte alors sa table de liens.

- Situation 1: tous les bits d'existence de la table de liens sont à zéro, tous les liens sont détruits. Le noeud peut se détruire. Il dépose donc dans son buffer de primitives une primitive de destruction et passe dans l'état résolu (figure 6).

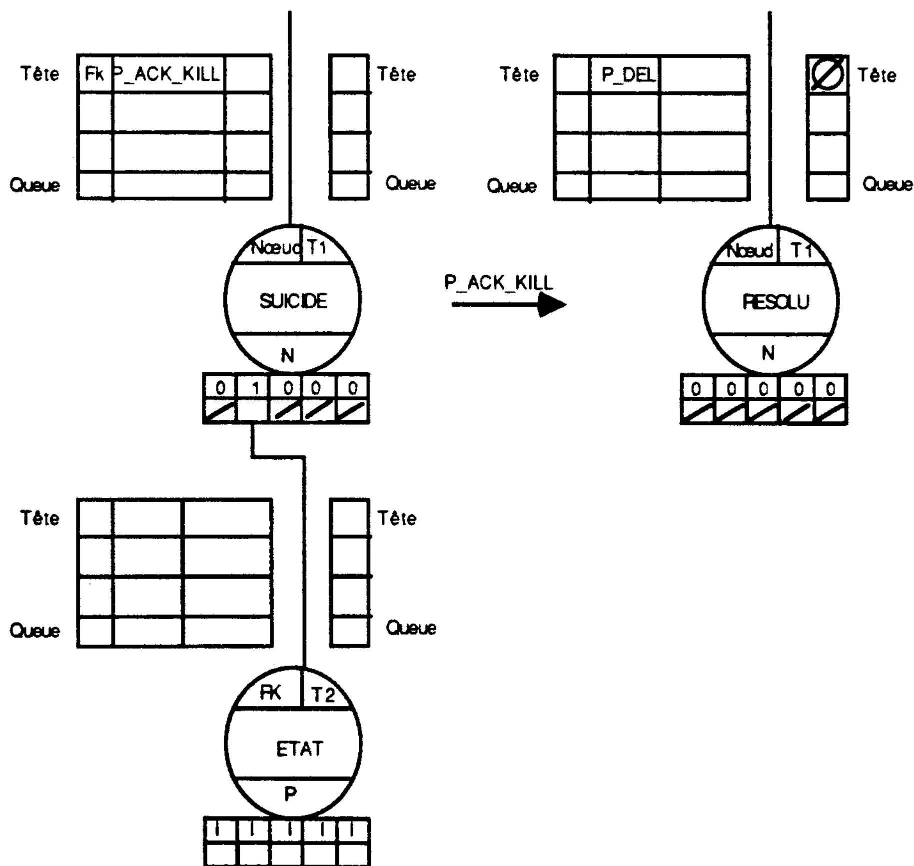


Figure 6.

- Situation 2 : Il reste au moins un lien dans la table de liens. Un noeud fils, au moins, n'a pas renvoyé son accusé de réception. L'état du noeud demeure inchangé (figure 7).

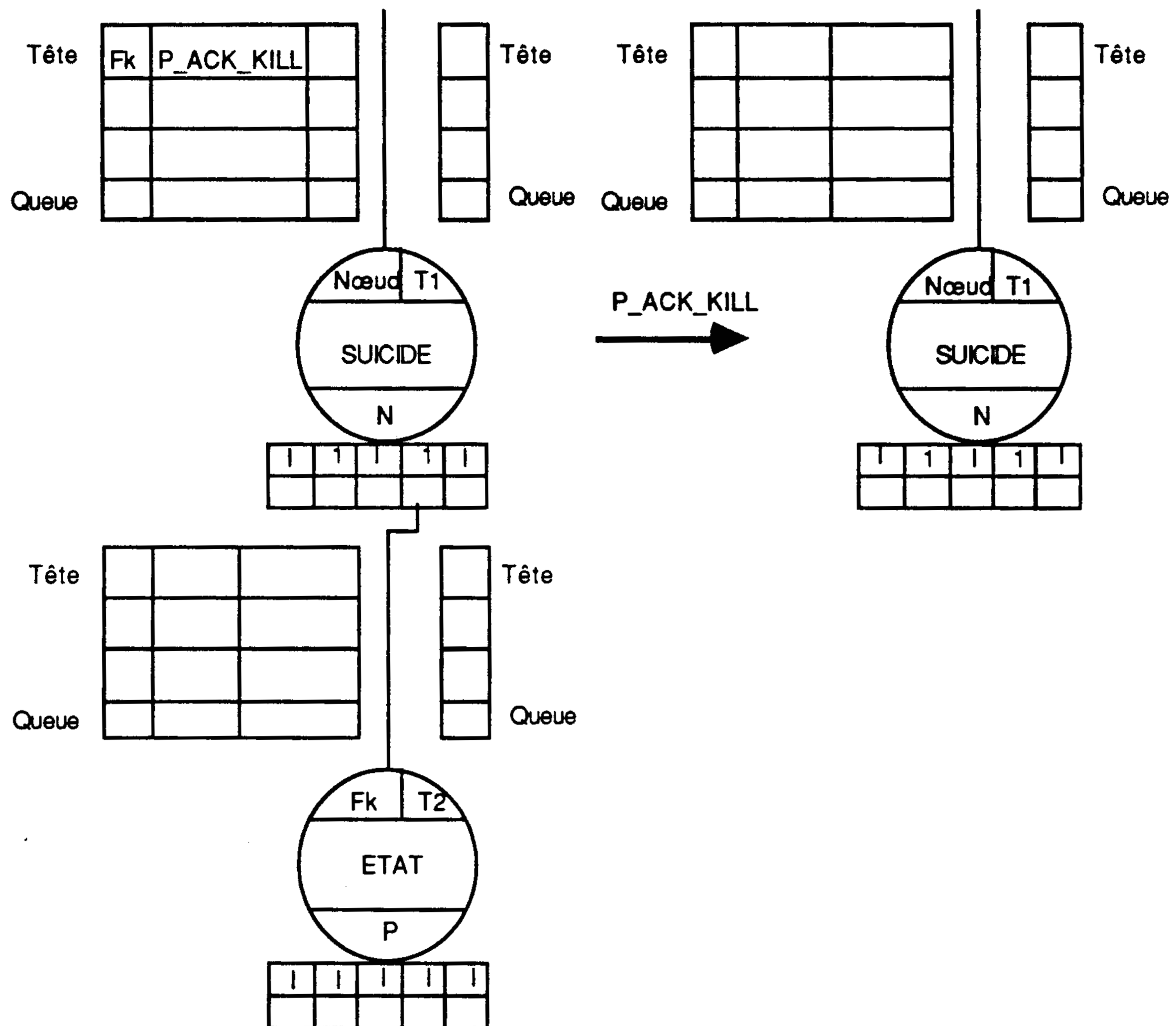


Figure 7.

III.2.e.2. Le noeud est le noeud père du CUT.

Comme précédemment, le noeud détruit le lien qui l'unissait au noeud fils Fk et remet à zéro le bit d'indice k correspondant. Il teste alors sa table de liens et agit en fonction de son état courant. Cet état courant est Attente_Ack_Kill. Il a donc été activé par une primitive P_ACT.

Cas 1: Tous les liens des nœuds d'indice inférieur à celui du CUT sont détruits. Le nœud s'assure de l'existence (voire de la non-existence) d'un nœud fils, successeur droit du CUT.

Cas 1.a). Le nœud CUT a un successeur : le nœud l'active et se met en attente de solution (figure 8).

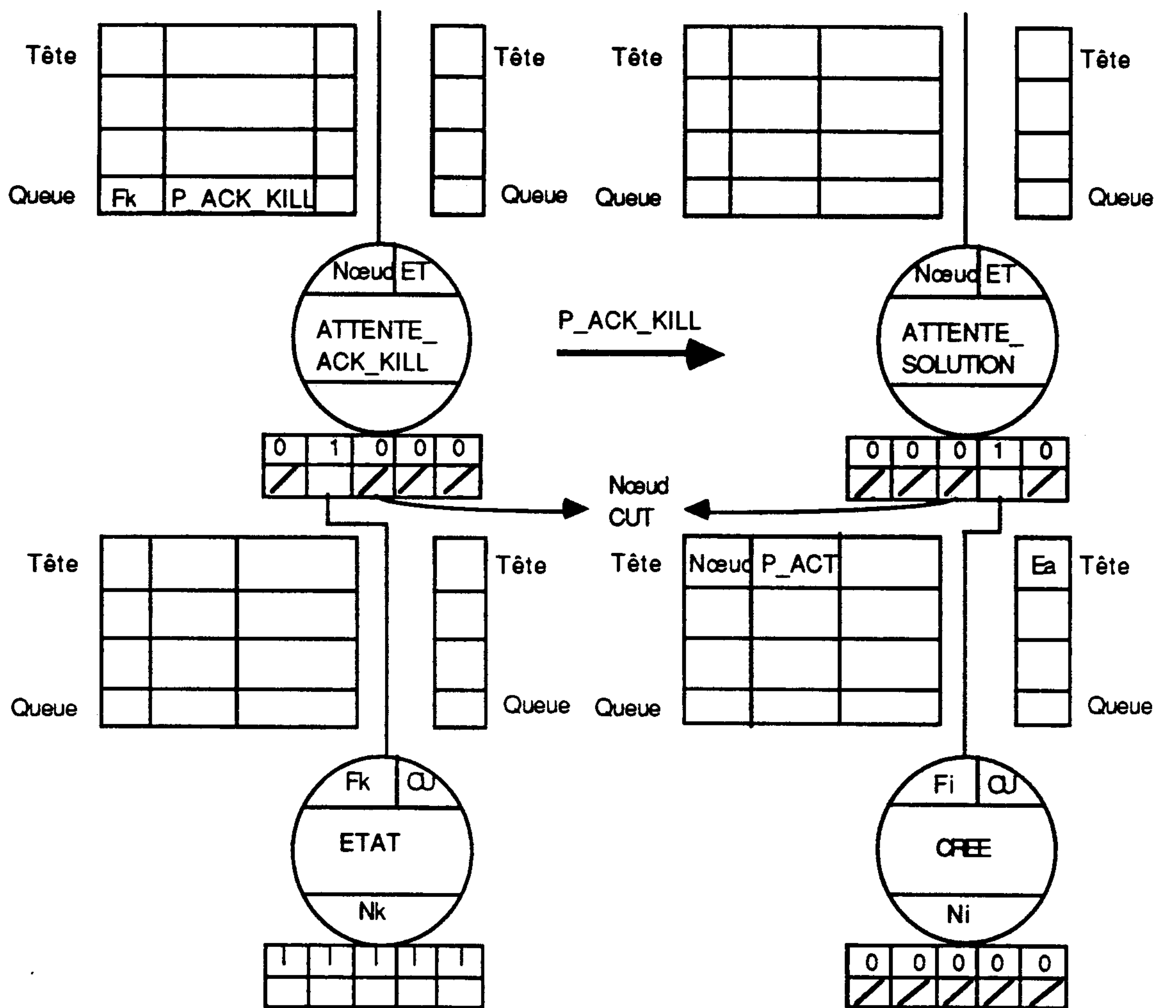


Figure 8.

Cas 1.b) Le CUT n'a pas de successeur et est donc le dernier fils du nœud. Ce dernier a évalué une solution, qui est unique par ailleurs. Le nœud a par conséquent mémorisé un environnement-solution dans son buffer d'environnements. Il construit donc une primitive de résultat dont les attributs premier et dernier ont la valeur vrai, qu'il dépose dans le buffer de primitives de son nœud père. Il commence la procédure d'auto-destruction, et passe dans l'état résolu (figure 9).

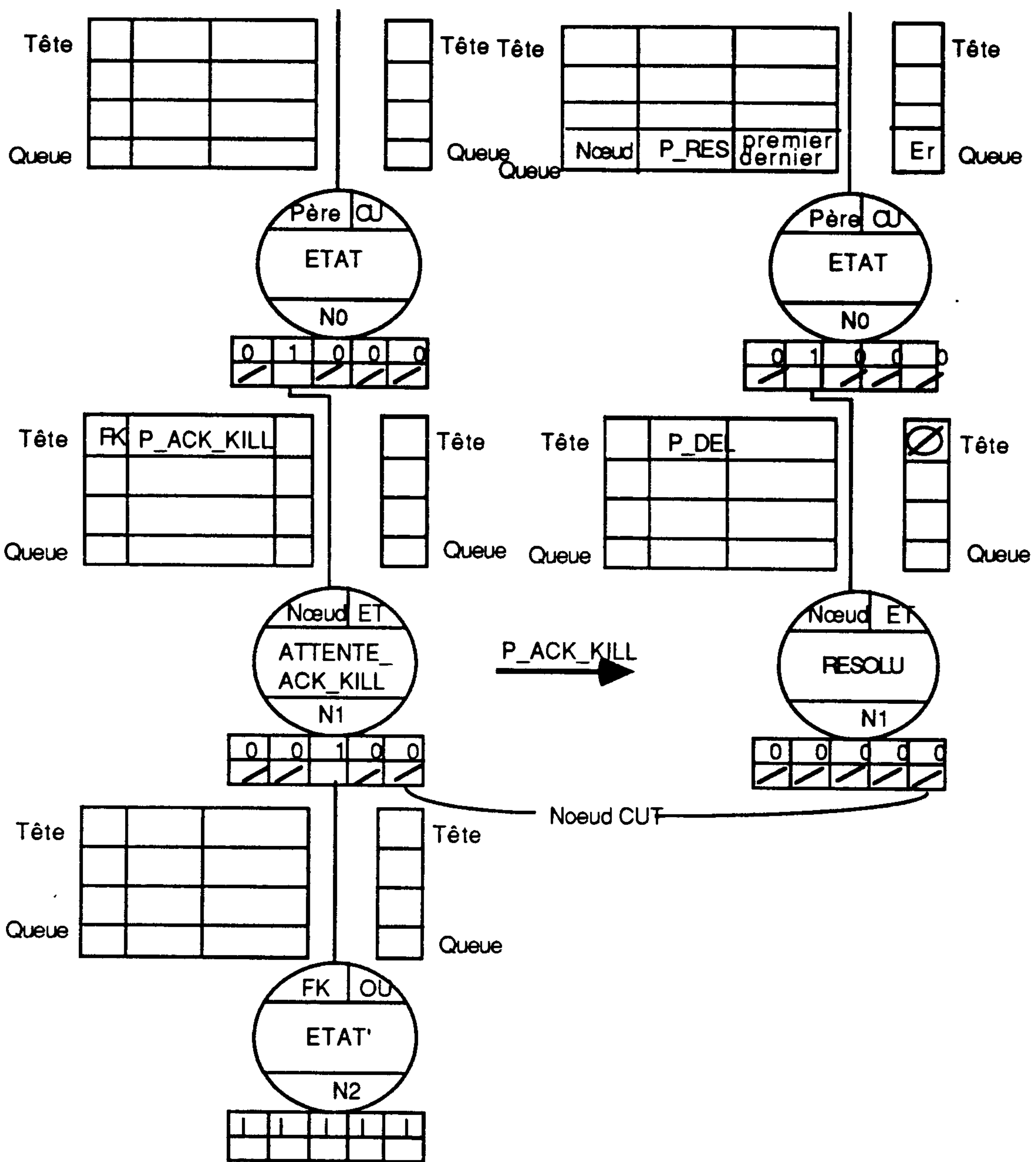


Figure 9.

Cas 2 : Il existe encore un lien vers un noeud fils d'indice inférieur à celui du CUT. Le noeud ne fait aucune action en dehors de la modification de la table de liens. L'état du noeud reste inchangé (figure 10).

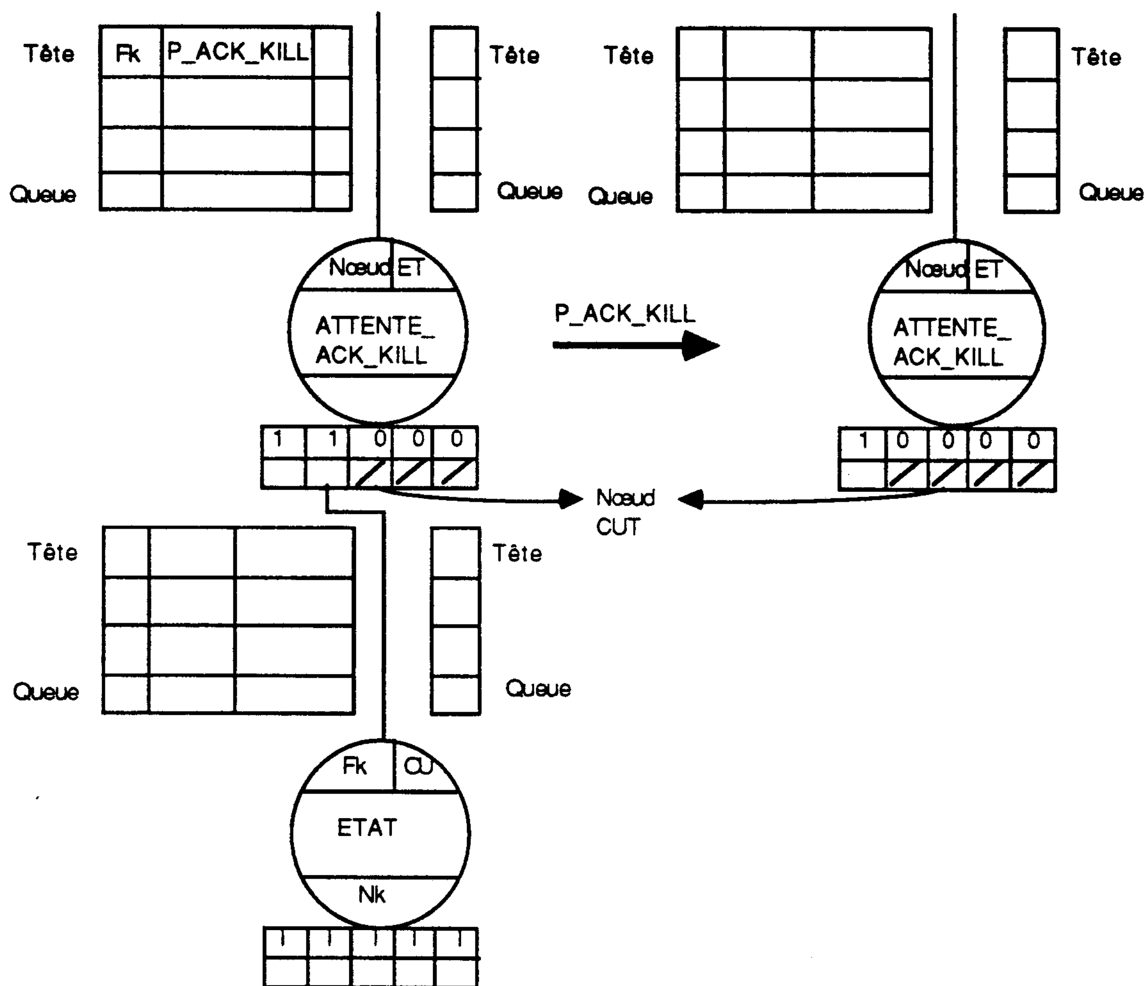


Figure 10.

III.2.f. La primitive P CUT(F_k, Nœud).

a) Le nœud qui interprète la primitive P_CUT est de type ET.

Le nœud fils F_K est donc le nœud CUT lui-même. Un nœud ET qui reçoit une telle primitive est en pleine évaluation de sa première solution. En conséquence, il vient d'être activé soit par une primitive P_ACT(selon nos hypothèses de présentation).

Le nœud doit alors d'une part avertir son propre nœud père de la présence d'un CUT, d'autre part envoyer un ordre de destruction à chacun des éventuels nœuds fils qui précédent le nœud CUT. Cette

dernière opération est fonction de la position du nœud CUT parmi l'ensemble des nœuds fils.

Cas 1 : Le CUT est l'unique fils du nœud. Le nœud a donc évalué une solution unique. Il émet alors consécutivement deux primitives, en l'occurrence une primitive P_CUT et une primitive de résultats dont les attributs premier et dernier sont tous deux positionnés. Ces deux primitives sont déposées dans le buffer de primitives du nœud père. La primitive P_CUT précède la primitive de résultat. Le nœud peut alors être détruit : il passe en état résolu et déclenche son auto-destruction (figure 11).

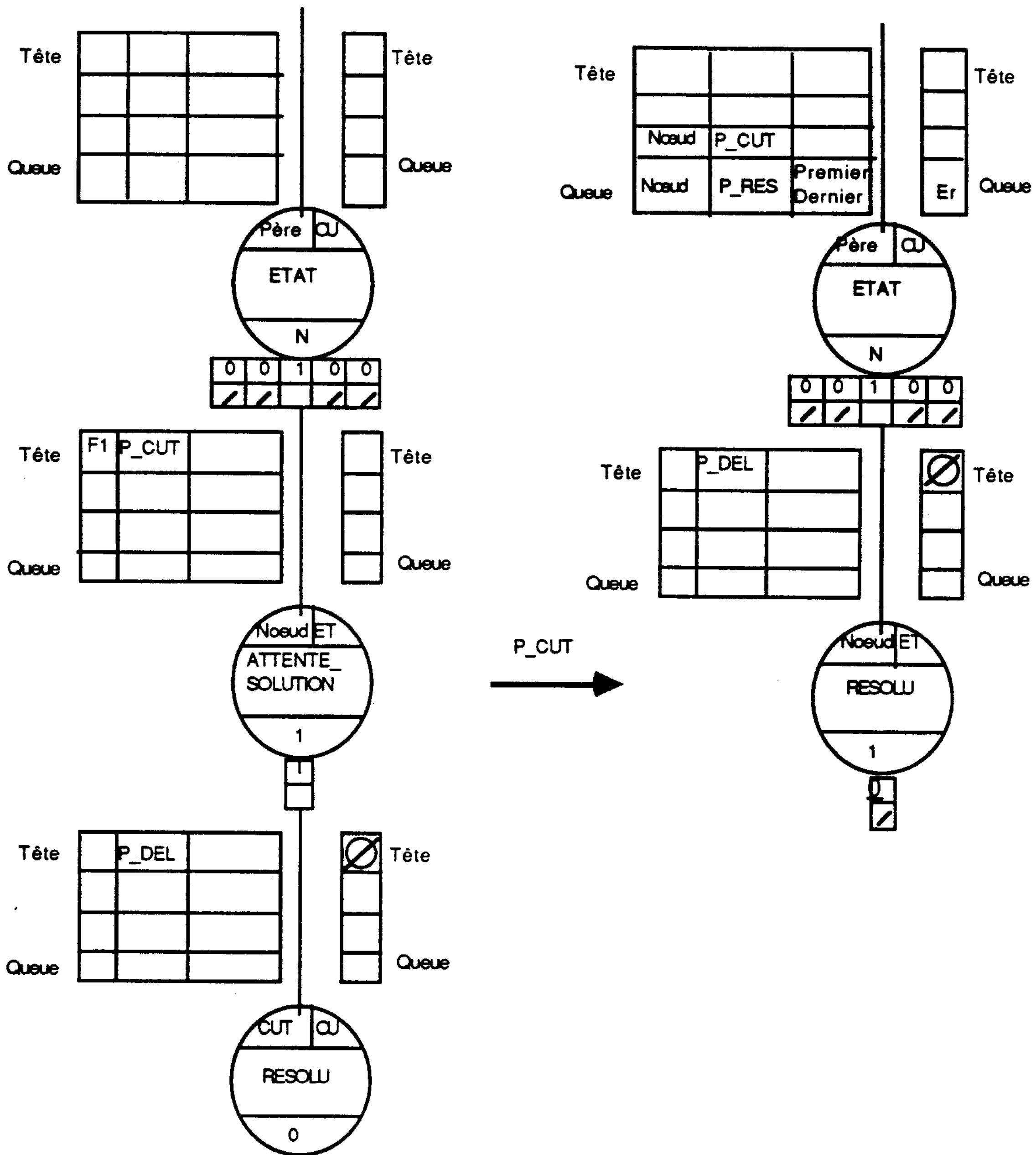


Figure 11

Cas 2. Le CUT est le premier fils du nœud, mais le nœud a d'autres fils. Il active le nœud fils, successeur du CUT en déposant dans le buffer de primitives de ce dernier une primitive P_ACT (figure 12).

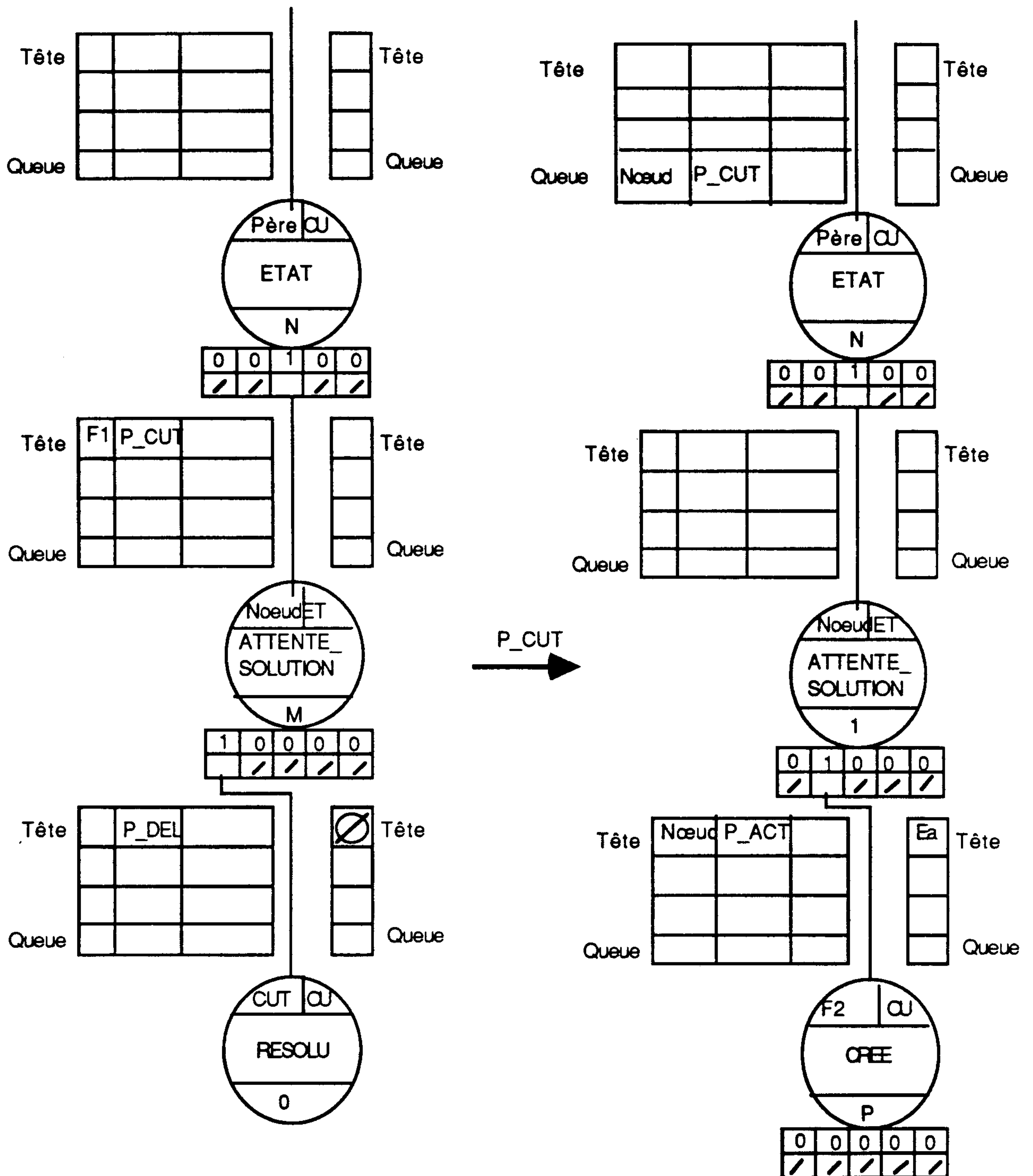


Figure 12.

Cas 3. Le nœud CUT n'est pas le premier fils du nœud. En conséquence, le nœud doit lancer la destruction des prédécesseurs du nœud CUT. Il consulte sa table de liens afin de déterminer les nœuds fils, d'indice inférieur à k, qui sont encore présents dans la table. Il émet à l'intention de chacun de ces nœuds l'ordre de destruction, en déposant la primitive P_KILL dans le buffer de primitives de chacun d'entre eux.

Enfin, il change d'état : d'Attente_Solution, il passe à Attente_Ack_Kill (figure 13).

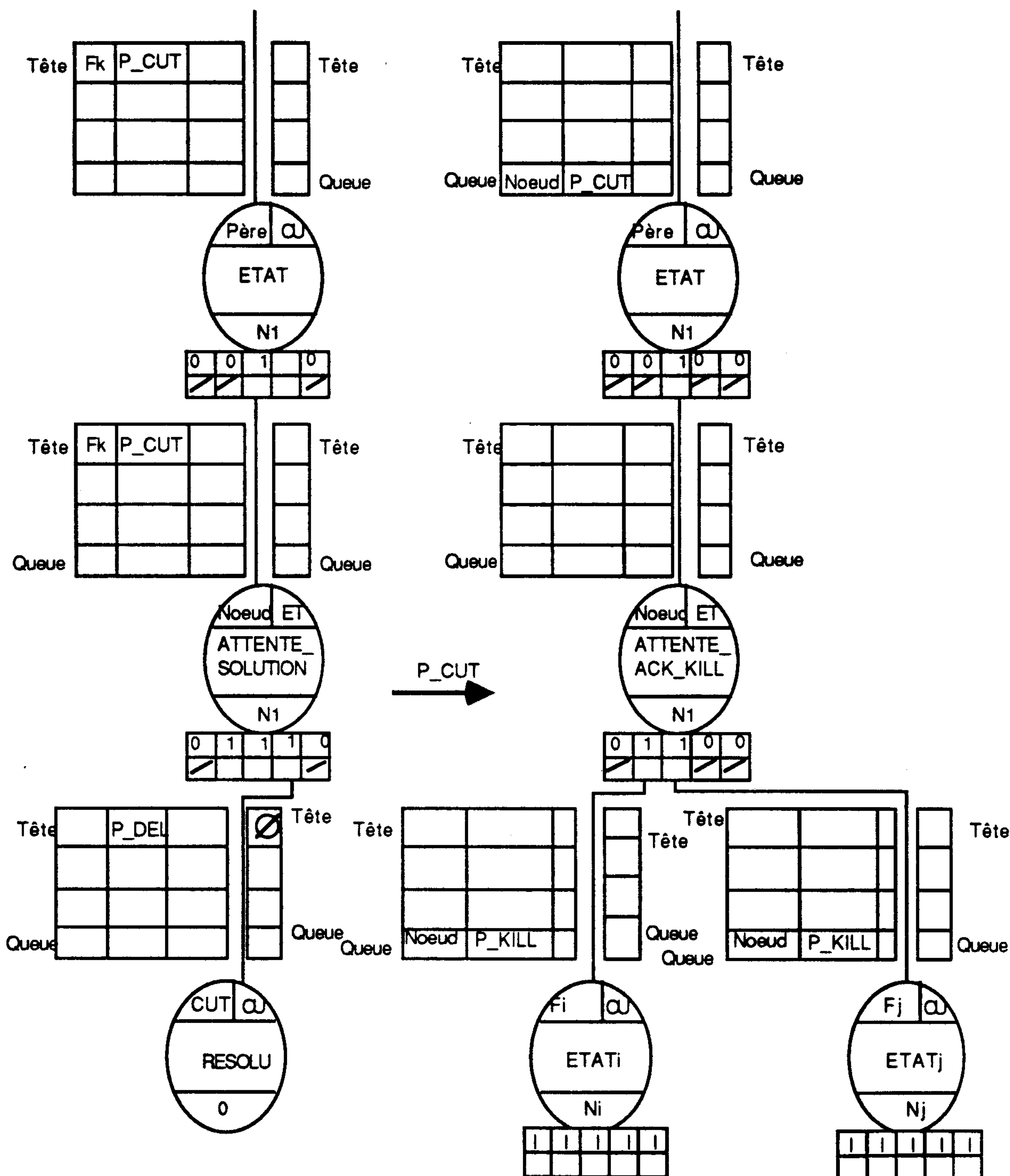


Figure 13.

b) Le nœud qui interprète la primitive est de type OU. Le nœud est donc le "grand-père" du nœud CUT. Il doit donc empêcher toute activation des nœuds frères successeurs du nœud père du CUT.

Les études précédentes sur le modèle d'évaluation, font ressortir les quatre points suivants :

- Un nœud OU peut dialoguer avec au plus deux de ses fils : l'un est activé en mode normal, l'autre en mode restreint.
- Un nœud OU n'active en mode restreint l'un de ses fils que sur retour de la première solution du fils précédent.
- La détection d'un CUT par un nœud ET se fait avant l'évaluation d'une solution complète par ce nœud.
- Le buffer de primitive est géré comme une file : donc, les primitives mémorisées dans ce buffer sont interprétées selon leur ordre d'apparition.

Ces constatations entraînent les conséquences suivantes :

- Une primitive P_CUT émise par le nœud père du nœud CUT à l'intention de son propre nœud père précède toute primitive de résultat.
- Le nœud frère, successeur immédiat du nœud père du CUT est encore inactif.

La seule action que le nœud doit donc effectuer consiste à terminer l'échange avec le nœud ET, père du CUT. Ce dialogue prend fin sur la réception soit d'une primitive d'échec, soit d'une primitive de résultat dont l'attribut dernier est vrai. Tout dépend alors de l'état courant du nœud, qui attend soit une primitive de résultat seule, soit une primitive de résultat ainsi qu'une demande de solutions.

Cas 1. Le nœud attend une solution. Son état passe à Attente_Cut_Solution(figure 14).

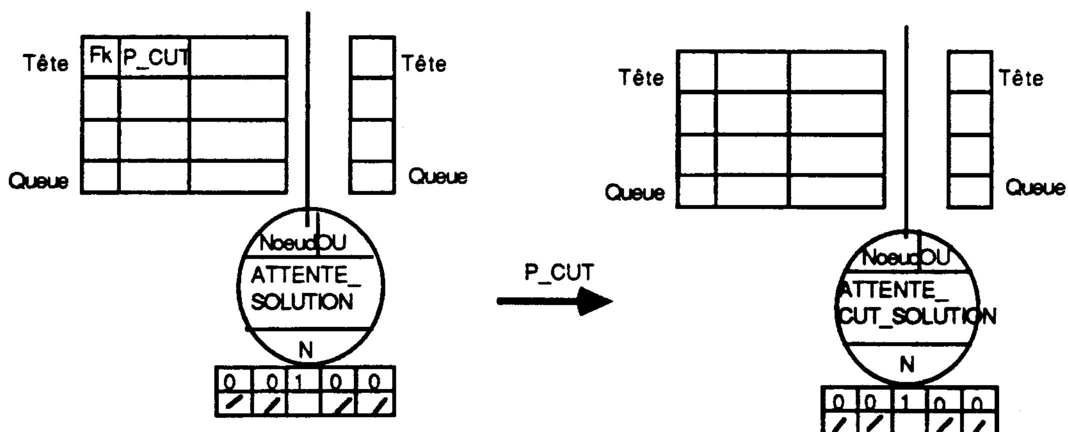


Figure 14.

Cas 2. Le nœud attend à la fois une solution et une demande de solution. Seul, l'état du nœud est modifié : il passe en Attente_Cut_Double (figure 15).

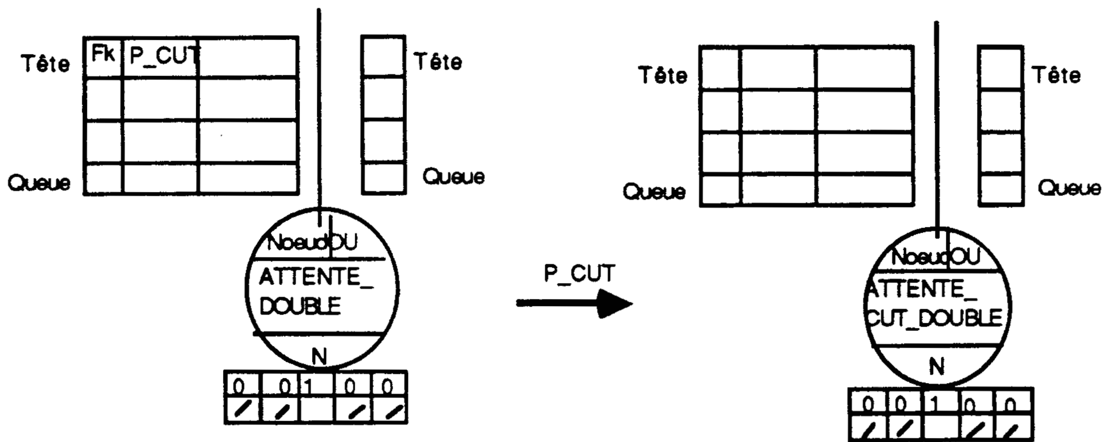


Figure 15.

III.2.g. La primitive P_UNLOCK(Nœud).

Un nœud OU qui interprète une primitive P_UNLOCK est soit dans l'état Attente_Double, soit dans l'état Attente_Levée_Verrou, états détaillés dans le chapitre 3, soit dans l'état Attente_Cut_Double, soit dans l'état Attente_Cut_Levée_Verrou. Un nœud OU dans ces deux derniers états est le "grand-père" d'un nœud CUT.

Cas 1. L'état du nœud OU est Attente_Cut_Double. Il attend conjointement une demande de solutions émanant du nœud père et une solution en provenance du nœud ET dont l'un des fils est le nœud CUT repéré. La réception d'une primitive P_UNLOCK ne modifie que l'état du nœud : d'Attente_Cut_Double, il passe à Attente_Cut_Solution (figure 16).

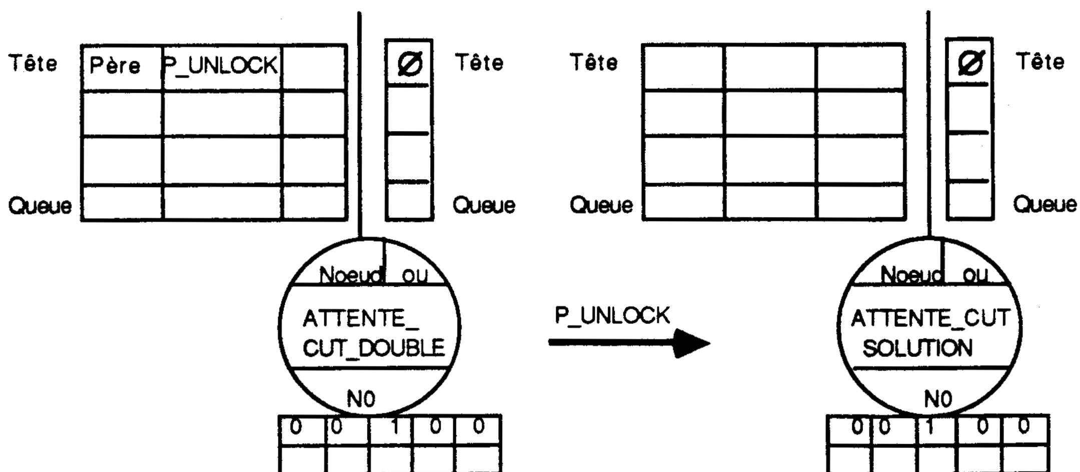


Figure 16.

Cas 2. L'état du nœud est Attente_Cut_Levée_Verrou. Le buffer d'environnements de ce nœud contient un environnement-solution. Un fils est encore présent dans la table de liens. Il s'agit du nœud ET père

du CUT. Il est encore en mesure de retourner au moins une solution. Le nœud émet une primitive de résultat destinée à son propre père ainsi qu'une primitive de demande de solution destinée au nœud ET, père du CUT. Il change d'état (figure 17) : ce dernier passe à Attente_Cut_Double.

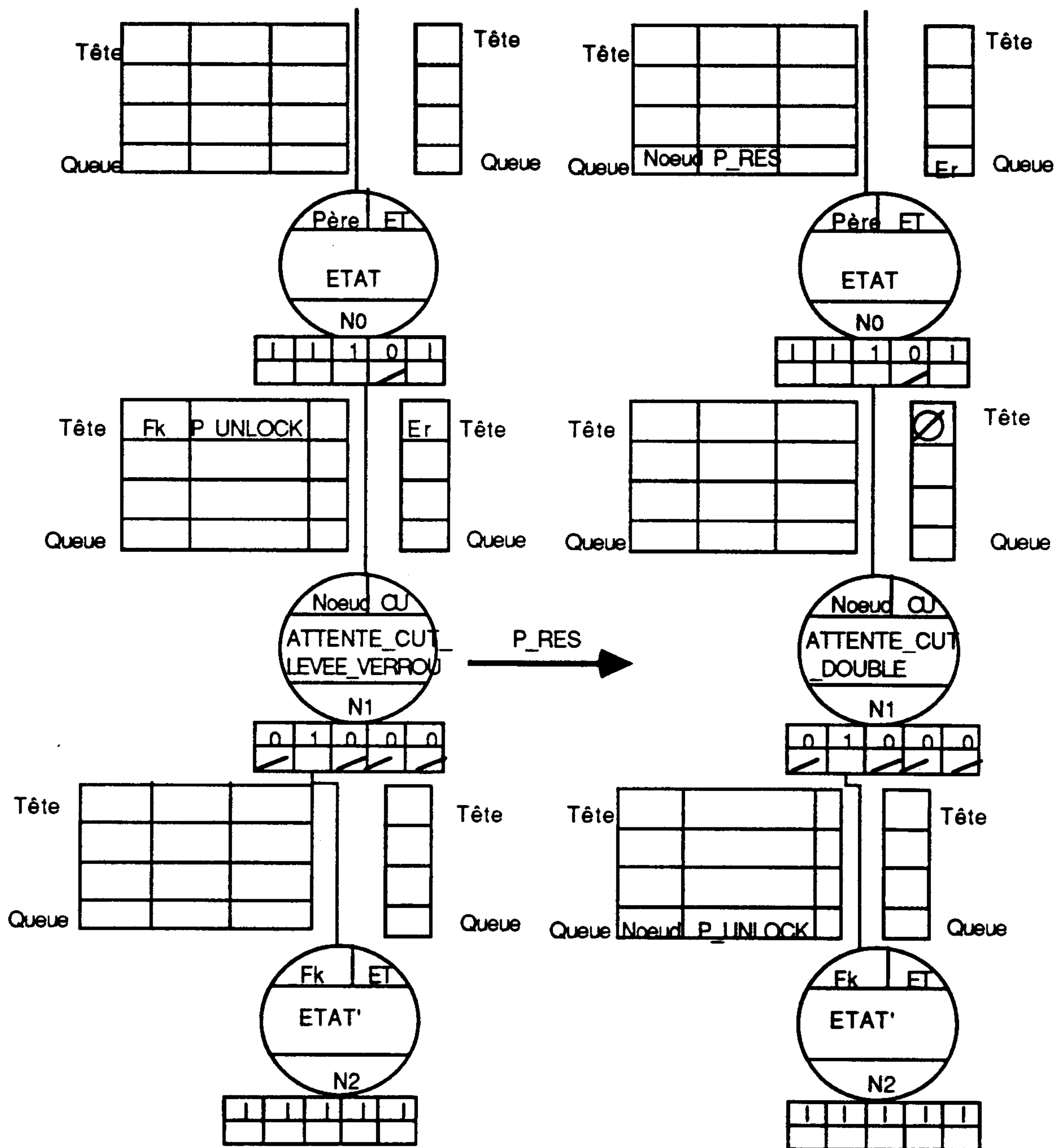


Figure 17.

<u>III.2.h. La primitive P RES(Fk, Noeud, Attributs, Environnement).</u>

L'introduction des nouveaux états Attente_Cut_Solution et

Attente_Cut_Double, relatifs au nœud OU, grand-père du CUT, modifie l'interprétation de la primitive de résultat par le gérant du nœud OU.

Un nœud OU qui reçoit une primitive de résultats est soit dans les états Attente_Double ou Attente_Levée_Verrou, états décrits dans les chapitres précédents, soit dans les états Attente_Cut_Double ou Attente_Cut_Solution. Dans ces deux derniers états, le nœud attend une solution d'un de ses nœuds fils, qui a d'ailleurs "détecté" un CUT. En conséquence, le nœud OU ne doit activer ses autres fils sous aucun prétexte, même si la solution reçue est la première retournée par le nœud fils courant actif. Le comportement du nœud dépend donc de l'état courant du nœud ainsi que de la valeur des attributs premier et dernier de la primitive résultat en cours d'interprétation.

Cas 1. L'état du nœud est Attente_Cut_Solution. Le nœud attend une. solution évaluée par le nœud fils F_k, père du CUT.

La valeur de chacun des attributs premier et dernier de la primitive P_RES interprétée est soit vrai, soit faux. Quatre cas sont donc envisageables : premier (respectivement dernier) est seul vrai, les deux attributs sont vrais (respectivement faux).

> Cas 1.a. L'attribut premier est seul vrai. Le nœud construit une primitive résultat et une primitive de demande de solutions qu'il dépose respectivement dans les buffers de primitives des nœuds père et fils (F_k). Son état passe à Attente_Cut_Solution (figure 18).

Remarque : L'attribut premier de la primitive de résultat construite par le nœud OU est vrai si le fils Fk est le premier des fils du nœud (en d'autres termes, k vaut un) ou si le nœud OU n'a reçu de ses précédents fils que des primitives d'échec (lorsque k est supérieur à un).

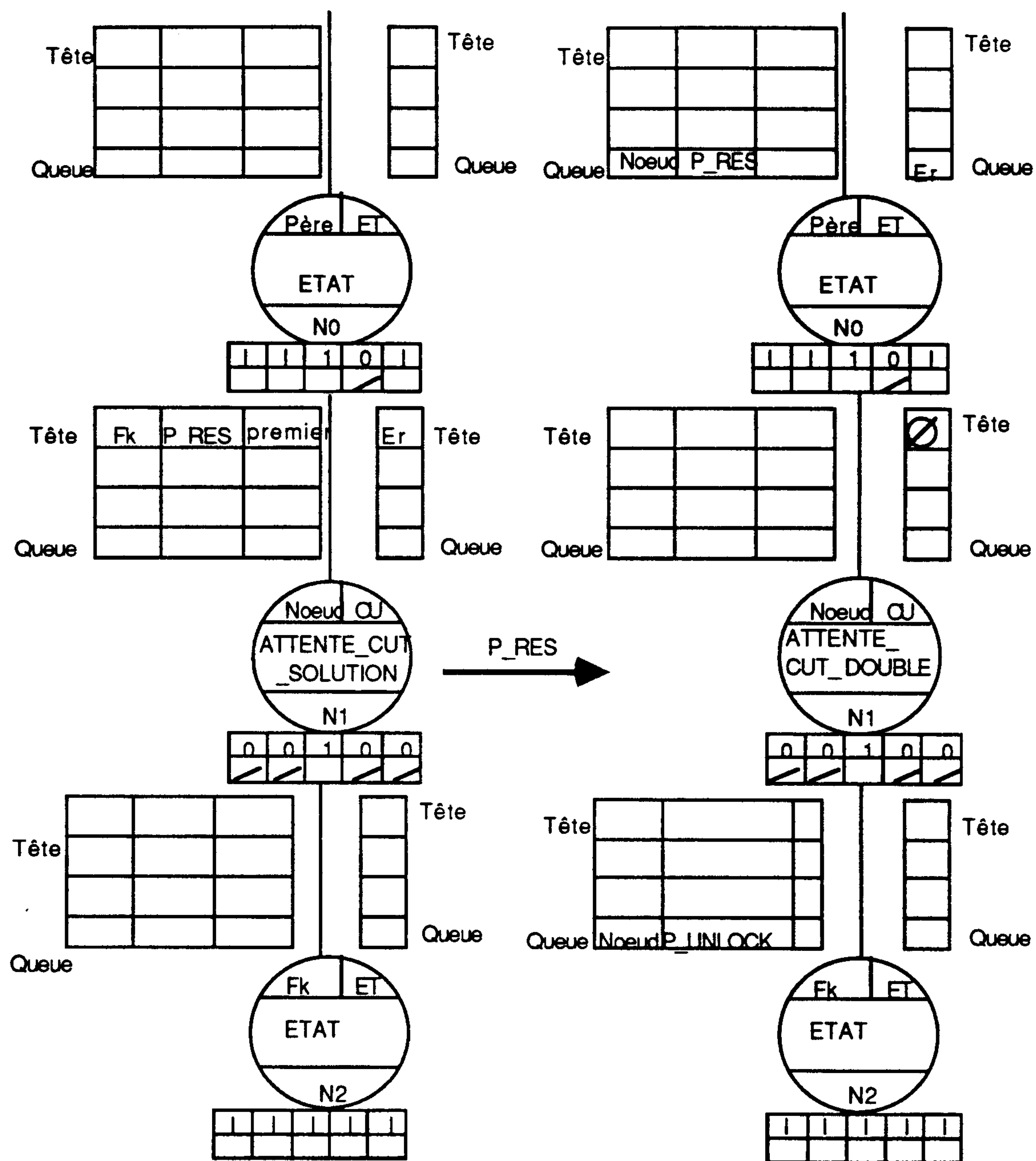


Figure 18

<u>Cas 1.b. Les attributs premier et dernier sont vrais</u>. Le nœud fils F_k a envoyé la dernière solution qu'il est en mesure d'évaluer. Cette dernière solution du nœud fils F_k est aussi la dernière calculée par le nœud OU. Le nœud OU met à jour sa table de liens : le bit d'existence prend la valeur 0, le lien est retiré de la table. Le nœud émet deux primitives, qui sont respectivement une primitive de résultat destinée au nœud père, primitive dont l'attribut dernier est vrai, et une primitive de destruction à sa propre intention. Le nœud passe dans l'état résolu.

<u>Remarque</u> : Comme dans le cas précédent, l'attribut premier de la primitive émise vaut vrai si le nœud Fk est le premier fils du nœud (k vaut un).

<u>Cas 1.c. L'attribut dernier est seul vrai</u>. Le nœud vient donc de recevoir la dernière des solutions calculées par ses nœuds fils. Il effectue donc des actions identiques aux précédentes à savoir la mise à jour de la table de liens et l'émission des primitives de résultat et destruction. Son état devient donc résolu (figure 19).

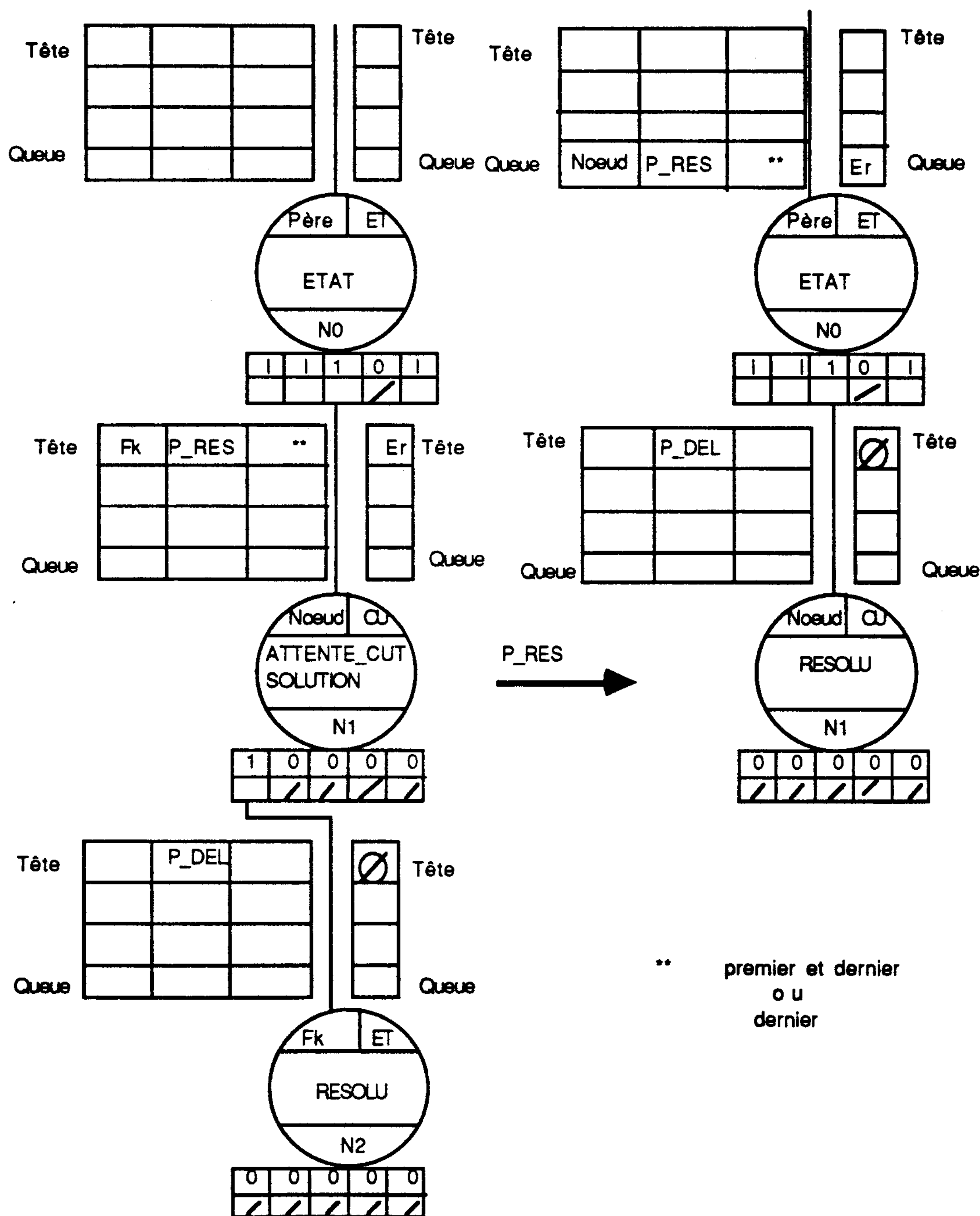


Figure 19.

Remarque : Les deux cas précédents sont donc gérés de manière identique.

Cas 1.d. Aucun attribut n'est vrai. Le nœud émet deux primitives, l'une de type résultat à destination de son père et l'autre de type demande de solution destinée au fils Fk. Son état passe en Attente_Cut_Double (figure 20).

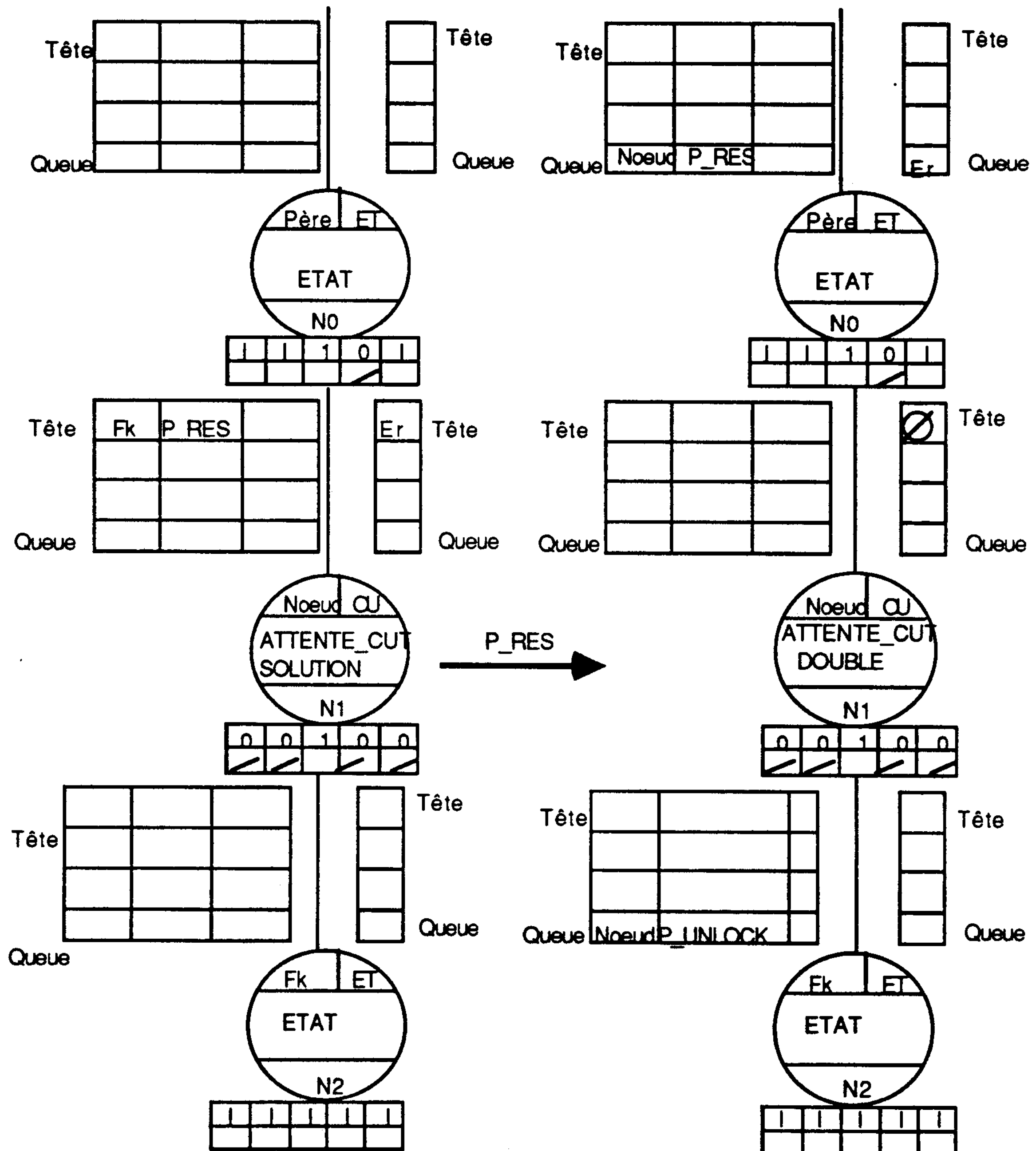


Figure 20.

Cas 2. L'état du nœud est Attente_Cut_Double. Le nœud attend conjointement une demande de solution en provenance de son père et une solution d'un de ses nœuds fils Fk. L'environnement-solution est

mémorisé dans le buffer d'environnements.

La valeur des attributs premier et dernier étant toujours vrai ou faux, on retrouve les quatre cas précédemment étudiés.

Cas 2.a. L'attribut premier est seul vrai. Le nœud attend maintenant une demande de solutions émanant de son père. Son état devient donc Attente_Cut_Levée_Verrou (figure 21).

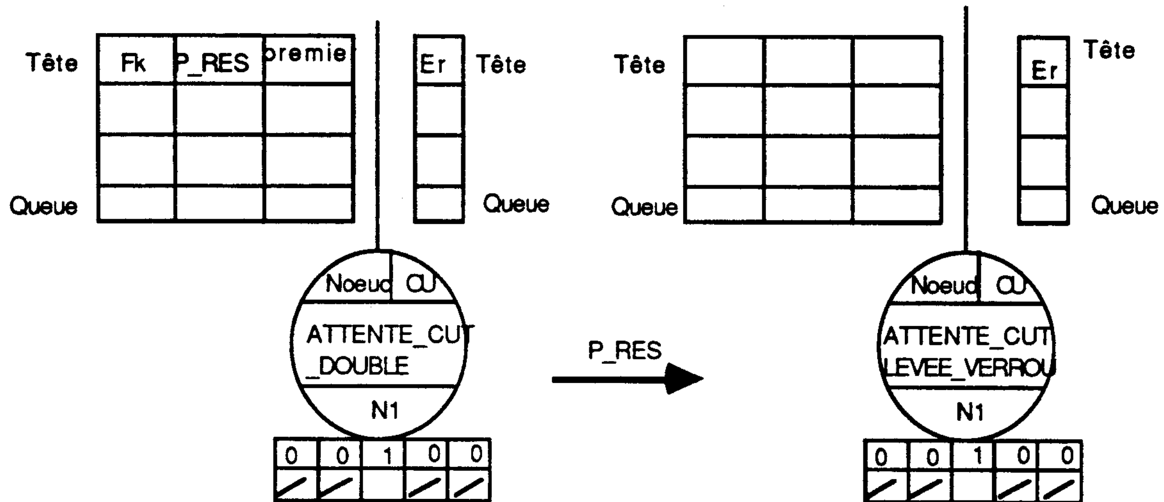


Figure 21.

Cas 2.b. Les attributs premier et dernier sont tous deux vrais. Le nœud réactualise sa table de liens : le bit d'existence passe à 0, le lien vers le fils F_k est retiré de la table.L'état du nœud devient Attente_Cut_Levée_Verrou (figure 22).

Cas 2.c. L'attribut dernier est seul vrai. Le nœud effectue des actions identiques au cas précédent : la table de liens est mise à jour (figure 22).

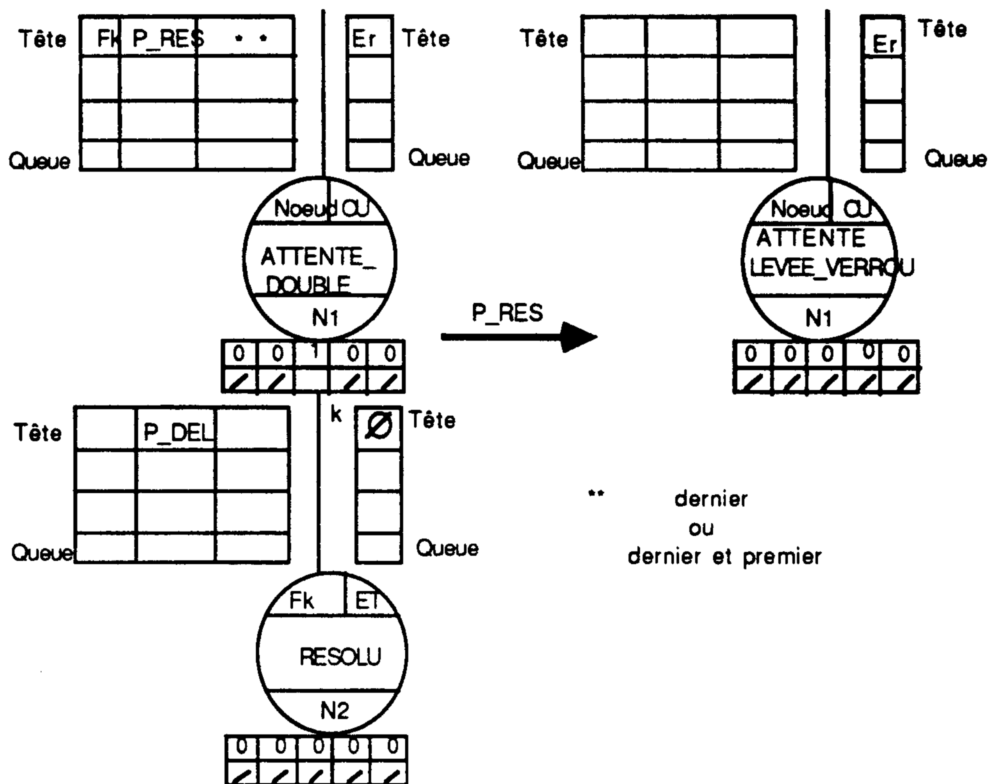


Figure 22.

Cas 2.d. Aucun attribut n'est vrai. Seul l'état du nœud est modifié. D'Attente_Cut_Double, il passe à Attente_Cut_Levée_Verrou (figure 23).

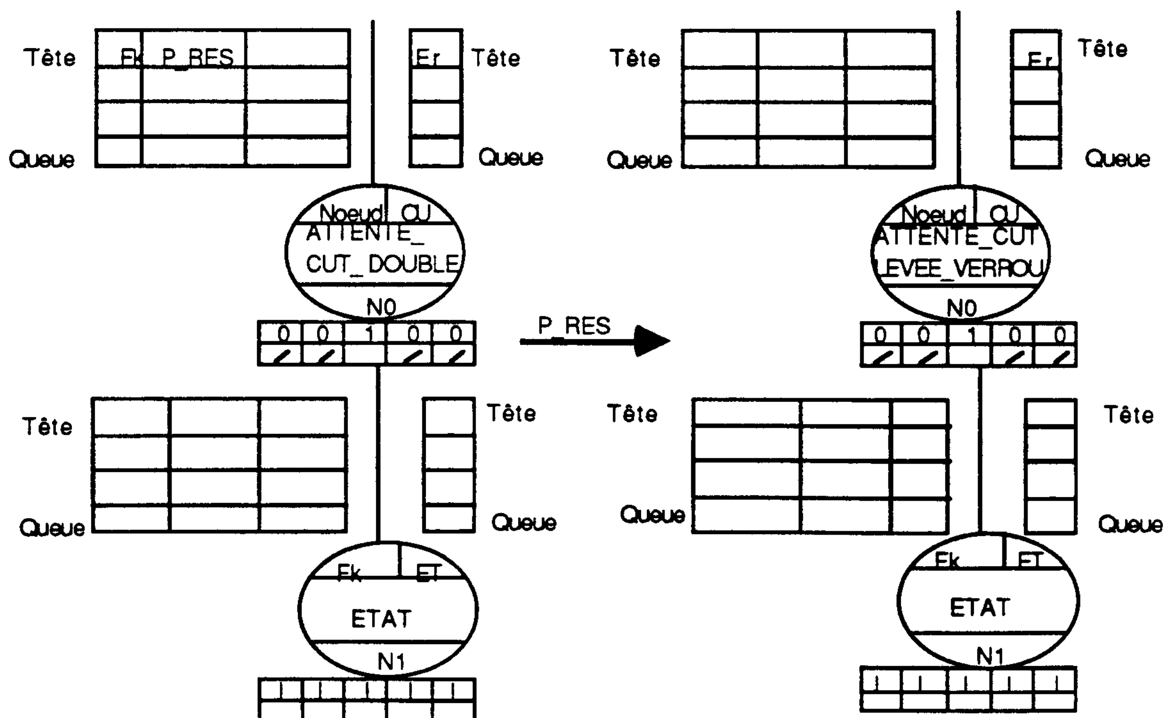


Figure 23.

III.2.i. La primitive P_ECHEC(Fk,Nœud) .

Un nœud OU qui reçoit une telle primitive est dans l'un des états suivants: Attente_Solution, Attente_Double, Attente_Cut_Solution, Attente_Cut_Double. Les deux premiers états ont été abordés dans les chapitres précédents. Nous nous attarderons donc sur l'interprétation de la primitive échec par un nœud OU qui est en Attente_Cut_Solution ou en Attente_Cut_Solution.

Cette primitive transmet l'échec évalué par le nœud Fk en l'occurrence le père du nœud CUT, vers le nœud. Ce dernier réactualise sa table de liens : il met à zéro le bit de présence d'indice k, et retire le lien correspondant. Tous les bits d'existence sont maintenant à zéro. La table de liens est donc vide. Le comportement du nœud est fonction de son état.

Cas 1. Le nœud est en Attente_Cut_Solution. Il est autorisé à retourner l'échec à son propre père. Il émet deux primitives. L'une est une primitive d'échec, qu'il dépose dans le buffer de primitives de son père, l'autre est une primitive d'auto-destruction qu'il dépose dans son propre buffer de primitives. Son état passe à résolu (figure 24).

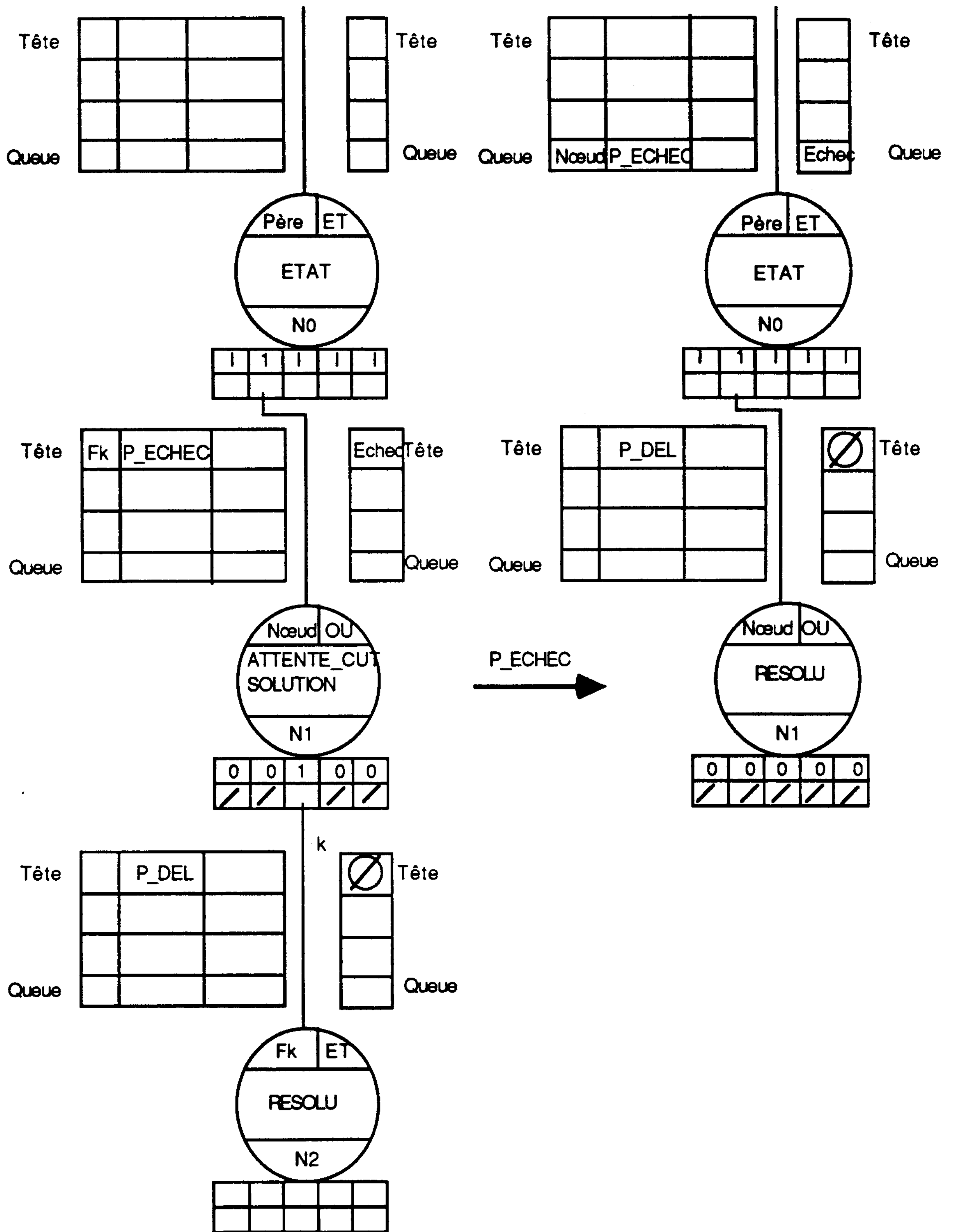


Figure 24.

Cas 2. Le nœud est en Attente_Cut_Double. Il mémorise l'échec évalué par son fils dans son buffer d'environnements, et attend une demande de solutions en provenance de son père. L'état du nœud passe à Attente_Cut_Levée_Verrou (figure 25).

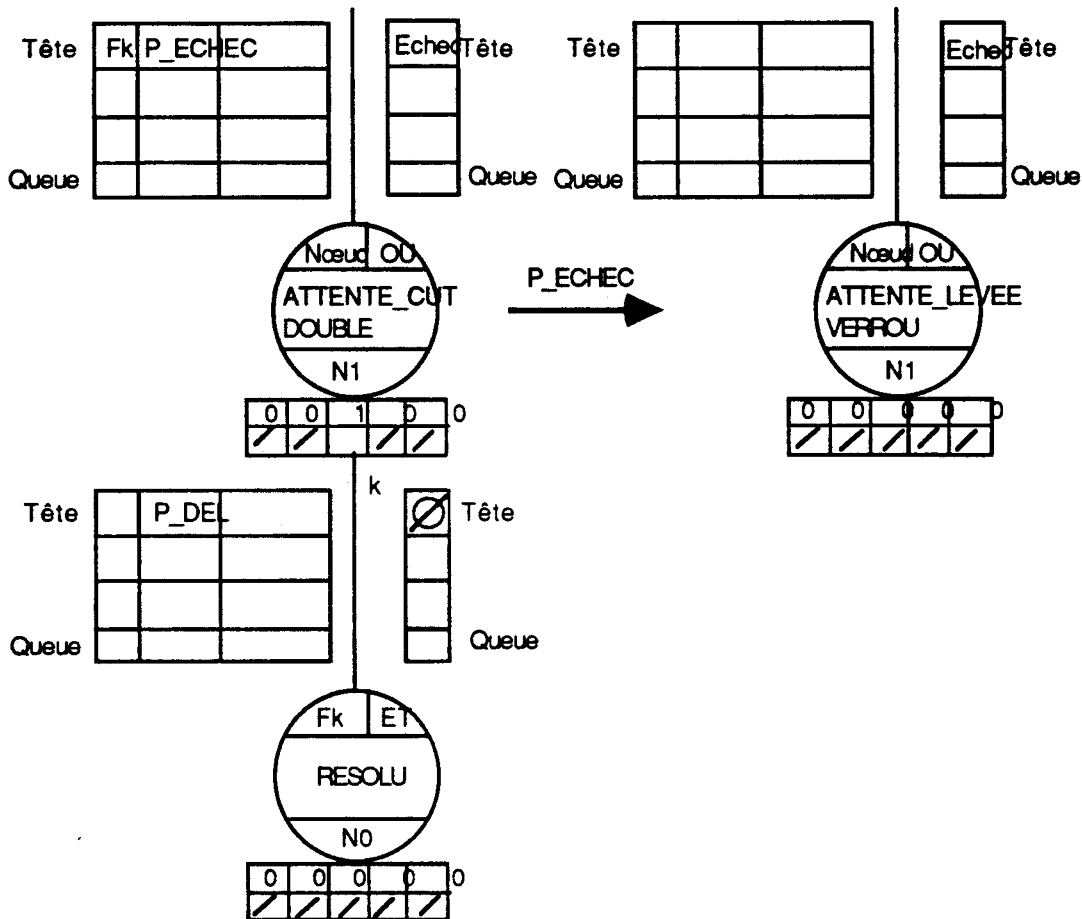
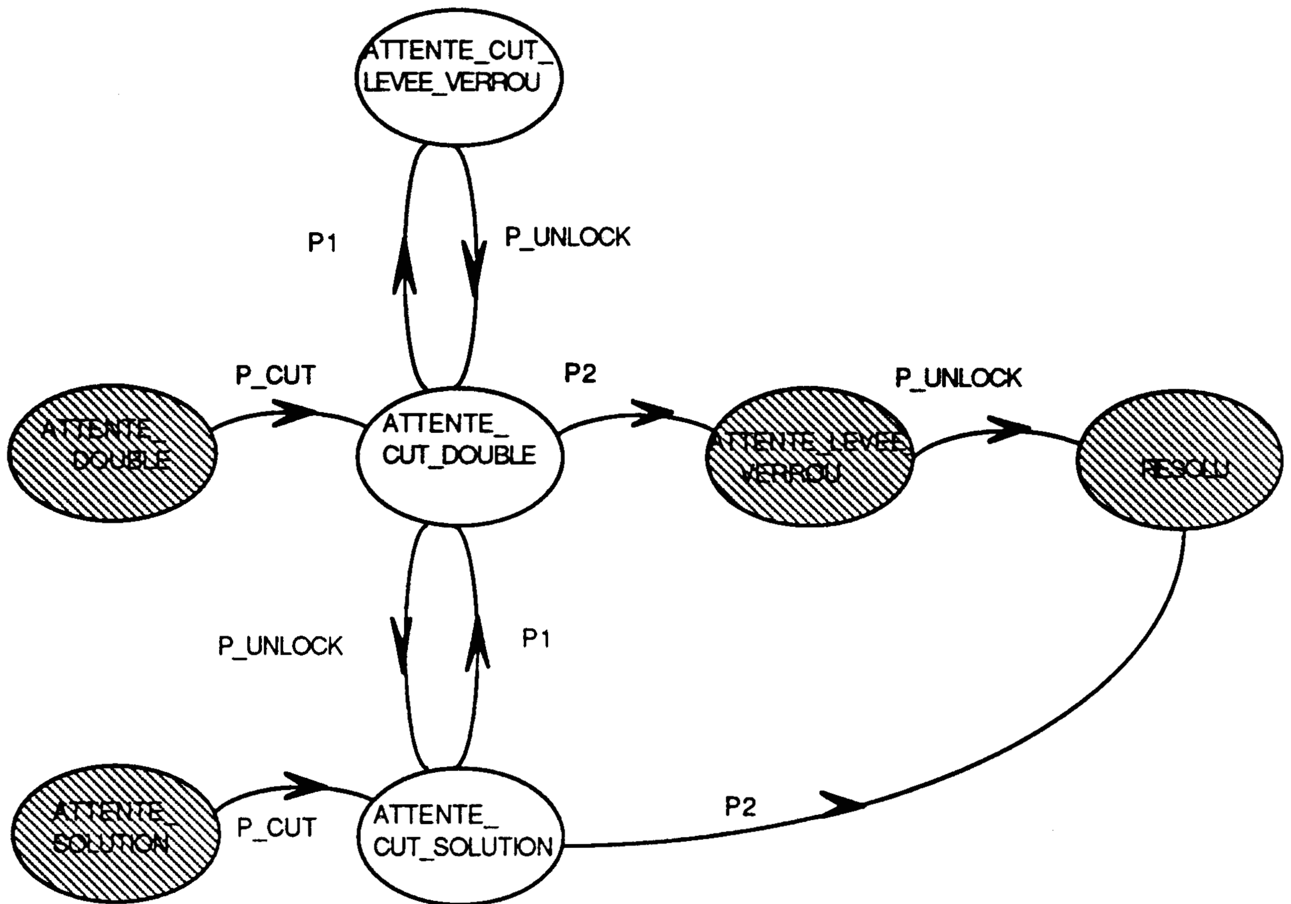


Figure 25.

<u>Remarque</u> : Les modifications apportées au gérant d'un nœud OU peuvent se résumer en le graphe suivant, dit graphe de transitions (figure 26). Les sommets de ce graphe sont les états du nœud, les arcs sont quant à eux étiquetés par la prochaine primitive interprétée par le nœud. Les sommets hachurés représentent les états déjà décrits dans le chapitre.



où *) P1 est une primitive P_RES dont l'attribut dernier n'est pas vrai

*) P2 est soit une primitive P_RES dont l'attribut dernier est vrai soit une primitive d'échec

Figure 26.

III.2.j. Etude du comportement d'un nœud ET, père d'un CUT et activé en mode restreint.

Dans le paragraphe précédent, nous avons présenté la prise en compte d'un CUT par un nœud ET activé en mode "normal". Le traitement d'un CUT par un nœud ET activé en mode restreint est légèrement différent.

En effet, un nœud ET, père d'un CUT émet une primitive P_CUT à l'intention de son propre nœud père, afin de l'avertir de la présence de ce CUT. Or, un nœud ET activé en mode restreint ne peut communiquer avec son père. Il doit donc mémoriser l'information relative au CUT jusqu'à la réception de la primitive demande de solution.

Ainsi, un nœud ET, activé en mode restreint, dont l'un des nœuds fils est un CUT, diffuse aux éventuels prédécesseurs du nœud CUT un ordre de destruction, puis se met en attente des accusés de réception correspondants. L'état du nœud est alors Attente_Ack_Kill_Double. Lorsque tous les accusés de réception sont parvenus au nœud ET, ce dernier active le successeur éventuel du nœud CUT, tout en mémorisant l'information relative à la présence d'un CUT. L'état du nœud exprime cette information. L'interprétation d'une demande de solution permet le dépôt d'une primitive P_CUT dans le buffer de primitives du nœud père. Le nœud ET gère alors normalement l'avancée du pipeline. Nous allons détailler chacune des transitions du graphe présenté en figure 27.

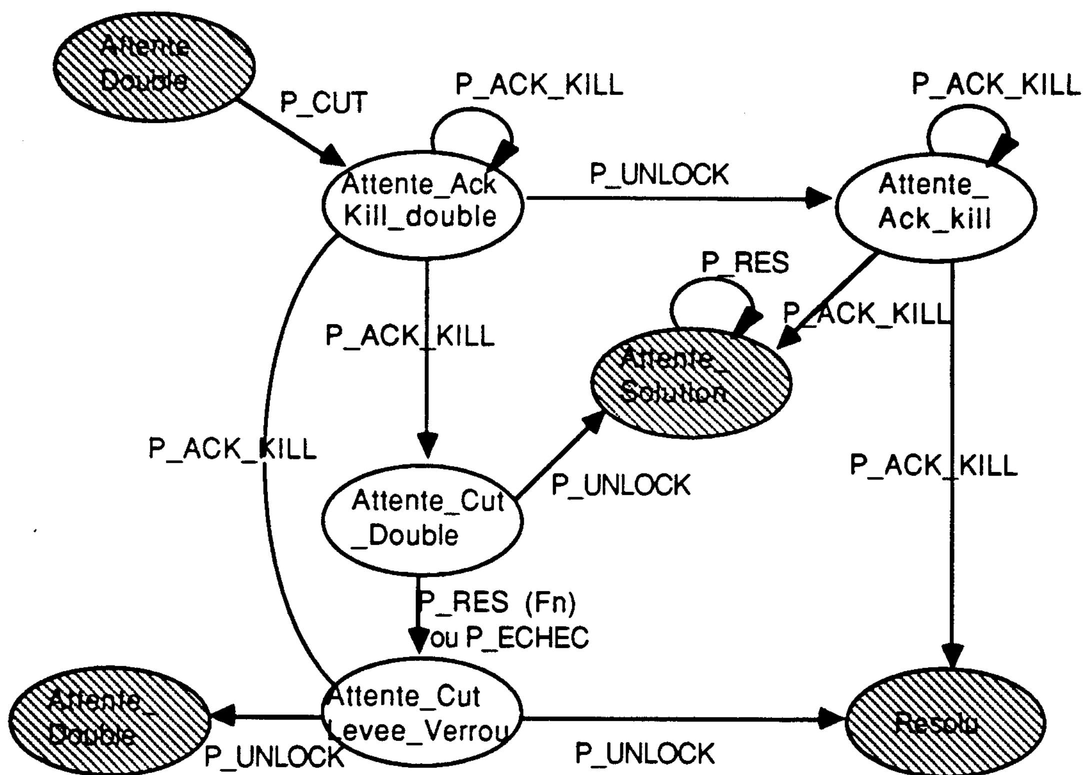


Figure 27.

1). L'état du nœud est Attente_Ack_Kill_Double.

Dans cet état, le nœud ET ne tient compte que des accusés de réception, et de la demande de solutions. Toute autre primitive, notamment celle émise par un nœud fils en cours de destruction, est ignorée.

1.a). La primitive interprétée est une primitive demande de solutions. Le nœud passe à l'état Attente_Ack_Kill. De plus, le nœud émet une primitive P_CUT, qu'il dépose dans le buffer de primitives de son père.

1.b).La primitive interprétée est un accusé de réception émanant d'un fils F_k. Le nœud émet une primitive P_CUT, qu'il dépose dans le buffer de primitives de son père. De plus, le nœud met à jour sa table de liens. Le bit d'existence d'indice k est mis à zéro, le lien est détruit.

1.b.1). Si l'un au moins des prédécesseurs du nœud CUT n'a pas encore retourné son accusé de réception, l'état du nœud demeure inchangé.

A l'inverse, lorsque le nœud a reçu les accusés de réception des prédécesseurs du CUT, il teste sa table de liens afin de déterminer un éventuel successeur du nœud CUT à activer.

1.b.2). Le nœud CUT est le dernier fils du nœud. Ce dernier a alors évalué un environnement-solution complet et unique qu'il mémorise dans son buffer d'environnements. L'état du nœud passe à Attente_Cut_Levée_Verrou.

1.b.3). Le nœud CUT a au moins un successeur, que le nœud active. Il attend alors un environnement-solution de ce successeur. Son état passe à Attente_Cut_Double.

2). L'état du nœud est Attente_Cut_Levée_Verrou.

Le nœud attend une demande de solution. Lorsqu'elle lui parvient, il émet consécutivement deux primitives. L'une est la primitive P_CUT, qui lui permet d'avertir son père de la présence d'un CUT, l'autre est une primitive résultat. La valeur de l'attribut dernier de la primitive résultat émise est calculé en fonction du contenu de la table de liens.

2.a). Si tous les bits d'existence de la table de liens sont à zéro, l'état du nœud devient résolu. L'attribut dernier de la primitive est alors vrai.

2.b). Si le fils toujours actif est le dernier fils du nœud, le nœud active ce fils. L'état du nœud devient Attente_Double.

2.c). Si le nœud fils actif n'est pas le dernier fils du nœud, seul l'état du nœud est modifié. Il passe en Attente_Double.

3). L'état du nœud est Attente_Cut_Double.

Le nœud attend deux types de primitives : des primitives résultat émanant de ses fils (voire des primitives d'échec), une primitive demande de solutions construite par son père.

3.a). La primitive interprétée est une primitive demande de solutions. Le nœud émet une primitive P_CUT destinée à son père, et son état passe à Attente_Solution. La gestion de l'avancée se poursuit normalement.

La primitive interprétée est une primitive résultat.

3.b). Si elle émane d'un fils autre que le dernier, le nœud ET active le successeur de ce nœud. L'état du nœud demeure inchangé.

3.c). Si elle provient du dernier fils, le nœud ET mémorise l'environnement-solution dans son buffer d'environnements et son état passe à Attente_Cut_Levée_Verrou.

Remarquons que si la primitive interprétée est une primitive échec, le nœud ET exécute la procédure détaillée dans le chapitre précédent. Ainsi, il ignore l'échec si l'un au moins de ses nœuds fils est encore actif et le considère sinon comme définitif.

III.3. Mise en œuvre du CUT dans le modèle généralisé.

La mise en œuvre du CUT dans un modèle généralisé (modèle où le nombre de branches activées en mode restreint est supérieur à un) est très similaire à celle décrite plus haut. Il faut toujours empêcher tout retour en arrière sur les prédicats qui précèdent le CUT dans le corps de la clause, ainsi que l'activation des alternatives du paquet non encore envisagées. Or, celles-ci ont pu être activées en mode restreint. Il convient donc de déclencher la destruction des arborescences correspondantes sans perturber le fonctionnement du modèle.

Considérons à nouveau l'exemple de la figure 3 et supposons que les nœuds B(3) et B(4) aient été activés en mode restreint. Le nœud B doit donc déclencher la destruction des arborescences de racine respective les nœuds B(3) et B(4). Le nœud B diffuse à ces deux fils l'ordre de détruire et attend les accusés de réception correspondants. Le nœud B ne pourra être détruit qu'à la triple condition suivante .

- Le nœud B(2) a émis sa primitive "fin de travail".
- Le nœud B(3) a émis son accusé de réception.
- Le nœud B(4) a émis son accusé de réception.

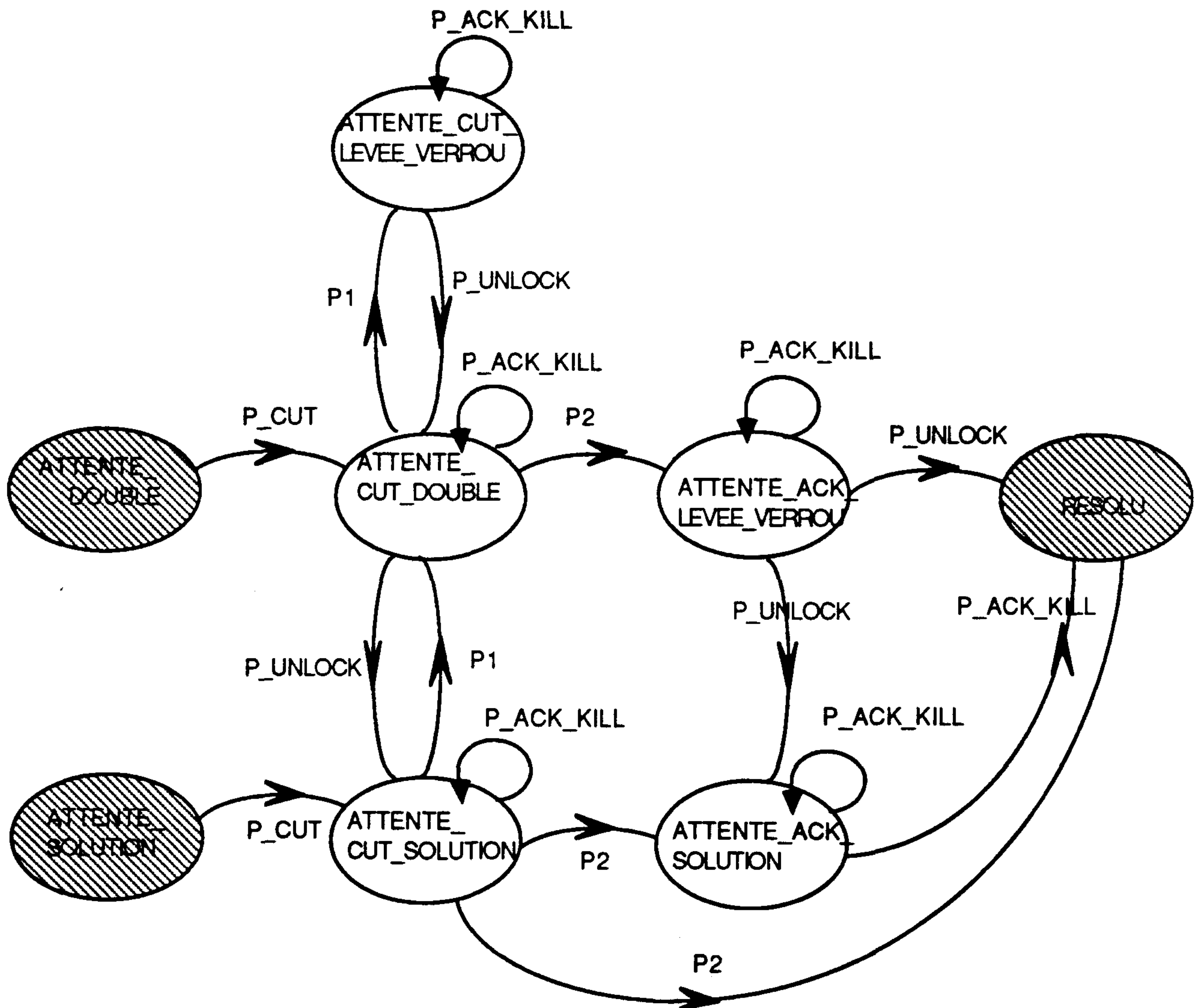
Or les nœuds B(2), B(3), B(4) travaillent de façon asynchrone. Plusieurs cas de figure peuvent se produire, dont les deux principaux suivants :

- Les accusés de réception émis par les nœuds B(3), B(4) parviennent au nœud B avant la primitive "fin de travail" expédiée par le nœud B(2).

- La primitive émise par le nœud B(2) précède les accusés de réception.

Une solution, simple à mettre en œuvre, consiste à retarder l'émission de la dernière solution émise par le nœud B jusqu'à ce que tous les accusés de réception soient parvenus au nœud B.

L'algorithme du nœud OU est donc légèrement modifié. On obtient le graphe de transitions suivant (figure 28) :



où *) P1 est une primitive P_RES dont l'attribut dernier n'est pas vrai
*) P2 est soit une primitive P_RES dont l'attribut dernier est vrai soit une primitive d'échec

Figure 28.

Un nœud OU qui reçoit la primitive P_CUT, passe soit dans l'état Attente_Cut_Double soit dans l'état Attente_Cut_Solution. Il diffuse alors l'ordre de destruction aux fils activés en mode restreint et attend leur accusé de réception.

La réception d'une primitive P_ACK_KILL dans l'un des états suivants

* Attente_Cut_Double,
* Attente_Cut_Solution
* Attente_Cut_Levée_Verrou

ne modifie en rien le comportement du nœud. Seule, la table de liens est réactualisée : le bit d'existence est remis à zéro, le lien vers le nœud fils détruit.

La réception d'une primitive "fin de travail" est plus délicate à gérer. Le nœud, qui interprète cette primitive peut soit attendre uniquement une primitive résultat, soit attendre une primitive résultat ainsi qu'une primitive de demande de solution. Nous allons examiner chacune de ces alternatives.

1). L'état du nœud est Attente_Cut_Solution:

1.a) Si la table de liens est vide, le nœud adopte un comportement "normal". Il dépose , dans le buffer de primitives de son noeud père, une primitive "fin de travail" (dont la classe est fonction de celle qu 'il a reçue) et passe en état résolu.

1.b) Si la table de liens n'est pas vide (l'un au moins des fils activés en mode restreint n'a pas renvoyé son accusé de réception), le nœud passe dans l'état **Attente_Ack_Solution**. L'environnement solution est mémorisé dans le buffer correspondant.

2). L'état du nœud est Attente_Cut_Double:

Le nœud mémorise l'environnement solution dans son buffer d'environnements. Son état devient alors **Attente_Ack_Levée_Verrou**. Le nœud peut alors recevoir deux types de primitives, en l'occurrence une demande de solutions émanant de son père, des accusés de réception en provenance de nœuds fils activés en mode restreint et non encore détruits.

Il nous reste à observer le comportement du nœud dont l'état est soit Attente_Ack_Levée_Verrou, soit Attente_Ack_Solution

3). L'état du nœud est Attente_Ack_Levée_Verrou :

3.a). Si la table de liens est vide, le nœud peut retourner à son

père la dernière solution (voire l'échec) qu'il a évaluée. Le nœud peut alors être détruit: il passe dans l'état résolu et attend de son père l'autorisation de se détruire.

3.b). Si la table de liens n'est pas vide (l'un au moins des fils activés en mode restreint n'a pas renvoyé son accusé de réception), le nœud passe dans l'état **Attente_Ack_Solution**. L'environnement solution est mémorisé dans le buffer correspondant.

* *L'état du nœud est Attente_Ack_Levée_Verrou :*

Dans cet état, un seul type de primitives est accepté, à savoir l' accusé de réception en provenance des fils activés en mode restreint. La table de liens est modifiée en conséquence. Dès que cette table est vide, le nœud émet une primitive de "fin de travail" destinée à son nœud père et passe en l'état résolu.

IV. LES PREDICATS DE LECTURE ET D'ECRITURE.

IV.1. Principes de la mise en œuvre des prédicats de lecture et d'écriture dans le modèle.

En Prolog séquentiel, l'ordre d'exécution des prédicats de lecture et écriture a son importance. Notre modèle respectant la sémantique opérationnelle de Prolog, et construisant en conséquence un ensemble de solutions identique à celui obtenu par une exécution séquentielle du programme, se doit de respecter cet ordre. Sans contrôle, l'exécution parallèle de plusieurs branches OU peut en effet perturber l'ordre requis.

Or, le modèle de résolution proposé possède la propriété suivante : à tout instant t, il existe une seule branche dans l'arbre dont les nœuds sont tous actifs. En d'autres termes, les nœuds de cette branche sont tous en mesure de retourner à leur père la solution qu'ils sont en train d'évaluer.

- En effet, à tout instant t, un nœud OU n'accepte de solutions qu'au plus d'un de ses nœuds fils. Ses autres descendants sont soit inactifs, soit activés en mode restreint.

- Un nœud ET,quant à lui, peut dialoguer avec plusieurs de ses nœuds fils : en effet, il gère un pipeline ET, qui ne doit pas être "désamorcé". Toutefois, le nœud ET maintient un lien "privilégié" avec le nœud fils actif , le plus "à droite" du pipeline.

Une solution simple à mettre en œuvre consiste alors à n'autoriser

l'exécution immédiate (c'est-à-dire dès qu'elle est demandée) d'une opération de lecture (respectivement d'écriture) que si cette dernière survient sur la branche active de l'arbre. Les autres opérations de lecture (respectivement d'écriture) sont suspendues jusqu'à ce que la branche concernée devienne à son tour la branche active.

Il s'avère donc nécessaire de reconnaître la branche active, c'est à dire la branche sur laquelle aucun verrou de solution n'est posé.

Dans ce but, on associe à chaque nœud un booléen BA (Branche Active). Si BA vaut vrai, le nœud appartient à la branche active. Dans le cas contraire, le nœud n'est pas sur la branche active: un verrou, au moins, existe entre la racine et le nœud.

Lorsque le nœud a une opération de lecture (voire d'écriture) à effectuer, il teste la valeur du booléen BA qui lui correspond. Si BA est vrai, alors l'opération est réalisée. Sinon, le nœud ne fait aucune action et attend la prochaine remise à jour du booléen BA. Il teste alors la nouvelle valeur de BA et agit en conséquence. L'opération de lecture (voire d'écriture) reste suspendue jusqu'à ce que le booléen BA prenne la valeur vrai.

Il nous reste à calculer la valeur d'un booléen BA. Un nœud appartient à la branche active courante si tous ses ancêtres sont sur cette branche, et si son propre verrou de solutions n'est pas positionné. Aussi, le calcul de la valeur du booléen est fonction d'une part de la valeur du booléen de son père, et d'autre part de la primitive d'activation reçue ou de la primitive résultat retournée. Dans le paragraphe suivant, nous allons détailler tour à tour le calcul de la valeur du booléen en fonction de la classe de la primitive reçue ou expédiée.

IV.2. Implantation en terme de primitives.

IV.2.a. Calcul de la valeur du booléen en fonction d'une primitive reçue par le nœud.

Les primitives P_ACT et P_LOCK fixent la valeur initiale du booléen BA associé au nœud. La primitive P_UNLOCK, quant à elle, ne fait que modifier cette valeur.

Cas 1. La primitive interprétée par le nœud est une primitive P_ACT. La valeur du booléen transmis par la primitive P_ACT est calculée par le nœud père en fonction de son propre booléen, de son type (ET/OU) et de la position du destinataire parmi l'ensemble des descendants du nœud père.

*) Si le booléen BA du nœud père a la valeur faux, le booléen BA du nœud revêt la même valeur, en l'occurrence faux.

*) Le booléen du nœud père est à vrai. Le nœud père est donc sur la branche active.

-> Le nœud père est un nœud OU. Or, à tout moment, un nœud OU ne peut dialoguer qu'avec un seul de ses fils. Le destinataire de cette primitive est ce nœud. Le booléen BA prend donc la valeur vrai.

-> Le nœud père est un nœud ET, qui peut donc dialoguer simultanément avec plusieurs de ses descendants. Cependant, il a une relation privilégiée avec un seul de ses fils : le dernier du pipeline. Si le destinataire est le dernier fils du pipeline, le booléen transmis est vrai. A l'inverse, le booléen transmis est faux.

Cas 2. La primitive interprétée par le nœud est une primitive P_LOCK. La primitive P_LOCK positionne le verrou de solutions associé au nœud: la valeur de BA est donc faux.

Cas 3. La primitive interprétée par le nœud est une primitive P_UNLOCK. La valeur du booléen BA d'un nœud qui reçoit une primitive P_UNLOCK est toujours faux (le nœud ne pouvant émettre aucune primitive de résultat à l'intention de son père avant que ce dernier ne lui en fasse la demande explicite). La valeur du booléen transmis peut être soit faux, soit vrai. Examinons ces deux possibilités.

1) les deux booléens sont à faux, il y a identité de valeur entre le booléen BA du nœud, et le booléen transmis dans le message. Dans cette alternative, le booléen BA conserve sa valeur.

2) Le booléen transmis vaut vrai. Les valeurs des booléens différent donc : la valeur booléenne BA du nœud est modifiée. Le nœud transmet cette nouvelle valeur à un seul de ses fils, le fils "actif":

*) Le fils "actif" d'un nœud OU est le fils le plus à gauche du nœud, encore activé.

*) Le fils "actif" d'un nœud ET est le fils le plus à droite du nœud, encore activé.

Une primitive supplémentaire, dite primitive de propagation est nécessaire. Cette primitive prend deux arguments à savoir les nœuds émetteur et destinataire. Elle est transmise de nœud père en nœud "actif".

Cas 4. La primitive reçue est une primitive " fin de travail" (c'est-à-dire, une primitive P_ECHEC, ou une primitive P_RES, dont l'attribut dernier est positionné). Seule la classe des nœuds ET est concernée.

Lorsque l'émetteur de la primitive est le fils "actif" du nœud ET, le nœud ET doit remettre à jour la valeur du booléen BA du prédécesseur actif de l'émetteur dans le pipeline (ce prédécesseur devient à son tour le nœud fils "actif" du pipeline"). Cette remise à jour n'est bien sûre réalisée que si le booléen du nœud ET a la valeur vrai.

IV.2.b Calcul de la valeur du booléen consécutif à l'émission d'une primitive par le nœud.

- L'émission d'une primitive résultat, qui n'est pas une primitive "fin de travail", modifie aussi la valeur du booléen BA. En effet, l'envoi de cette solution vers le nœud père suspend momentanément le dialogue père-fils (ce dialogue reprend sur la réception d'une primitive demande de solution émanant du nœud père). En conséquence, le booléen BA prend la valeur Faux.

- L'émission d'une primitive résultat, qui est une primitive "fin de travail" n'a pas de réelle incidence sur la valeur du booléen BA. En effet, le nœud est prêt à être détruit.

Remarque:

Les primitives d'activation immédiate et restreinte, les primitives demande de solution et de résultat nécessitent donc un argument supplémentaire, de nature booléenne, en l'occurrence le booléen BAP (Branche Active du père). La diffusion restreinte qui permet la mise à jour de la valeur du booléen nécessite une primitive supplémentaire.

V. CONCLUSION.

Ce chapitre a essentiellement montré la mise en œuvre d'une part du prédicat CUT, d'autre part des prédicats de lecture/écriture dans notre modèle.

La plupart des modèles multi-séquentiels intègrent des algorithmes de reconnaissance de la branche la plus à gauche de l'arbre (c'est-à-dire de la branche active de l'arbre).Ces algorithmes, déclenchés lors de l'évaluation d'un CUT, sont nécessaires car ces modèles ne tiennent pas compte de l'ordre des solutions.

Dans notre modèle, la prise en compte du CUT n'implique pas la reconnaissance de la branche la plus à gauche de l'arbre. En effet, le mécanisme de verrouillage permet l'ordonnancement naturel de la production des solutions, nécessaire à l'évaluation du prédicat CUT.

Le traitement des prédicats de lecture et d'écriture dans notre modèle induit un minimum d'overhead. En effet, le calcul de la branche active est partiellement pris en charge par les messages d'activation. Le seul overhead induit par l'évaluation des prédicats de lecture et d'écriture se traduit par d'une part la transmission d'un argument supplémentaire (en l'occurrence, un argument booléen BA) dans chaque primitive, et d'autre part la mise en œuvre d'une primitive nouvelle, la primitive de propagation (cette primitive est diffusée le long d'un chemin de l'arbre afin de modifier la valeur du booléen BA de chaque nœud de ce chemin).

SIMULATION

SIMULATION

I.INTRODUCTION.

Nous allons évoquer, dans ce chapitre, les deux actions distinctes qui ont été menées parallèlement afin de valider le modèle et d'obtenir des informations relatives aux performances de ce dernier. Ces deux actions sont les suivantes :

- Une simulation abstraite du modèle
- Une observation des programmes Prolog réels exécutés selon le modèle proposé.

Nous présenterons le principe de chacune de ces actions, et en particulier l'architecture du simulateur, ainsi que les hypothèses émises et les résultats obtenus. Puis, nous exposerons le mécanisme d'introduction de prédicats espions dans un programme Prolog grâce auquel le comportement de programmes réels a été observé ainsi que les résultats obtenus lors de cette étude.

II. SIMULATION ABSTRAITE DU MODELE.

Le but de ce simulateur est double : d'une part, il permet de valider le modèle, d'autre part il permet de recueillir des informations sur le taux moyen de parallélisme développé.

II.1. Limitations.

Lors de l'écriture du simulateur, nous avons émis les hypothèses suivantes :

a) Lors de cette simulation, notre étude s'est essentiellement concentrée sur le taux de parallélisme développé lors de l'exécution d'un programme. Aussi, avons nous supprimé la notion de variables, donc d'environnements, non nécessaires dans le cadre de ce travail. Cette absence de variable ne permet pas a priori la prise en compte des succès à l'unification. Nous avons donc introduit la notion de probabilité de succès à l'unification afin de générer automatiquement des arbres de profondeur importante (c'est-à-dire des programmes récursifs).

b) L'exécution du programme se fait en deux temps. La première phase génère un arbre ET/OU statique. Cet arbre est construit à partir du programme source et du but initial. Lors de la seconde phase, l'arbre ET/OU ainsi construit est parcouru selon les règles imposées par le modèle.

c) Une horloge logique cadence le parcours de l'arbre ET/OU. En une même unité de temps logique, un nœud peut à la fois:

- émettre soit une primitive résultat (voire une primitive échec) à l'intention de son nœud père, soit une primitive d'activation (restreinte ou non), voire une demande de solution à l'intention d'un de ses nœuds fils.
- recevoir une primitive émise par son père ou par l'un de ses fils.
- procéder à une unification lorsque le nœud considéré est de type ET.

L'architecture du nœud est donc multitraitement.

d) Aucune considération architecturale n'est prise en compte. Ainsi, le nombre de processeurs est infini (on peut considérer qu'à chaque nœud est attaché un processeur).

Le respect des quatre points précédents font que les mesures de parallélisme obtenues sont optimales.

II.2. Présentation du simulateur.

Ce simulateur a été développé en langage C, sur station SUN, dans l'environnement UNIX.

II.2.a. Paramètres d'entrée.

Afin de réaliser la simulation la plus souple possible, il s'est avéré intéressant d'introduire les paramètres suivants :

- le temps qu'un nœud ET consacre à une unification.
- le temps de traitement d'une primitive par un nœud (ce temps varie selon la nature du traitement induit par la primitive dont l'interprétation est en cours).
- le temps de transit d'une primitive entre deux nœuds.
- le nombre de branches activées en mode restreint. Ce nombre varie de zéro au nombre de fils d'un nœud.

II.2.b. Paramètres de sortie.

La première phase génère un arbre ET/OU. Cette phase permet donc l'obtention des renseignements suivants :

- le nombre total de nœuds de l'arbre
- la répartition de ces nœuds par type (ET/OU)
- le nombre de niveaux de l'arbre. Ce nombre de niveaux représente la profondeur de l'arbre, ou encore le nombre d'appels de prédicats.

Lors de la seconde phase, c'est-à-dire lors du parcours de l'arbre

ET/OU, les trois fichiers de résultats suivants sont créés :

a) Un fichier de statistiques générales. Ce fichier regroupe deux types d'informations :

D'une part, il rappelle le nom du programme à exécuter, le but initial et les différents paramètres d'entrée adoptés. D'autre part, il présente :

- le nombre de primitives émises et interprétées par les nœuds.
- la répartition de ces primitives par classe, telles qu'elles ont été définies dans les chapitres précédents. Nous retrouvons donc les primitives d'activation (restreintes ou non), les primitives demandes de solutions et les primitives résultats.(échec ou non).
- le temps global d'exécution.
- le nombre total d'unifications.
- le nombre d'unifications qui ont échoué.
- le taux maximal de parallélisme et le taux moyen de parallélisme atteints, qui désignent respectivement le taux d'activité maximal obtenu à un instant donné de l'exécution du programme, et la moyenne pondérée du taux d'activité.

b) La trace détaillée des primitives par unité de temps logique. A chacune d'entre elles, sont associés l'identificateur et l'état des nœuds émetteur et destinataire, le type de la primitive.

c) Un fichier décrivant le nombre de nœuds par état et par unité de temps logique.

II.2.c. Architecture du simulateur.

II.2.c.1. Architecture fonctionnelle du simulateur.

La figure 1 présente les trois modules principaux en l'occurrence l'analyseur, la construction de l'arbre et son parcours qui constituent le simulateur.

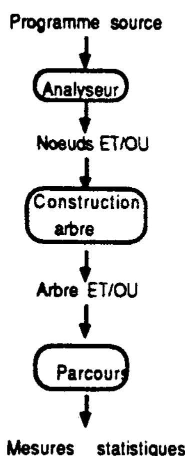


figure 1. Architecture fonctionnelle du simulateur.

L'analyseur transforme le programme à exécuter en un ensemble de modèle de nœuds ET/OU. La syntaxe du langage de simulation est la suivante :

<clause> ::= <fait> | <but> | <régle>.
<fait> ::= <tête>.
<but> ::= :- <corps>.
<règle> ::= <tête> :- <corps>.
<tête> ::= <littéral> | <littéral> <probabilité>.
<corps> ::= <suite de littéraux>.
<suite de littéraux> ::= <littéral> | <littéral> <suite de littéraux>.
<littéral ::= <chaîne>.
<chaîne> ::= <lettre> <suite de caractères>.
<caractère> ::= <lettre> | <chiffre>.
<suite de caractères> ::= <> | <caractère> <suite de caractères>.
<chiffre> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.
<suite de chiffres> ::= <> | <chiffre> <suite de chiffres>.
<probabilité> ::= 0 | 1| 0.<suite de chiffres>.
<lettre> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z.

L'analyseur se décompose en un analyseur lexical et en un analyseur sémantique.

- L'analyseur lexical détermine les automates d'états finis nécessaires à la reconnaissance des différentes unités lexicales d'un programme source.
- L'analyseur sémantique vérifie la validité du programme source en contrôlant les séquences d'unités lexicales. Il mémorise de plus les modèles de clauses ainsi reconnues.

Ces analyseurs ont respectivement été écrits à l'aide des outils lex et yacc, qu'UNIX propose.Le premier, lex, crée des analyseurs syntaxiques, le second, yacc, des analyseurs sémantiques. Ainsi, le programmeur qui utilise lex, spécifie les règles syntaxiques du langage, ainsi que les actions à exécuter lorsque ces règles sont vérifiées. lex génère alors un analyseur lexical qui peut coopérer avec yacc. yacc, yet another compiler compiler, associe quant à lui, à chaque règle de la grammaire du langage décrit, une action à exécuter lorsque la règle est appliquée. Les actions, en lex comme en yacc, sont écrites en langage C.

Le module construction de l'arbre construit l'arbre ET/OU en fonction d'une part des modèles de clause obtenus par l'analyse, d'autre part du but initial. En particulier, à chaque appel de prédicat, la fonction probabiliste qui simule le taux d'échec à l'unification est mise en œuvre.

Le module parcours gère le parcours de l'arbre selon le modèle pro-

posé. La gestion des primitives est réalisée par l'intermédiaire d'une file d'attente modélisée par une liste chaînée. Cette file d'attente est ordonnée en fonction des dates attachées aux primitives. Cette date représente l'instant de consommation de la primitive. A chaque unité de temps logique, la file d'attente est consultée afin de déterminer les éventuelles primitives à traiter. Chaque primitive ainsi sélectionnée est interprétée. De nouvelles primitives sont construites, datées et insérées dans la file d'attente. Les nœuds qui interprètent les primitives changent d'état.

L'utilisation de l'horloge logique permet de recueillir à tout instant l'état de chacun des nœuds de l'arbre ainsi que l'ensemble des primitives interprétées. Une trace détaillée du parcours est ainsi obtenue.

II.2.c.2. Structures de données.

Les différents modules décrits précédemment gèrent d'une part les nœuds de l'arbre et d'autre part les primitives. Chacune de ces entités est une structure de données que nous allons maintenant détailler.

La structure d'un nœud se compose des informations suivantes :

- l'identificateur du nœud
- le type du nœud (ET/OU)
- l'état du nœud (conformément au modèle décrit dans les chapitres II, III, IV)
- la probabilité éventuelle associée au nœud s'il est de type ET
- les liens père et fils.

La structure d'une primitive est quant à elle constituée des informations suivantes :

- l'identificateur de la primitive
- l'identificateur de l'émetteur de la primitive
- l'identificateur du destinataire de la primitive
- la date de consommation de la primitive : cette date est calculée en fonction de la date d'émission de la primitive, du temps de traitement de la primitive et du temps de communication de la primitive.

II.3. Jeux d'essai.

Dans le cadre de cette étude, nous nous sommes plus particulièrement intéressés à la structure de l'arbre ET/OU (c'est-à-dire largeur et profondeur), qu'à la sémantique associée. Trois classes de programmes ont été étudiées:

- *Classe 1* : des arbres ET/OU relativement peu profonds mais assez larges. Ces arbres correspondent aux programmes type base de données.

L'arbre généalogique fait partie de cette classe.

- *Classe 2* : des arbres ET/OU très profonds mais assez étroits. Ces arbres correspondent aux programmes récursifs, comme les traitements de liste par exemple.

- *Classe 3* : des arbres ET/OU regroupant les caractéristiques des deux précédents.

Les trois classes de programmes précédentes ont été testées avec des arbres, pour lesquels on a fait varier les trois paramètres suivants :

- le nombre de nœuds
- le nombre de niveaux
- le nombre de solutions

II.3.a. Les résultats obtenus sur la classe 1.

Les résultats observés sur des programmes de classe 1 sont présentés en figure 2. Par_Max représente le taux maximal d'activité obtenu à un instant du programme, Par_Moy la moyenne pondérée du taux d'activité.

Nb_Nds	Nb_Sols	Par_Moy	Par_Max	Tps_Exec	Nb_Mess
716	0	1.96	6	3129	2956
716	3	2.88	8	847	1022
716	16	2.65	9	1091	1239
716	40	2.61	9	1878	2139
716	60	2.29	11	6523	6348
716	192	2.2	11	10124	10622

figure 2.

On constate que le taux de parallélisme moyen est bas. Deux raisons expliquent ce phénomène. D'une part, l'arbre est peu profond (le nombre de niveaux est peu élevé), d'autre part le parallélisme OU n'apparaît que dans le bas de l'arbre, impliquant une terminaison rapide des processus OU.

On observe de plus un faible taux moyen de parallélisme lorsque le nombre de solutions est nul. Ceci est dû à l'importance du taux d'échec à l'unification. Celui-ci est entraîne la recherche de nouvelles solutions, mais peu de remontées de solutions. Les nœuds ancêtres restent ainsi plus longtemps en attente de solutions.

Le tableau de la figure 2 représente le parallélisme moyen obtenu en activant une seule branche en mode restreint, lors de la remontée de la première solution. Le tableau de la figure 3 décrit quant à lui l'évolution

de ce taux de parallélisme moyen, soulevée par l'activation en mode restreint de l'ensemble des fils d'un nœud (c'est-à-dire l'examen des alternatives, jusque là suspendues).

Nb_Sol	0	3	16	40	60	192
1 branche activée en mode restreint	1.96	2.88	2 .65	2.41	2.29	2.2
toutes les branches activées en mode restreint	1.99	2.9	2.67	2.45	2.33	2.21

figure 3.

On remarque une faible augmentation du parallélisme moyen, dont le manque de profondeur de l'arbre est responsable. En effet, le pipeline de remontées de solutions est lui très sollicité et les solutions restent bloquées dans l'arbre.

A l'inverse, le nombre de messages croît entre les deux exécutions. Cette augmentation du nombre de messages traduit le nombre d'activations en mode restreint, supplémentaires effectuées. La figure 4 donne le nombre d'activations en mode restreint dans les deux cas.

Nb_Sol	0	3	16	40	60	192
1 branche activée en mode restreint	168	39	64	124	271	564
toutes les branches activées en mode restreint	170	54	67	131	276	570

figure 4.

II.3.b. Les résultats obtenus sur la classe 2.

Les figures 5.a et 5.b. présentent les résultats respectivement obtenus sur des programmes de la classe 2 : dans le premier de ces tableaux, une clause contient un seul sous-but récursif, dans le second une des clauses contient deux sous-buts récursifs. On observe une légère augmentation du parallélisme moyen par rapport à celui généré par les programmes de classe 1. Toutefois, les étages du pipeline, même s'ils sont nombreux, ne sont pas suffisamment alimentés par les processus OU.

Premier jeu d'essais: un pipeline ET avec une branche récursive.

Nb Niveaux	Nb Solutions	Par_Max	Par_Moy
20	2	5	3,5
20	4	6	3,3
20	14	10	2,82

figure 5.a.

Deuxième jeu d'essais: un pipeline ET avec deux branches récursives.

Nb Niveaux	Nb Solutions	Par_Max	Par_Moy
12	4	6	3,31
12	6	10	2,77
12	22	11	1,96
16	11	10	2,83
18	17	12	3,1

figure 5.b

II.3.c. Les résultats obtenus sur la classe 3.

L'étude a été réalisée en fixant le nombre de solutions, et en faisant varier soit le nombre de nœuds, soit le nombre de niveaux.

II.3.c.1. Le nombre de nœuds varie.

La figure 6 montre l'évolution du parallélisme moyen en fonction du nombre de nœuds de l'arbre. Chaque courbe correspond à un nombre donné de solutions.

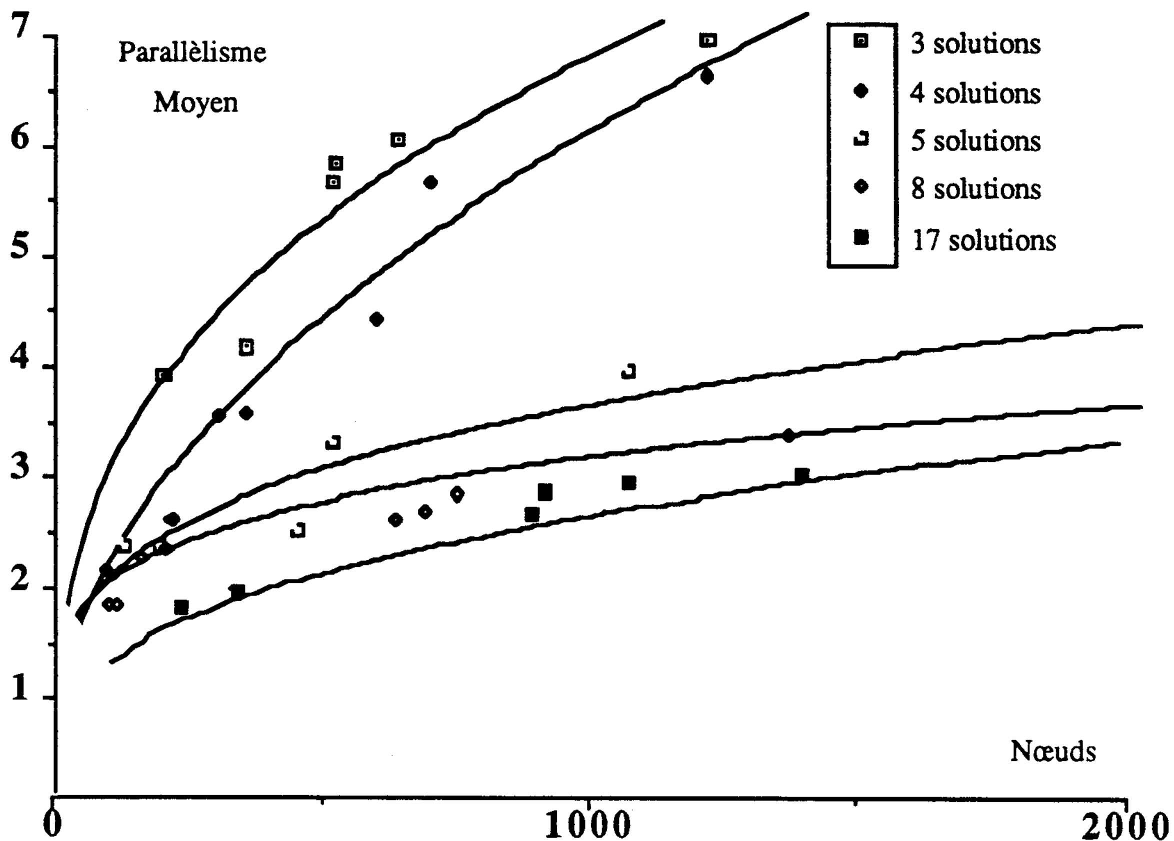


figure 6.

Elle suscite les commentaires suivants :

- Pour un nombre de nœuds fixé, l'augmentation du nombre de solutions, entraîne une diminution du taux moyen de parallélisme.

-Pour un nombre donné de solutions, l'accroissement du nombre de nœuds s'accompagne d'une montée du taux de parallélisme moyen.

II.3.c.2. Le nombre de niveaux varie.

Les courbes présentées en figure 7 donnent l'évolution du taux moyen de parallélisme en fonction du nombre de niveaux. Une courbe correspond à un nombre fixe de solutions.

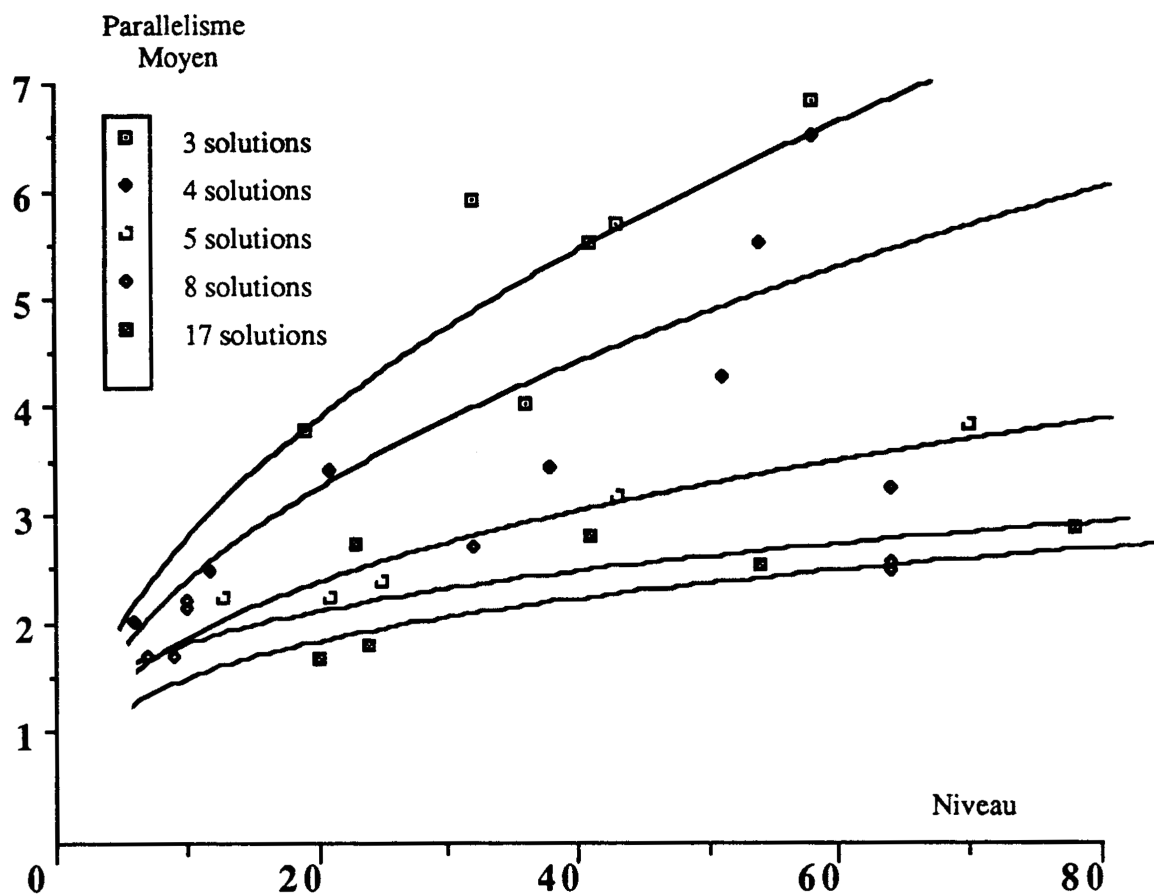


figure 7.

Par simple lecture de cette courbe, on constate que :

- Pour un nombre de niveaux fixé, l'augmentation du nombre de solutions entraîne une diminution du taux moyen de parallélisme.
- De même, lorsque le nombre de solutions est invariant, une augmentation du nombre de niveaux entraîne une montée du taux de parallélisme moyen.

Il apparaît donc que le taux de parallélisme moyen développé lors de l'exécution de cette classe de programmes, est plus important que celui obtenu lors de l'exécution des programmes des classes précédentes. De même, plus le nombre de solutions retournées est faible, plus le taux de parallélisme moyen augmente. En effet, si le nombre de solutions est important, le pipeline de remontée des solutions est très sollicité. La remontée des solutions s'effectue à la cadence de consommation des solutions par le nœud racine de l'arbre. Les solutions évaluées dans le bas de l'arbre y séjournent donc plus longtemps, car elles sont bloquées par les verrous de solutions.

III. SIMULATION REELLE.

III.1. Principe.

Une observation de l'exécution de programmes Prolog réels a été menée, dans le cadre d'un mémoire de D.E.A, par F. Boulier [Boul 90]. Cette observation est réalisée à partir d'une trace de l'exécution de programmes Prolog dont l'intérêt principal réside dans le suivi de la progression des solutions dans l'arbre ET/OU.

Cette trace est obtenue en insérant des appels à un prédicat, dit prédicat espion, entre les littéraux du corps de clause à exécuter. Les trois arguments, dont le calcul est présenté dans le paragraphe suivant, de ce prédicat espion sont respectivement le numéro de la clause (CID), le numéro d'ordre du littéral dans le corps de clause (ORD), le numéro du nœud OU courant (NOD). La trace d'un programme est donc constituée de la suite de triplets (CID, ORD, NOD).

Ces triplets sont assimilés à des messages émis par un nœud OU vers un autre nœud OU. Les nœuds ET sont implicitement identifiés par les couples (CID, ORD). Chacun des messages circule en une unité de temps. Le modèle supposant la circulation simultanée de plusieurs messages, l'activité du programme est mesurée en nombre de messages par unité de temps. Un histogramme représente cette activité.

III.2. Présentation informelle du calcul des triplets (CID, ORD, NOD).

Nous allons décrire de manière informelle la transformation d'un programme Prolog. En particulier, nous allons détailler le calcul de chacun des arguments CID, ORD,ORD.

III.2.a. Calcul de l'argument CID.

Les clauses du programme sont numérotées à partir de zéro dans l'ordre où elles sont rencontrées. Le but initial reçoit le numéro d'ordre maximum. Le CID d'une clause est donc le numéro d'ordre attaché à la clause.

III.2.b. Calcul de l'argument ORD.

L'argument ORD est lui aussi calculé lors de la transformation du programme. Il est initialisé à zéro et incrémenté entre chaque appel au prédicat espion. Sa valeur finale est forcée à 99 . Toutes les clauses y compris les faits, subissent cette transformation.

Ainsi, la règle p(X) :- q(X), r(X) est réécrite en la règle :

```
p(X) :- espion(      ,ORD=0,      ),
        q(X) espion(   ORD=1    ),
        r(X) espion(  ORD = 99  ).
```

et le fait r(c) est réécrit en

```
r(c) :- espion(   ORD=0  ), espion(   ORD=99   ).
```

III.2.c. Calcul de l'argument NOD.

Le calcul de l'argument NOD est plus délicat à gérer. En effet, l'argument NOD représente le numéro du nœud OU courant. Il est donc évalué non pas lors de la transformation du programme mais lors de l'exécution du programme. Il est calculé lors de l'appel du littéral qui a donné naissance au nœud. L'arité du littéral est alors modifiée afin de prendre NOD comme argument. L'argument NOD du but initial est égal à zéro.

La règle p(X) :- q(X), r(X) implique la création de deux nœuds OU. Le premier d'entre eux correspond à l'évaluation de q, le second à celle de r. Il faut donc générer deux arguments NOD et modifier en conséquence l'arité des prédicats q et r.

L'arité du prédicat a été modifiée préalablement. La règle p(X) :- q(X), r(X) devient donc :

```
p(X,NOD0) :- générer(NOD1), q(X,NOD1), générer(NOD2), r(x,NOD2).
```

La transformée finale de la règle p(X) :- q(X), r(X) est donc (si la règle est la première du programme à transformer)

p(X,NOD) :- espion(0,0,NOD), générer(spy,NOD1), q(X,NOD1), espion(0,1,NOD), générer(spy,NOD2), r(X,NOD2), espion(0,99,NOD).

III.3. Résultats.

Cette seconde simulation a donc permis de tester le comportement de programmes réels exécutés suivant le modèle proposé. Les résultats ont ensuite été comparés à ceux obtenus lors de l'exécution de programmes selon un modèle qui exploite un parallélisme OU massif, c'est-à-dire un parallélisme OU déclenché par le haut de l'arbre de résolution.

Les résultats sont donnés sous forme de deux histogrammes qui représentent le nombre de messages par unité de temps. Ces histogrammes montrent respectivement le comportement de programmes Prolog réels, exécutés d'une part selon le modèle activé restreint, et d'autre part selon le modèle exploitant le parallélisme OU massif. Les programmes des classes 2 et 3 ont été plus particulièrement étudiés.

Ainsi, un programme de permutations de listes représente les programmes de la classe 2, le problème des quatre reines ceux de la classe3.

III.3.a. Etude des programmes de classe 2.

Nous concentrerons notre étude sur le programme suivant :

```
permuter([],[]).
permuter([E|L], R) :- permuter(L, Ri), insérer(E,Ri,R).
insérer(E,L,[E|L]).
insérer(E,[X|L],[X|R]) :- insérer(E,L,R).
```

dont le but initial est :- permuter([a,b,c,d,e], L).

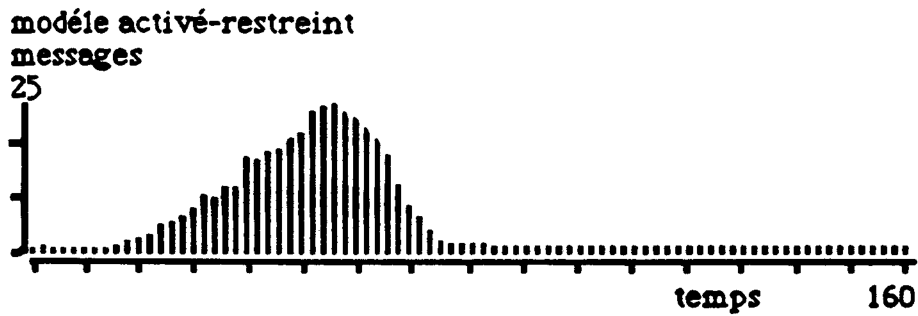
On remarque d'une part que deux des règles sont récursives et d'autre part que les règles permuter sont exclusives l'une de l'autre. L'exécution du programme entraîne donc en premier lieu un parcours séquentiel de la liste, suivi de la construction des listes, qui sont les permutations de la liste [a,b,c,d,e]. Le trait plat observé dans chacun des histogrammes suivants (figure 8) exprime le caractère séquentiel du parcours. L'appel du prédicat insérer engendre alors le parallélisme.

Comme le montre la figure 8, on constate que :

- Le parallélisme maximal du modèle LOGARCH est inférieur au parallélisme maximal d'un modèle exploitant le parallélisme OU massif. Ceci s'explique par le nombre d'activations simultanées effectuées à chaque étape de résolution (deux en l'occurrence), activations qui impliquent par ailleurs l'évaluation parallèle de solutions dont l'ordre n'est pas garanti.

- Le premier de ces histogrammes se termine par un trait plat (dont la durée est en l'occurrence de 85). Ce trait plat traduit le temps nécessaire à la remontée ordonnée des 120 solutions, qui double le temps d'exécution (par rapport à l'évaluation de la dernière solution, évaluation qui survient au temps 75).

Nombre de messages échangés : 1203



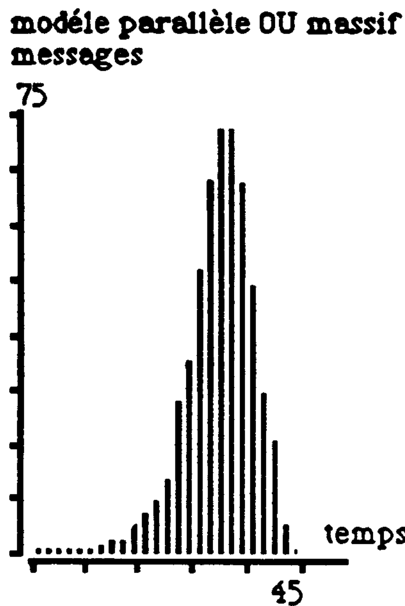


Figure 8

III.3.b. Etude des programmes de classe 3.

Le programme étudié est celui des quatre reines, qui ne donne que deux solutions.

```
reines(Taille_du_damier, Solutions) :-
        résoudre(Taille_du_damier, [],Solutions,0).

résoudre(Bs, L, L, Bs) :- !.
résoudre(Taille du damier, Courante, Finale, Colonne) :-
      Nouvelle_Colonne is Colonne+1,
      index(Nouvelle_Colonne, Taille_du_damier),
      sauve(Courante, Nouvelle_Colonne, Nouvelle_Ligne),
      résoudre(Taille_du_damier,
         [reine(Nouvelle_Colonne,Nouvelle_Ligne)|Courante],
         Finale,
         Nouvelle_Colonne).
```

```
index(Taille, Taille).
index(N,Taille) :- S1 is Taille-1, S1>0,index(N,S1).

sauve([],_,_).
sauve([reine(I,J)|L],X,Y) :- not menacée(I,J,X,Y), sauve(L,X,Y).

menacée(I,_,I,_) :- !.
menacée(_,J,_,J) :- !.
menacée(I,J,X,Y) :- U is I-J, U is X-Y, !.
menacée(I,J,X,Y) :- U is I+J, U is X+Y.
```

Le but initial de ce programme est ? :- reines(4, L).

La figure 9 présente les deux histogrammes correspondants.

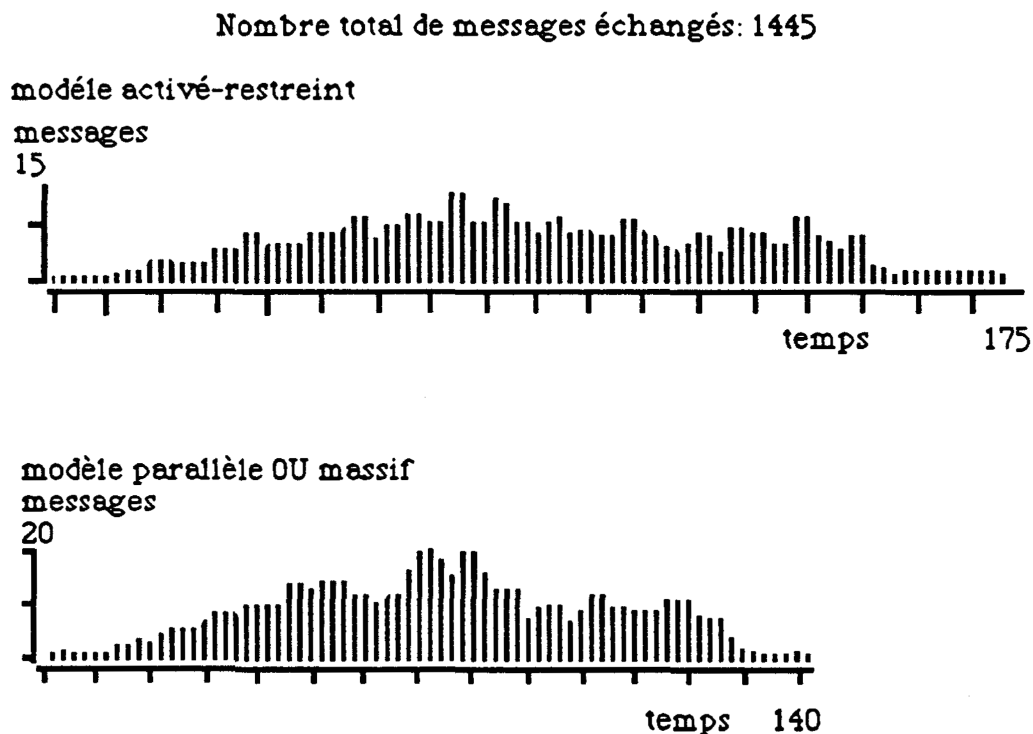


Figure 9

On remarque que si le taux de parallélisme maximal reste inférieur dans le modèle LOGARCH, l'écart relatif entre les deux pics est nettement moins important que dans la classe de programmes précédente. De 66%, il passe à 25%.Les temps d'exécution des programmes selon les deux modèles restent quant à eux comparables. Certes, le pipeline de remontée de solutions existe toujours mais il est nettement réduit.

Il apparaît donc que pour ce type de problème à la fois récursif et non déterministe, élaborant peu de solutions, notre modèle fournit des performances comparables à celles du modèle OU massif, tout en respectant l'ordre classique de la production des solutions.

IV. CONCLUSION.

La simulation abstraite a permis par l'examen des fichiers trace, de vérifier la validité du modèle. Ainsi, le parcours de l'arbre est-il correct.

Les deux simulations, abstraite et réelle, ont mis en évidence l'importance de l'une des contraintes initiales, à savoir le respect de l'ordre des solutions. Ainsi, plus le nombre de solutions élaborées par le programme est élevé, moins les performances du modèle sont bonnes. La simulation abstraite souligne une diminution du taux de parallélisme moyen pour un nombre important de solutions, la simulation réelle montre quant à elle, l'allongement du temps d'exécution par rapport à un modèle exploitant le parallélisme OU massif.

Ces deux simulations ont de plus dégagé une classe de programmes à laquelle notre modèle est bien adapté : il s'agit de programmes dont la taille est relativement importante (donc de programmes présentant suffisamment de non-déterminisme) et qui élaborent un nombre peu important de solutions.

CONCLUSION

CONCLUSION

Nous avons défini un modèle d'évaluation parallèle de Prolog qui exploite un parallélisme OU restreint. Les principales caractéristiques de ce modèle sont un mode de contrôle dirigé par la nécessité, un respect de la sémantique opérationnelle de Prolog et un mécanisme de verrouillage qui contrôle la production et la consommation des solutions de l'arbre ET/OU. La conjonction de ces trois caractéristiques rend possible l'exécution de programmes Prolog déjà existants, sans ajout d'annotations supplémentaires, tout en assurant une extraction automatique du parallélisme.

Les deux simulations ont montré une limitation du degré de parallélisme due à la forte contrainte qu'est le respect de la sémantique opérationnelle de Prolog (cette contrainte permet également de construire des ensembles de solutions identiques à ceux obtenus lors de l'exécution de programmes Prolog selon un modèle d'évaluation séquentiel). A l'inverse, ces deux simulations ont souligné les performances du modèle, réalisées lors de l'exécution des programmes de taille conséquente, et qui évaluent peu de solutions. Ces performances sont comparables à celles établies par les modèles exploitant un parallélisme OU sans contrainte.

Il s'avère maintenant nécessaire de poursuivre l'enrichissement du modèle proposé. Ainsi, la prise en compte de prédicats tels que le prédicat Not , les prédicats Assert et Retract sera étudiée. Si l'insertion du prédicat Not, qui traduit la négation, semble relativement simple de par l'existence des prédicats Cut et Fail, celle des prédicats Assert et Retract est plus délicate. En effet, ces deux prédicats modifient, en cours d'exécution, la base de règles (en ajoutant, voire supprimant, des règles à cette dernière) et par conséquent la forme de l'arbre ET/OU. Une première approche consiste d'une part à identifier les prédicats susceptibles d'être concernés par un Assert (voire un Retract), à déclarer ces prédicats dynamiques (les autres étant statiques) et d'autre part à suspendre l'exécution des prédicats dynamiques lorsqu'elle survient sur une branche activée en mode restreint. L'exécution de ces prédicats reprend au moment où la branche devient la branche active. Cette approche suppose que les arguments des prédicats Assert et Retract sont connus. L'introduction d'autres prédicats, tels que le prédicat Bagof peut être ensuite considérée.

De plus, une implantation du modèle sur une architecture donnée est envisagée. Cette implantation soulève certains problèmes, dont les plus importants sont relatifs à la gestion des environnements et au scheduling. Des travaux sur la gestion d'environnements sont d'ailleurs actuellement menés par un autre chercheur. D'autre part, quelque soit l'architecture choisie, le nombre de processus créés lors de l'exécution d'un programme tend à être nettement supérieur au nombre de processeurs. Une étude préliminaire, effectuée par N. Bennani [Benn 90] dans le cadre d'un mémoire de D.E.A, propose d'associer à chaque processus un degré de priorité et d'allouer les processeurs aux processus en fonction de leur degré de priorité. La méthode proposée repose sur une numérotation des environnements, construits au fur et à mesure de l'exécution du programme. Les numéros d'environnements sont transmis dans les messages d'activation, lors de la phase d'activation. Lors de la phase de résultats, les numéros d'environnements sont redistribués aux branches OU activées en mode restreint. Une phase de renumérotation partielle des environnements succède à chaque évaluation d'une solution complète (c'est-à-dire à chaque réception d'une solution par la racine de l'arbre ET/OU). L'inconvénient majeur de cette méthode réside en la possibilité de l'attribution d'un numéro identique à deux environnements distincts. Lorsque deux processus à exécuter ont deux numéros identiques, le scheduler choisit l'un d'entre eux aléatoirement.

Le modèle que nous avons proposé a été conçu dans le respect de la sémantique opérationnelle de Prolog (sélection de la première clause du paquet, sélection du premier littéral du corps de clause). Une adaptation du modèle pour d'autres stratégies pourra à l'avenir être étudiée.

REFERENCES BIBLIOGRAPHIQUES

PUBLICATIONS ET COMMUNICATIONS.

"A new parallel evaluation scheme for Prolog programs : an example."
G. Goncalves, I. Hannequin, P. Lecouffe, B. Toursel.
MIMI'89 -Zurich - 26-29 juin 89.

"Prolog : A new parallel evaluation scheme for Prolog programs."
G. Goncalves, I. Hannequin, P. Lecouffe, B. Toursel.
Euromicro Congress - Cologne - sept 89.
Microprocessing and Microprogramming 27 (1989) pp 391-396.

"Exécution de programmes Prolog sur un réseau de processeurs."
G. Goncalves, I. Hannequin, B. Toursel.
Journées AFCET- GROPLAN : "développement de programmes pour machines parallèles".
Chamonix -17-19 mai 1989 et publication dans Bigre+Globule.

"Deep first OR parallelism in the Log-arch project."
H. Bourzoufi, G. Goncalves, I. Hannequin, P. Lecouffe, B. Toursel.
International Conference on Parallel and Distributed Computing and Systems. New_york. October 1990.

"Une nouvelle exécution OU-parallèle de Prolog sur la machine Log-Arch."
H. Bourzoufi, G. Goncalves, I. Hannequin, P. Lecouffe, B. Toursel.
Publication interne à paraitre.

[Apt 82] K.R. Apt et M.H. Van Emden .
Contribution to the theory of logic programming
Journal of the Association for Computing Machinery, Vol. 29, n° 3, 1982, pp. 841-862.

[Arna 89] P. Arnaud.
Proposition d'une méthode de répartition de la charge sur un réseau de processeurs: conséquences sur la topologie et simulation.
Thèse de Doctorat Informatique, LIFL, Université de Lille I, Nov. 1989.

[Baro 88] U. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J.C. Syre, H. Wesphal.
The parallel ECRC Prolog System PEPSys : An overview and evaluation results.
Proceedings FGCS'88, Chicago, August, 1988.

[Benn 90] N. Bennani.
Gestion de la priorité d'exécution des clauses d'un programme PROLOG sur la machine LOG-ARCH.
Mémoire de D.E.A. L.I.F.L.1990.

[Boiz 89] P. Boizumault.
Prolog l'implantation.
Edition Masson.1989.

[Boul 90] F. Boulier.
Parallélisme en Prolog.
Mémoire de D.E.A. L.I.F.L.1990.

[Boye 72] R.S. Boyer, J.S. Moore.
The sharing of structure in theorem-proving programs
in Machine Intelligence 7, pp 101-116, Edinburgh University Press, 1972.

[Bria 90] J. Briat, M. Favre, C. Geyer.
OPERA : OU Parallélisme et Régulation adaptative en Prolog.
Tregastel, 1990.

[Brun 76] M. Bruynooghe.
An interpreter for predicate logic programs : basic principles.
Report Cw10, Kathilieke Universiteit Leuven, Belgium, 1976.

[Brun 80] M. Bruynooghe.
The memory management of Prolog implementations
in Workshop'80, pp 12-20, 1980.

[Chas 89a] J. Chassin de Kergommeaux, P. Codognet, P. Robert, JC. Syre.
Une programmation logique parallèle : les langages gardés.
TSI 1989.

[Chas 89b] J. Chassin de Kergommeaux, P. Codognet, P. Robert, JC. Syre.
Une programmation logique parallèle : Systèmes non déterministes.
TSI, 1989.

[Clar 81] K.Clark, S. Gregory.
A Relational Language for Parallel Programming.
In proc. Conference on Functional Programming Languages and Computer Architecture. ACM, 1981, pp 171-178.

[Clar 84] K.Clark, S. Gregory.
Parlog: parallel programming in logic.
Technical Report DOC 84/4, Imperial College London, April, 1984.

[Clar 85] K.Clark, S. Gregory.
Notes on the implementation of Parlog.
Journal of logic programming.1, 1985, pp 17-42.

[Colm 82] A. Colmerauer
Prolog II, Manuel de référence et modèle théorique.
G.I.A. Fac des Sciences de Luminy, Marseille, 1982.

[Cond 86] M. Condillac.
Prolog : Fondements et Applications
Dunod, 1986.

[Cone 81] J.S. Conery, D.F. Kribler.
Parallel interpretation of logic programs.
Proc. conf on functionnal programming languages and computer architecture. ACM October, 1981, pp 163-170.

[Cone 83] J.S. Conery, D.F. Kribler.
And-Parallelism in logic programs.
IJCAI, pp 539-543, August 1983.

[Cram 86] J. Crammond.
An Execution Model for Commited-Choice Non deterministic Languages. In Proc. 3rd Symposium on Logic Programming, London, July 1986, pp 283-297.

[Degr 84] D. Degroot.
Restricted And-Parallelism.
Proc of FGCS, ICOT, November 1984, pp 471-478.

[Degr 87] D. Degroot.
Restricted And-Parallelism and Side Effects.
In 4th Symposium on Logic Programming, pages 80-89. San Francisco, Sept., 1987.

[Deve 90] N. Devesa
Proposition d'un schéma d'évaluation parallèle du lanagage FP sur un réseau de processeurs.
Thèse de doctorat informatique. L.I.F.L. Université de Lille I. Jan. 1990.

[Dura 86] I. Durand.
Un modèle d'interprétation Répartie pour une Architecture Multiprocesseur Prolog.
Thèse de Doctorat, Université P.Sabatier Toulouse, Oct 1986.

[Gonc 88a] G. Goncalves, M.P. Lecouffe, B.Toursel.
A distributed associative network for a parallel reduction architecture.
IFIP WG103, Working Conference on Parallel Processing, Pisa (Italy), 25-27 Apr. 1988.

[Gonc 88b] G. Goncalves, M.P. Lecouffe, B.Toursel.
A parallel reduction machine with a distributed content adressable memory.
MIMI 88, San Feliu (Espagne), 27-29 June 1988.

[Herm 86] M. V. Hermenegildo.
An Abstract Machine for Restricted AND-parallel execution of logic programs.
In 3rd Int. Conf. on Logic Programming, pages 25-39. London, July, 1986.

[Ichi 87] N. Ichiyoshi, T. Miyazaki, K. Taki.
A distributed implementation of Flat GHC and the Multi-Psi.
In 4th Int Conference on logic programming, Melbourne, May 1987, pp 257-274.

[Ito 83] N. Ito, R. Onai, K. Masuda, H. Shimizu.
Prolog Machine based on the data-flow mechanism.
Technical Report - tm -0007, ICOT, May, 1983.

[Kano 82] H. Kanoui
Manuel d'exemples de Prolog II
G.I.A. Fac. des Sciences de Luminy, Marseille, 1982

[Kowa 79] R. Kowalski.
Logic for Solving Problem.
The computer Science Library, Elsevier, 1979.

[Levy 86] J. Levy.
Shared memory execution of commited-choice languages.
In Proc. 3rd Symposium on Logic Programming, London, July 1986, pp 298-312.

[Lloy 86] J.W. Lloyd
Fundations of logic Programming.
2nd Extended Edition, Springer Verlag, 1986.

[Mart 82] M. Martin
Interpréteur Prolog en Pascal sous Multics
in [Cnet82], pp 10-19, 1982.

[Mart 83] M. Martin
Les nouvelles possibilités de Prolog/Pascal Multics
in [Cnet83], pp 37-45,1983.

[Mell 80] C.S. Mellish.
An alternative to structure-sharing in the implementation of a Prolog interpreter
in [Worksshop'80], pp 21-32, 1980

[Niar 89] S. Niar.
Contribution à l'étude des architectures d'ordinateurs parallèles. Structure de la machine N-ARCH et émulation sur un réseau de transputers.
Thèse de doctorat informatique, Université de Lille I, Oct. 1989.

[Nico 91] J.C. Nicolas.
Machines de bases de données parallèles. Contribution aux problèmes de la fragmentation et de la distribution.
Thèse de doctorat informatique, Université de Lille I, Jan. 1991.

[Perc 86] C. Percebois, I. Futo, I. Durand, C. Simon, B. Bonhoure.
COALA : Un Calculateur Orienté Acteur pour la Logique et ses Applications.
Journées AFCET_GROPLAN : Architectures de Machines, Gien, 29-31 Janvier 1986.

[Robi 65] J.A. Robinson
A machine oriented logic based on the resolution principles.
Journal of the ACM 12, pp 23-44, 1965.

[Sato 87] M. Sato and all.
KL1 execution model for PIM_cluster with shared memory.
4th, Int Conf on logic programming, Melbourne, May 1987, pp 339-355.

[Shap 83] E.Y. Shapiro.
A subset of Concurent Prolog and its interpreter.
Technical Report. Weizman Institute. Revohot, February 1983.

[Shap 87] E.Y. Shapiro, (Editor).
Concurent Prolog: Collected Papers, Vols. 1 and 2, MIT Pres, 1987.

[Syre 87] J.C Syre, P. Robert, J. Chassin de Kergommeaux et l'équipe PEPSys.
Le système logique parallèle PEPSys.
S Bourgault, M Dincbas (ed)., Programmation en logique, 7ème séminaire, Trégastel, May 1988, pp 425-454.

[Ueda 85] K. Ueda.
Guarded Horn Clauses,Ph.D. Thesis.
Information Engineering Course,University of Tokyo, Tokyo, 1986.

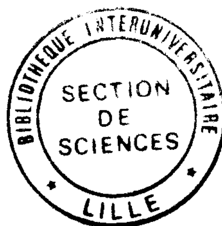
[Ueda 86] K. Ueda.
Guarded Horn Clauses.
Technical Report TR-102, ICOT, Tokyo, June 1985.

[VanC 82] Van Caneghem M.
Manuel d'utilisation de Prolog II
G.I.A. Fac. des Sciences de Luminy, Marseille, 1982

[Warr 77] D.H.D. Warren.
Implementing Prolog : compiling predicates logic programs.
D.A.I. Research report n°39/40, univ of Edinbugh,1977.

[Warr 83] D.H.D. Warren.
An abstract prolog instruction set.
Technical report 309, SRI, October 1983.

[Warr 87] D.H.D. Warren.
The SRI Model for OR parallel execution of Prolog. Abstract design and Implementation Issues.
4th Symposium on logic programming, San Francisco, Sept 1987, pp 46-53.


BIBLIOTHEQUE INTERUNIVERSITAIRE
SECTION DE SCIENCES
LILLE